THE UNIVERSITY OF CHICAGO

ADAPTIVE FRAMEWORK FOR CONFIGURATION TUNING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

SHU WANG

CHICAGO, ILLINOIS
DECEMBER 2021

Dedicated to my beloved family.

“The only real mistake is the one from which we learn nothing.” – Henry Ford.

Table of Contents

LIST OF FIGURES vii

LIST OF TABLES viii

ACKNOWLEDGMENTS ix

ABSTRACT xi

1 INTRODUCTION 1
1.1 Motivation 1
1.2 Contributions 4
1.3 Dissertation Organization 7

2 BACKGROUND AND RELATED WORK 8
2.1 Control theory 8
2.1.1 Traditional Controller 8
2.1.2 Generalized Controller for Software 9
2.2 Misconfiguration 11
2.3 Automatic Configuration Tuning 11
2.3.1 Control theoretic frameworks 12
2.3.2 Machine Learning 14
2.4 Adaptive Control 14

3 UNDERSTANDING PERF-SENSITIVE CONFIGURATIONS 16
3.1 Methodology 16
3.2 Findings 17
3.2.1 How Common are PerfConf Problems? 17
3.2.2 What are PerfConfs' Impact? 18
3.2.3 How to Set PerfConfs? 19
3.3 Summary 20

4 *SMARTCONF* 21
4.1 *SmartConf* Overview 21
4.2 *SmartConf* Framework 22
4.2.1 Developers' effort 23
4.2.2 Handling Special Configuration Types 24
4.2.3 Users' Effort 26
4.3 *SmartConf* Controller Design 27
4.3.1 How to Decide the Pole Parameter 28
4.3.2 Handle Hard Goals 28
4.3.3 Handle Configurations with Indirect Impact 30
4.3.4 Handle Multiple, Interacting PerfConfs 31

4.3.5 Other Implementation Details . . . 32
4.3.6 Formal Assessment and Discussion . . . 34
4.4 Evaluation . . . 35
4.4.1 Evaluation methodology . . . 35
4.4.2 Does *SmartConf* Satisfy Constraints? . . . 36
4.4.3 Does *SmartConf* Provide Good Tradeoffs? . . . 39
4.4.4 Alternative Design Choices . . . 40
4.4.5 Other results . . . 42
4.4.6 Limitations of *SmartConf* . . . 43
4.5 Conclusions . . . 43

5 *AGILECTRL* . . . 45
5.1 Introduction . . . 45
5.2 Background . . . 49
5.2.1 Self-Adaptive Framework . . . 49
5.2.2 Control-based Self-adaptive Frameworks . . . 50
5.3 Motivating Example . . . 52
5.3.1 Different Run-time Resources . . . 53
5.3.2 Different Run-time Workloads . . . 55
5.4 *AgileCtrl* Design . . . 56
5.4.1 α Sign Module . . . 57
5.4.2 α Value Module . . . 58
5.4.3 Virtual Goal (vg) Module . . . 62
5.5 Evaluation . . . 62
5.5.1 Evaluation Methodology . . . 62
5.5.2 Module Evaluation . . . 65
5.5.3 Case Study . . . 67
5.5.4 Limitations of *AgileCtrl* . . . 74
5.6 Conclusions . . . 75

6 CONCLUSIONS AND FUTURE WORK . . . 76
6.1 Contributions . . . 76
6.2 Limitation and Future Work . . . 77

REFERENCES . . . 79

List of Figures

1.1 An overview of *SmartConf* and *AgileCtrl* presented in this dissertation. 5

2.1 Traditional Controller 8

4.1 Using a controller to adjust PerfConf (gray parts are extra controller-related components in *SmartConf*.) 22
4.2 *SmartConf* configurations 23
4.3 *SmartConf* class 24
4.4 *SmartConf* sub-class 25
4.5 Trade-off performance comparison. Normalized upon the best-performing static configuration; x: fail the perf. constraint. The numerical PerfConf settings are above each bar. 38
4.6 *SmartConf* vs. static optimal on HB3813. workload changes at ∼200s. Throughput is accumulative. 38
4.7 *SmartConf* vs. alternative controllers. 40
4.8 *SmartConf* adjusts two related PerfConfs. 41

5.1 HD4995 under different CPU resources 53
5.2 HB3813 under different workloads(1MB, 2MB, 5MB, and 10MB) 54
5.3 Agile Controller 56
5.4 The statistics difference between OLR and *AgileCtrl*. For OLR, we checked the percentage of getting an α with opposite sign, a conservative α (x5 larger than optimal), an aggressive α (x5 smaller than optimal), and remaining as a proper α. For *AgileCtrl*, we checked the percentage of accurate estimate/overestimate/underestimate predicted α against the optimal. 60
5.5 Both Initial α's value and initial virtual goal are 10x different from ideal setting, and initial α's sign is also flipped. All modules of *AgileCtrl* are enabled and able to fix wrong α and virtual goal for HB3813 71
5.6 Both Initial α's value and initial virtual goal are 10x different from ideal setting, however the α's value and sign are both flipped around time 1 second. All modules of *AgileCtrl* are enabled and able to fix wrong α and virtual goal for HB3813 . . 72
5.7 HB3813 with workload of 10MB, and a comparison on different approaches in α Value Module 73
5.8 *AgileCtrl* with different initial alpha 0.1 and 10 while ideal alpha is 2.4 74

List of Tables

1.1 Traditional configuration vs *SmartConf* configuration 5

3.1 Empirical study suite 16
3.2 Different types of PerfConf patches 18
3.3 How a PerfConf affects performance (one PerfConf can affect more than one metric) 19
3.4 How to set PerfConfs 19

4.1 Benchmark suite and runt-time setting for module evaluation. In issue description, the main constraint that users complain about is put earlier, the trade-off description is later, and finally the metrics of constraint performance (goal) and trade-off performance. 33
4.2 Lines of code changes for using *SmartConf* 42

5.1 Partial AdapConfs used in self-adaptive frameworks (**E**: explicit AdapConf and **I**: implicit AdapConf) 51
5.2 Benchmark suite and runt-time setting for module evaluation. In issue description, the main constraint that users complain about is put earlier, the trade-off description is later, and finally the metrics of constraint performance (goal) and trade-off performance. 63
5.3 α Sign Module Evaluation (**Goal**: the normalized main constraint performance *w.r.t* the goal (close to 1 means better). **Tradeoff**: the trade-off performance speedup *w.r.t SmartConf* (the high, the better).) 66
5.4 Overall trade-off performance speedup *w.r.t* to ACif constraint performance converged to goal(The higher, the better.). Otherwise, system crashed(**C**) or constraint performance oscillate around the goal(**O**). 68
5.5 Overall Error Tolerance for *AgileCtrl* compared with alternative approaches (ET_l/ET_h: the lowest/highest alpha that corresponding approach can correct without causing performance oscillations or system crashes. **0**: No such alpha without causing performance oscillations or system crashes) 69
5.6 Virtual Goal(vg) Module Evaluation of trade-off performance speedup *w.r.t SmartConf*. 69

ACKNOWLEDGMENTS

I am grateful to all the people who helped me during my Ph.D. journey at the University of Chicago.

Foremost, I would like to express my sincere gratitude to my Ph.D. supervisor, Professor Shan Lu, for her continuous support during my Ph.D. study and research. I greatly benefited from her deep knowledge as well as novel research ideas. Without her, this dissertation would not exist. Despite her knowledge, I am also deeply inspired by her enthusiasm and hardworking towards academic excellence. This spirit encourages me to keep an optimistic attitude with hard work and perseverance, as I climb the "steep paths" during my Ph.D. journey. Most importantly, Shan also taught me how to be a better person. Shan is more than my academic supervisor but a mentor for life.

I am also sincerely grateful to Professor Henry Hoffmann during our collaboration. Henry is always kind and helpful whenever I encounter research problems. He provided me with excellent suggestions for how to apply control theory in software. His kindness encourages me to work with patients.

I would also like to thank Professor Haryadi S. Gunawi for being on the committee with guidance and feedback. I am extremely fortunate to have Haryadi as both my MS Exam and Ph.D. Defense committee. Specifically, Haryadi provide me with detailed feedback during my MS Exam.

Thanks go to all members of Shan's research group. I want to thank Chi Li and William Sentosa for their help with *SmartConf* project. I want to thank Yuxi Chen for leading our transactional memory project. I want to thank Junwen Yang, Haopeng Liu, Guangpu Li, Chengcheng Wan, Lefan Zhang, Bogdan Stoica, and Utsav Sethi for discussing the research problems and making our group an excellent environment.

I would also thank Kate Keahey to have me as an intern at Argonne National Laboratory(ANL). At Argonne National Laboratory(ANL), I learned more cloud computing

infrastructure as well as the importance of reproducibility.

I would like to thank my friends Xuefeng Liu, Huan Ke, and Xiaoan Ding for appearing in my Ph.D. journey and helping me whenever I needed to. Your appearances made my Ph.D. journey more colorful and memorable. Thanks go to Fan Yang for not only the best CS department logo ever but also her perpetual spirit, which will be remembered by all of us. Thanks Fan Ye and Shichao Lu. Thank you Yunting Tao. I also want to thank myself for staying optimistic about my life and doctors from UChicago hospital, as I am suffering from chronic cough over the last one and half years during my Ph.D. journey.

Finally, I would love to thank my dear family for their selfless and unconditional love. Thanks for always being there and having my back. Thank you, Dad and Mom! For this, I want to dedicate this dissertation to them.

ABSTRACT

Modern software systems are often equipped with hundreds to thousands of configuration options, many of which greatly affect performance. However, determining the best configuration is difficult, both because it requires some knowledge of software internals, and often the best configuration changes due to unpredictable changes in workload or operating environment. Among all types of configurations, performance-sensitive configurations (PerfConfs) are challenging to set because they represent tradeoffs; *e.g.* , between memory usage and response time.

Aiming for automatic configuration tuning and improve the modern software performance and reliability, this dissertation works on these three parts and makes the following contributions:

First, this dissertation conducts an empirical study to understand performance-sensitive configurations and the challenges of setting them in the real-world. We look at 80 developer-patches and 54 user-posts concerning PerfConfs in 4 widely used large-scale systems. The study reveals several main findings: (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*.

Second, 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.

Third, observed that existing self-adaptive approaches are required to configure a new set of internal configurations (AdapConfs), we take a different approach: building self-adaptive software with as few externally configurable components as possible. Specifically, we create a framework—called *AgileCtrl*—by extensively modifying an existing control-theoretic framework for self-adaptive software. *AgileCtrl* monitors the quality of its adaptations and reconfigures its own internals to provide even greater robustness in the face of user error or unexpectedly volatile environments. We evaluate *AgileCtrl* by comparing against other self-adaptive frameworks that require careful human tuning. Across a number of case studies, we find *AgileCtrl* can withstand user errors of up to $10^6\times$ while achieving similar performance.

CHAPTER 1

INTRODUCTION

"all constants should be configurable, even if we can't see any reason to configure them." — HD4304

For modern software, the number of software configurations increases dramatically over the years [88]. The Explosion of configuration naturally brings numerous misconfiguration problems. Although great efforts have been made in automated software configuration management(including both fixing and tuning) recently, the role of configuration in software hasn't been exclusively explored. Specifically, lack of understanding configuration could cause cascading effects in cloud computing era. The underline requirements including performance trade-off and hard constraints pose several new challenges for automatic configuration tuning.

Given the challenge of setting proper configurations, this dissertation conducts an empirical study of performance-related configurations, proposes a novel control-theoretic automatic configuration framework, and further enhances the framework from both robustness and usability.

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 burden 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 [25, 27, 90]. Misconfiguration leads to both incorrect functionality (e.g., wrong outputs, crashes) and poor performance. Although

recent research has tackled functionality issues arising from misconfiguration [90, 89], 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. Previous studies find that 20% of user-reported misconfiguration problems result in severe performance degradation, and yet, performance-related misconfigurations are under-reported [93]. Additional surveys show about a third of Hadoop's misconfiguration problems result in `OutOfMemoryError`s [67].

Setting performance-related 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 [28]. Even with clear documentation, the workload and system interaction are often too *complicated* or *change* too quickly for users to maintain a proper setting [67]. In many cases, there is simply no satisfactory static setting [19].

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 3.

Control theoretic frameworks handle constrained optimization for non-functional software

properties [23]. Typical control solutions, however, require deep understanding of a specific application or system (e.g., [33, 52, 77, 49, 97]), and hence, are impractical for real-world developers to adopt. Even general control synthesis techniques (e.g., [20]) 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 [81, 92, 12, 101]. Some approaches employ ML to meet resource constraints in dynamic environments [18]. 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 [79]. In contrast, control theory specifically addresses formal analysis of system dynamics [30]. Empirical studies of computer resource management confirm that control theoretic solutions do a better job of meeting constraints in practice [53].

Recently, the self-adaptive approach, including both Control theoretic or machine learning based solutions has also been applied to software configuration tuning [85, 37, 36, 20]. Through offline profiling, these techniques approximate the performance–configuration model of the target software. At run time, they observe performance metrics and automatically adjust configurations based on the model so that the performance metrics satisfy operating constraints and stick close to optimization goals. It has been demonstrated that these self-adaptive techniques have the advantage in combating dynamics (e.g., unpredictable environmental disturbances or workload fluctuations) while achieving optimal performance under constraints (latency, memory usage, power budget, etc.)[85, 21, 54, 37, 36, 8].

Although effective, the above self-adaptive frameworks all face the challenges of how to adjust their own models to accommodate different applications and dynamic environments.

Some frameworks decide their adaptation logic purely through **offline profiling**. For

example, the software adaptation conducted by control-based techniques often relies on a simple regression model obtained through offline profiling that quantifies the relationship between the software configuration settings and the software performance [85, 21, 54, 37, 36]. Similarly, the software adaptation conducted by learning-based approaches relies on complicated machine learning models (e.g., artificial neural network, random forest, and so on) obtained through offline training [95, 13, 69, 82]. Unfortunately, those offline profilings may become invalid when workload or environment changes dramatically [95, 13, 69, 82].

Some frameworks use **online learning** and reinforcement learning to accommodate for unexpected online environments like workload changes or resource contentions [32, 8, 51]. Although requiring no offline profiling, these techniques still require pre-configured attributes in their adaptation logic. For example, reinforcement-learning based techniques rely on users to statically configure attributes [32, 8] to balance the trade-off between exploring new configurations and exploiting existing configurations, which again may not work for all applications and all dynamic environments.

1.2 Contributions

In this dissertation, we first conduct an empirical study to understand real-world performance-related configuration problems. The study's results motivate us to construct a general framework, *SmartConf*. Unlike traditional configuration frameworks—where users set PerfConfs at system launch—*SmartConf* automatically sets and dynamic 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 1.1. We further identify that the existing control framework for software system could still suffer from insufficient profiling, workload/resource changes, and even human errors, and demonstrate its potential consequences of system instability or crashes. Guided by those findings, we propose *AgileCtrl* to further boost the

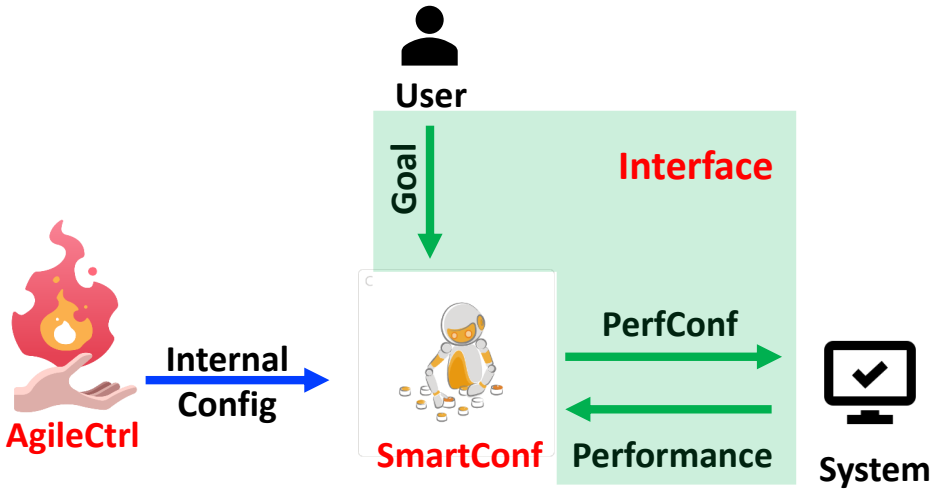

Figure 1.1: An overview of *SmartConf* and *AgileCtrl* presented in this dissertation.

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*

Table 1.1: Traditional configuration vs *SmartConf* configuration

existing software control framework from the robustness and usability aspects.

Empirical study We look at 80 developer-patches and 54 user-posts concerning PerfConfs in 4 widely used large-scale systems. We find that (1) PerfConfs are common among configuration-related patches (>50%) and forum questions (∼30%); (2) almost half of PerfConf patches fix performance problems caused by improper default settings; (3) properly setting most 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. 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 that, although not a panacea, *SmartConf* framework solves many PerfConf problems in real-world server applications.

Dynamic factors exploration We identify prior configuration tuning frameworks that rely on either offline profiling or the manual effort from domain experts to decide key internal model attributes called AdapConfs. However, AdapConfs could become new error-prone

components in the software system. We demonstrate that several dynamic factors — including insufficient profiling, workload/resource changes, and erroneous human interventions — will cause system performance degradation, system instability, or even crash.

***AgileCtrl* framework** To combat against dynamic factors, we propose *AgileCtrl* to **completely** eliminate any PerfConfs or AdapConfs requiring the human to set, while system performance and stability are ensured. *AgileCtrl* monitors the quality of its adaptations and reconfigures its own internals to provide even greater robustness in the face of user error or unexpectedly volatile environments. We evaluate *AgileCtrl* by comparing against recent control-based approaches to self-adaptation that require some user configuration. Across a number of case studies, we find *AgileCtrl* can withstand user errors or changes in workload of up to $10^6\times$ in most cases.

Besides the *SmartConf* from this dissertation was published in **ASPLOS'18** [85], I also participated in the following projects that are not included in this dissertation. (1) **LearnConf** [50] presents a taxonomy on how configuration affects the performance, and we develop a tool to infer the configuration properties statically. (2) **BugTM** [14, 15] is concurrency-bug recovery tool that leverages Hardware Transactional Memory. (3) **ReGen** [86] is a prototype tool for reproducible experiment used in Chameleon Cloud[44].

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 discuss the control background and related works for automatic configuration tuning. Chapter 3 presents our empirical study on performance-related configurations. Chapter 4 introduces our control-theoretic framework *SmartConf* for automatic configuration tuning. Chapter 5 presents our *AgileCtrl* for system robustness and usability enhancement. Chapter 6 concludes this dissertation and discusses future research work.

CHAPTER 2

BACKGROUND AND RELATED WORK

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

2.1 Control theory

Control theory has been developed since 19-th century, and has been successfully applied in many areas, from car's cruise control to accurate spacecraft control. The control systems are supported by rigors mathematical analysis of the stability, convergence, robustness and other properties. We will first introduce the traditional Controller, then extend it to the generalized controller for **software**.

2.1.1 Traditional Controller

The traditionally controller is widely used such as cruise control, anti-skid system, climate control, and *et al.*. To apply control theory in reality, the prerequisite is the model synthesis stage. Such a model could come from either physics laws(*e.g.* Newton's laws of motion or Kirchhoff's current law, and etc.) or experimental data. Usually, physics-based modeling is more accurate, while data-based modeling suffers from noise.

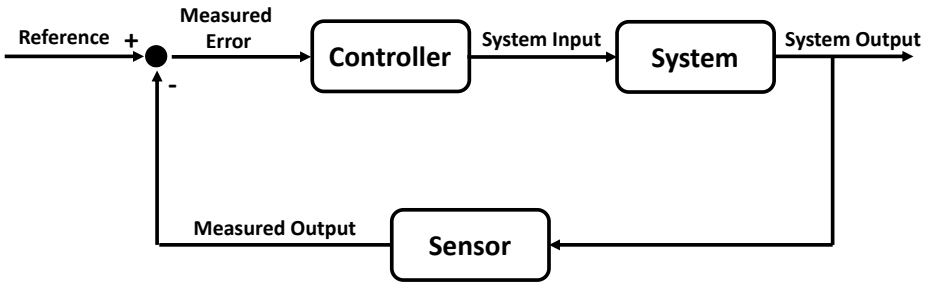

Figure 2.1: Traditional Controller

Based on the model as well as control specification(overshoot, settling time, etc), the control expert will manually synthesize a controller accordingly. During the runtime, the controller, will take measured performance as feedback, track the error against performance goal, and adjust the configuration accordingly. Figure. 2.1 shows how a traditional controller works during the runtime.

The traditional control system is expected to maintain its properties(e.g. stability, convergence, and robustness) if applied the system is slightly different from its model synthesis or environment changes. They are designed to meet the goal under the dynamic disturbances.

2.1.2 Generalized Controller for Software

Recently, a generalized control approach has been proposed for the self-adaptive software [20]. Such approach first *approximates* how the *current performance* (latency, energy consumption, etc) reacts to the *configuration* (queue size, cpu frequency, etc) through offline profiling. Specifically, a regression model is built to quantify the effects:

$$s_k = \alpha \cdot c_{k-1} \tag{2.1}$$

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

$$c_{k+1} = c_k + \frac{1-p}{\alpha} e_k. \tag{2.2}$$

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

SmartConf, a state-of-art adaptive control framework, further advanced the above controller from the following aspects to deal with the unique challenges for software control:

- **Virtual Goal**: Around half of performance related configuration threaten *hard* performance goal, *e.g.* , memory or disk[85]. Therefore, *SmartConf* introduced a the *virtual goal* $\hat{s}$ to provide some buffer space between *virtual goal* and *actual goal*. This helps avoiding overshooting the hard goal.

- **Indirect Control**: About half of those configuration affects performance indirectly through thresholds s for other system variables[85]. Therefore, it is important to correctly identify the actual configuration we should measure and control.

- **Parameter Free**: *SmartConf* is designed for introducing **no** additional parameters to the developer or user. Therefore, during the offline profiling, parameters (pole p and virtual goal $\hat{s}$) are *statically* set according to system instability. Pole p is statically determined by $p = 1 - 2/\Delta$, if $\Delta > 2$ and $p = 0$ otherwise. Specifically, Δ is determined by system's (in)stability based on offline 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. The virtual goal $\hat{s}$ can calculated by $(1-\lambda) * s$. $\lambda := \frac{1}{N}\sum_1^N \frac{\sigma_i}{m_i}$, where σ_i and m_i are the standard deviation and mean of the performance measured under the i-th sampled configuration value based on offline profiling. All in all, both pole p and virtual goal $\hat{s}$ are statically determined by *static* offline profiling and never changed during the runtime.

This generalized control model has been successfully applied to many different domains. For example, *CoPPer* aims at the meeting software latency goal by automatically tuning the power cap [35]. *SmartConf* leverage such control system to meet different performance metrics(including disk/memory usage, latency requirement) by tuning software level configurations [85]. *POET* meets time constraint by automatically tuning the hardware level configurations, e.g. core speeds and hyper-threads [37].

2.2 Misconfiguration

Many empirical studies have looked at misconfiguration [25, 60, 88, 94], but did not focus on PerfConfs. Much previous work has proposed using static program analysis [66, 89] or statistical analysis [87, 84, 96, 98] 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, 83] and misconfiguration-related performance problems [1]. They are complementary to *SmartConf* that helps avoid misconfiguration performance problems.

2.3 Automatic Configuration Tuning

Large modern softwares contain hundreds to thousands configurations and those configurations are usually badly documented and are hard for both developer and user to set[85]. Great efforts have been made towards automatic configuration tuning in recent year. Specifically, existing approaches can be classified as, model-based tuning, search-based tuning, and learning-based tuning. Model-based tuning [31, 3] relies on the accurate performance model of the system. Such model synthesis usually requires domain specific knowledge to abstract software with mathematical model. The model is highly abstracted and very specific to the analyzed software. Search-based tuning[101, 61] treats the software as black-box and use searching algorithm to find the optimal settings. However, those approaches suffers from exploration and exploitation problem, and not suitable for dynamic adjustment during the runtime. Learning-based tuning[95, 13, 69, 85] usually builds a performance model based on the profiling, and find the best configuration during the runtime. The performance models could be regression model[69, 85] or machine learning model[95, 13].

However, all those works mainly focus on improving the system performance for the **similar** workload or environment. In fact, both workload and environment have important impacts on software performance, and they are unusually hard to model and learn. *SmartConf* [85], a control theory based solution, provides a formal guarantee that systems can achieve desired performance when the environments changes are within a small and pre-defined boundary.

AgileCtrl is designed specific for extending the system **robustness** without sacrificing the performance gain. For previous works, the parameters are **statically** determined by the offline profiling workload and environment. However, *AgileCtrl* automatically adjusts those parameters during the runtime to accommodate the workload and environment changes. Consequently, the system error tolerance is greatly extended and system performance is improved. In fact, *AgileCtrl* is designed to eliminate the offline process; every parameter obtained from offline profiling should be adjusted as well. Though *AgileCtrl* has only been applied to *SmartConf* in this paper, the idea of *AgileCtrl* should be applied to any automatic configuration tuning framework based on either static performance model, static searching algorithm or offline profiling.

2.3.1 Control theoretic frameworks

Control theory provides a general set of mechanisms and formalisms for ensuring that systems achieve desired behavior in dynamic environments [48]. 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 [43, 29].

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

[52, 77] or mobile system [49]) 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 [23]. Towards this end, recent research synthesizes controllers for software systems [20, 22, 71]. Other approaches package control systems as libraries that can be called from existing software [99, 68, 38]. 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.

Many learning approaches have been proposed for predicting an optimal configuration within a complicated configuration space [31, 41, 81]. Perhaps the most closely related learning works are those based on reinforcement learning (RL) [79]. Like control systems RL takes online feedback. Several RL methods exist for optimizing system resource usage [24, 40, 39, 6, 58]. RL techniques, however, are not suited to meeting constraints in dynamic environments [78]. 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 [53].

2.3.2 Machine Learning

Machine learning has been widely used to learn the system performance model as the foundation for searching the optimal performance[95, 13, 69, 82]. In general, those approaches require huge amount of efforts on data collection, *e.g.* tens of hours for one workload [95], let alone infinitely many disturbances, workloads, and environment. A limited amount of training data is not enough for machine learning technique works in dynamics. The ability to adjusting the machine model itself based on dynamics during the runtime is needed. In contrast, control theory is designed for system dynamics with formal guarantee[30]. Moreover, the machine learning model, such as neural networks and deep learning, usually are hard to interpret as the target system is treated as a black-box[47]. It becomes much harder to adjust machine learning model online without understanding the model. Therefore, machine learning is not suitable for dealing with the dynamic.

2.4 Adaptive Control

Traditional control framework can still maintain its properties if the applied system is slightly different from its model synthesis or suffers from environment changes. To further advance the controller for unexpected dynamic disturbances, the adaptive control aims at adapting its underlying model during the runtime to compensate the environment or workload changes.

Though varies of adaptive control techniques have been proposed, they trend to be designed for specific systems. For example, it has been successfully applied to Aerial Vehicles[65], Engine Control [11], Distillation Column[46] and so on. However, the adaptive control is hard to be generalized to different applications because of different underlying system models. *AgileCtrl* is designed specific to enhance a general control frameworks[20], which approximates the system model as a linear model without taking the high-order correlations into consideration. *AgileCtrl* does not require additional assumption other than the control framework, so *AgileCtrl* itself is also general to different applications as well as

different types of disturbance to the system.

Most importantly, adaptive controllers are inherently nonlinear and complex[10] and prior researches focus on establishing stability analysis of the adaptive control. However, most adaptive control introduces additional parameters, which need the control expert to set. For non-expert, those parameters are hard to set and error-prone as well. Previous work, *CoPPer* and *POET* [35, 37], took the first step for adjusting controller key parameters using Kalman filter, however, it requires to set two additional parameters, namely,process and observation noise with assumption of Gaussian distribution. *AgileCtrl* aims at allowing non-expert to use without setting additional parameters.

CHAPTER 3

UNDERSTANDING PERF-SENSITIVE CONFIGURATIONS

"This is hard to configure, hard to understand, and badly documented." — HB13919

3.1 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. They 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 key word "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 3.1.

We also search StackOverflow [76] with key words like "config" to randomly sample 200–300 posts for each system. We then manually read through 1000 total posts to identify

Table 3.1: 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

configuration and PerfConf posts shown in Table 3.1. We find the StackOverflow information less accurate than the issue-trackers, and hence only discuss user complaints in Section 3.2.1, 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.

3.2 Findings

3.2.1 How Common are PerfConf Problems?

As shown in Table 3.1, 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 problems. Thus, the patch either changed a default setting or made a hard-coded parameter configurable. The other half simply added PerfConfs to support new features, as shown in Table 3.2.

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 [67], we found many posts related to OOM ($\sim$30%).

Table 3.2: 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

3.2.2 What are PerfConfs' Impact?

What Type of Performance do They Affect? As shown in Table 3.3, most PerfConfs affect user request latency. They also commonly affect memory or disk usage, threatening server availability through OOM/OOD failures (half the cases). Naturally, one metric could be affected by multiple PerfConfs simultaneously, with several coordination issues [55, 9].

As Table 3.3 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 5.5), 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 all the time, 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

Table 3.3: 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

Table 3.4: 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

thresholds on some system variables—e.g., queue size `ipc.server.max.queue.size`, number of operations per log file `dfs.namenode.max.op.size`, and number of outstanding packets `dfs.max.packets`—which, in turn, affect performance.

3.2.3 How to Set PerfConfs?

Format of PerfConfs A prior study shows configurations have many types [88]. PerfConfs, however, are dominated by numerical types. As shown in Table 3.4, 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 [88]. Thus, due to their numeric types, PerfConfs are naturally difficult for users to set.

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 3.4). 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 ($\sim$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, 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 at run time. With no support for dynamic adjustment, Cassandra developers chose a conservative setting that lowers the possibility of OOM by sacrificing write performance for many workloads.

3.3 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.

CHAPTER 4

SMARTCONF

4.1 *SmartConf* Overview

"I don't know what idiot set this [configuration] to that.. oh wait, it was me..." — HD4618

SmartConf is a control-theoretic configuration framework for PerfConfs. As shown in Figure 4.1, under *SmartConf*, users only need to specify performance goals, instead of the exact configuration settings. With 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.

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 [30], such as environment changes and workload fluctuation, which is crucial in setting many PerfConfs as discussed in Section 3.2.3. A controller dynamically *adjusts* a configuration based on the *difference* between the current performance and the goal, as illustrated in Figure 4.1. 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 3.3, many configurations affect memory and disk consumptions, where the main concern is not exceeding limits instead of achieving a specific optimal value. Even for those PerfConfs that affect request latencies, the corresponding goals are usually maintaining service-level-agreements, instead of achieving

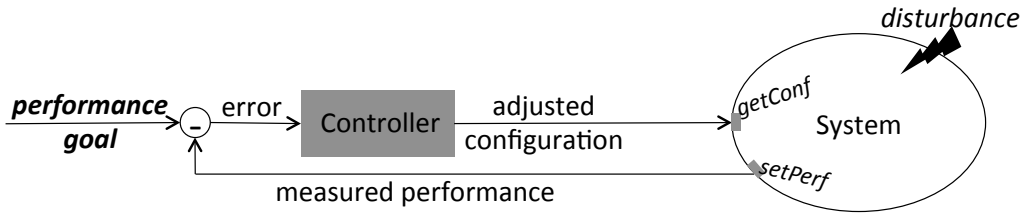

Figure 4.1: Using a controller to adjust PerfConf (gray parts are extra controller-related components in *SmartConf*.)

optimal latencies. Controllers are a good 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, and no guarantees were necessary.

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.

4.2 *SmartConf* Framework

"It will be even great if we can dynamically tune/choose a proper one." — HB7519

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 4.3 describes how *SmartConf* library is implemented with new control theoretic techniques.

```
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 4.2: *SmartConf* configurations

4.2.1 Developers' effort

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 4.2. 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 our evaluation, 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 4.2. For example, a goal about "memory consumption should be smaller than the JVM heap size" is a hard constraint.

Calling *SmartConf* APIs

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

```
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 updateGoal (double goal);
7  }
```

Figure 4.3: *SmartConf* class

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 4.3, 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 *SmartConf* 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 Every time 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.

4.2.2 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

```
1  /* For indirect configurations */
2  public class SmartConf_I extend SmartConf {
3    SmartConf_I (string ConfName, Transducer t);
4    void updatePerf (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 4.4: *SmartConf* sub-class

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 4.4. 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 4.4.

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 3, some configurations affect performance metrics conditionally. Consequently, the corresponding controller should only be invoked when the configuration takes 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.

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.

4.2.3 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 `updateGoal` API in Figure 4.3.

4.3 *SmartConf* Controller Design

"everything always has a tradeoff." — CA13304

Baseline controller We choose a recently proposed controller-synthesis methodology [20] as the foundation for the *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:

$$s_k = \alpha \cdot c_{k-1} + b \tag{4.1}$$

where s_k is the system performance measured at time k, c_{k-1} is the configuration value at time $k-1$ and b is the intercept. 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}:

$$c_{k+1} = c_k + \frac{1-p}{\alpha} e_{k+1}. \tag{4.2}$$

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 4.3.6.

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

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.

4.3.1 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 [30].

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).

4.3.2 Handle Hard Goals

Many PerfConfs are associated with a `hard==1` constraint, meaning that the goals like no OOM cannot be violated (Table 3.3). 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 overshoot-

ing unlikely. Unfortunately, this strategy does not work, as will be shown in experiments (Section 5.5). 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 [75] 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, 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 set the virtual goal $\tilde{s}^v$, without providing 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 from $\tilde{s}$ to avoid accidental over-shooting. Otherwise, $\tilde{s}^v$ can be set to be close to $\tilde{s}$ to allow better resource utility.

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. Consequently, 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 4.3.1. 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 in to 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 using the above algorithm (i.e., two poles and virtual goal) once the configuration file specifies a performance goal with the attribute `hard==1`. Experiments in Section 5.5 demonstrate the above two techniques are crucial to avoid over-shooting while maintaining high resource utility.

4.3.3 Handle Configurations with Indirect Impact

A PerfConf C may serve as a threshold for a deputy variable C' ($\sim$50% among PerfConfs in Table 3.3). 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 4.4). *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 4.4.

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 into the desired range.

4.3.4 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 3.3. 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 [22]. 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 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 4.3.1) and the virtual goal (Section 4.3.2). As we will discuss soon in Section 4.3.6, 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.

4.3.5 Other Implementation Details

Our *SmartConf* library is implemented in Java. The `SmartConf` classes shown in Figure 4.3 and Figure 4.4 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.

Table 4.1: Benchmark suite and runt-time setting for module evaluation. In issue description, the main constraint that users complain about is put earlier, the trade-off description is later, and finally the metrics of constraint performance (goal) and trade-off performance.

ID	Issue Description	Run-time Setting		
		α sign	α value	vg
HD4995	`content-summary.limit` limits #files traversed before `du` releases big lock. Too big, write blocked for long; Too small, `du` latency hurts. Write latency; `du` latency	Manually flip α sign	Limit CPU usage to 100%, 50%, 20%, 10%	Reduce vg to 20%, 40% 60%, 80%
HB2149	`global.memstore.lowerLimit` decides how much memstore data is flushed. Too big, write blocked for too long; Too small, write blocked too often. Tail write latency; The number of violated write latency			
HB3813	`ipc.server.max.queue.size` limits RPC-call queue size. Too big, OOM; Too small, read/write throughput hurts. Used memory; Write throughput		Increase request size to 1MB, 2MB, 5MB, 10MB	
HB6728	`ipc.server.response.queue.maxsize` limits RPC-response queue size. Too big, OOM; Too small, read/write throughput hurts. Used memory; Write throughput			
CA6059	`memtable_total_space_in_mb` limits the memtable size. Too big, OOM; Too small, write latency hurts. Used memory; Write latency			

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>`.

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 5.5, *SmartConf* produces effective and robust controllers without intensive profiling.

4.3.6 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 crashes. Based on analysis in previous work the controller in equation 5.2 is stable as long as $0 \leq p < 1$ and $p = 1 - 2/\Delta$ for $\Delta > 2$ [20]. 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 $0 \leq p < 1$ [30]. Such analysis, however, assumes no disturbances, but we know we

are deploying *SmartConf* into unpredictable environments.

With two enhancements discussed in Section 4.3.2, 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.

4.4 Evaluation

"I think going to 1G [default] works, ... let's do some testing before submitting a patch" – HB4374

4.4.1 Evaluation methodology

Benchmarks We apply *SmartConf* to 6 PerfConf issues in Cassandra, HBase, HDFS, and MapReduce, as shown in Table 5.2. 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 5.2. 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 5.2. 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 that we can evaluate how sensitive *SmartConf* is towards profiling; (2) the evaluation workload contains two phases

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

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 [17] benchmark workload-A, which has a 50-50 read-write ratio; for HDFS, we use a common distributed file system benchmark TestDFSIO [34]; and we use WordCount for MapReduce, as shown in Table 5.2.

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.

4.4.2 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 4.5, 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 4.6b 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 4.6c—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 4.6c—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 4.6c. Overall, the system never has OOM errors with *SmartConf*.

In comparison, the old default `max.queue.size` 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 4.6ab. However, there is no way for users or developers to predict what configuration will be conservative enough for future workload.

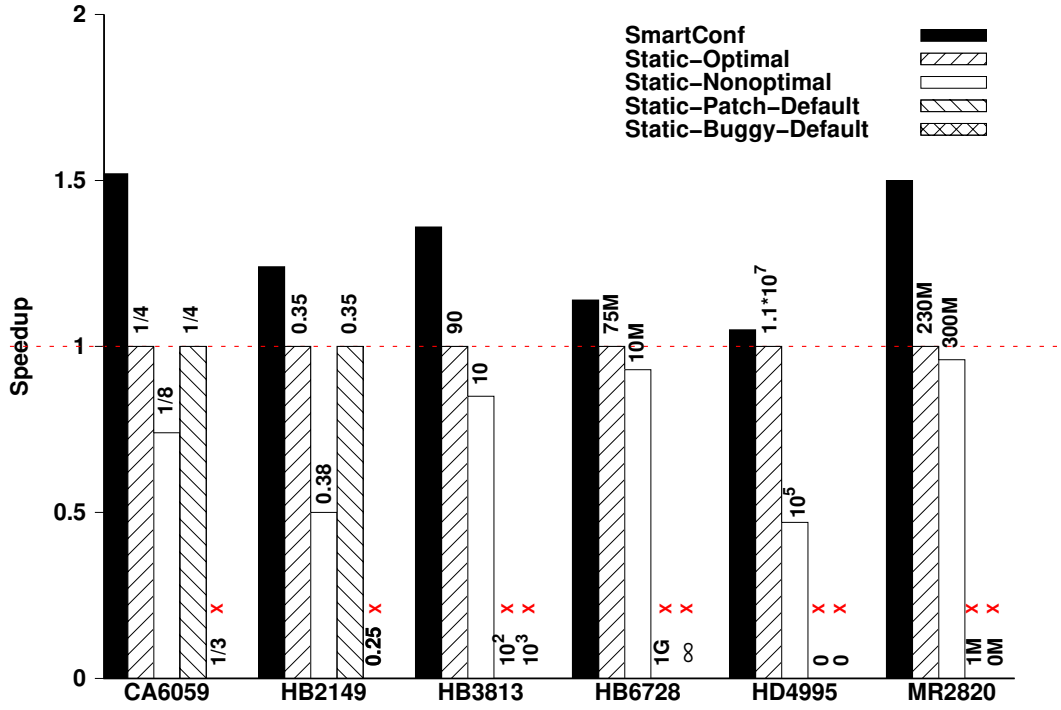

Figure 4.5: Trade-off performance comparison. Normalized upon the best-performing static configuration; x: fail the perf. constraint. The numerical PerfConf settings are above each bar.

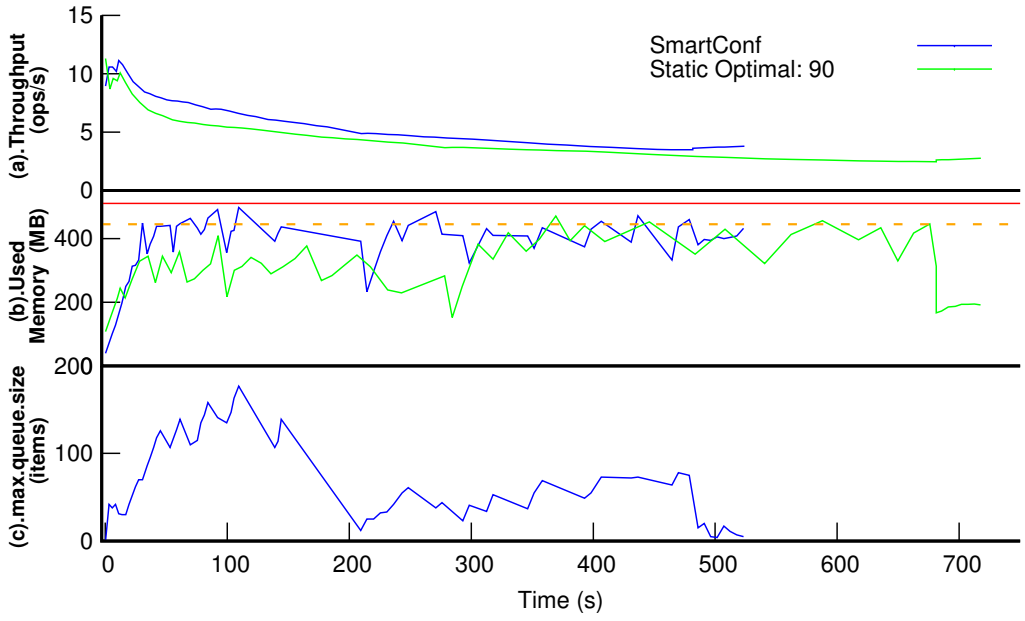

Figure 4.6: *SmartConf* vs. static optimal on HB3813. workload changes at ∼200s. Throughput is accumulative.

4.4.3 Does SmartConf Provide Good Tradeoffs?

Figure 4.5 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 the performance constraints once workloads change. Figure 4.5 also shows the performance under non-optimal static settings that we randomly choose.

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 4.6c), 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 4.6ab, 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. Consequently, *SmartConf* runs WordCount much faster in phase-1, and achieves 1.50× total speedup.

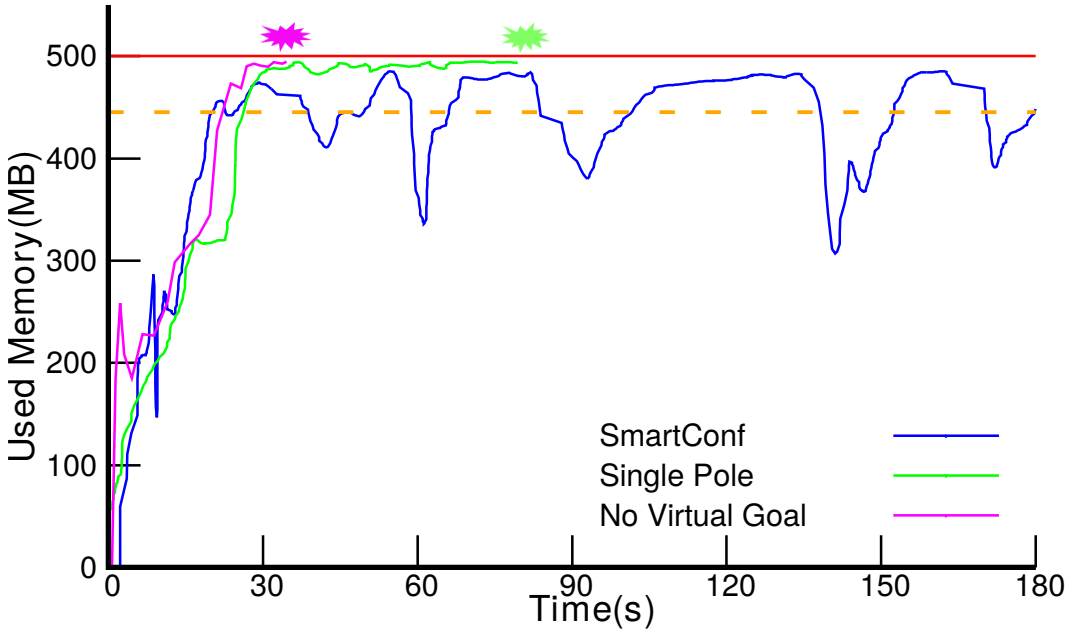

Figure 4.7: *SmartConf* vs. alternative controllers.

4.4.4 Alternative Design Choices

SmartConf's controller handles hard constraints differently from traditional control design in two ways (Section 4.3.2). We experimentally compare 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 [75]—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 4.7, *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

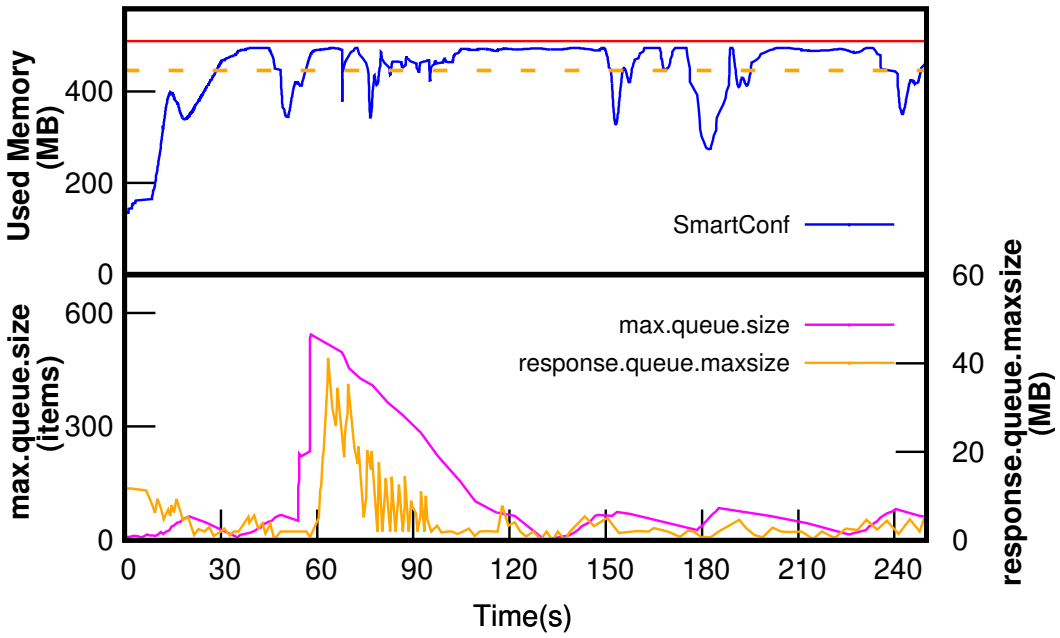

Figure 4.8: *SmartConf* adjusts two related PerfConfs.

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

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

Table 4.2: Lines of code changes for using *SmartConf*

4.4.5 Other results

Is *SmartConf* easy to use? As shown in Table 4.2, it takes little effort for developers, as few as 8 lines of code changes. The changes are dominated by implementing performance sensing. For HB-3813, extra changes are needed to tolerate the temporary inconsistency between a deputy variable (`queue.size`) and the indirect configuration (`max.queue.size`). For MR-2820, extra changes are needed to deliver the configuration computed by one node (Master) to another (Slave).

Interacting controllers To evaluate whether *SmartConf* can handle multiple interacting PerfConfs, as mentioned in Section 4.3.4, 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 4.8 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 request, 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 still guarantee the hard constraint. It also further motivates dynamically adjusting configurations: otherwise, we would have to pick very small sizes for both queues.

4.4.6 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, MR-5420 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 MR-5420 — 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. Third, some configurations might be inherently difficult to adjust dynamically, as the adjustment may be expensive. For example, changing `max_chunks_tolerable` dynamically may require copying files around. Finally, *SmartConf* provides statistical guarantees as discussed in Section 4.3.6, but cannot guarantee all constraints to be always satisfied.

4.5 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 often out-perform static optimal configuration setting, while satisfying performance constraints.

CHAPTER 5

AGILECTRL

5.1 Introduction

Modern software systems provide great flexibility to the user by allowing them to customize (or tune) the software's configuration parameters. These configuration parameters determine the size of critical data structures, the thresholds to trigger time-consuming operations, the parallelism of the process, the network addresses of primary and replicas nodes, and many other aspects of system operation[91, 100, 50, 85]. A recent study found that 65% of configuration-related issues in 4 major distributed systems involve performance concerns [85]. Those concerns include not only performance optimization like reducing user-request latency or improving internal job throughput but also requirements on performance like memory/disk consumption. Those performance-sensitive configurations, *PerfConfs* for short, are especially hard to set. In fact, 76% of studied PerfConfs from issue-tracking systems affect multiple performance metrics at the same time and PerfConfs represent tradeoffs; *e.g.* between memory usage and latency [85]. Moreover, the optimal PerfConf varies due to runtime dynamics (like workload changes and interference).

To ensure that software is configured optimally despite dynamic changes in operating environment, prior works have proposed self-adaptive frameworks like *SmartConf* [85], DAC [95], OtterTune [81], POET [37] , SimCA[74, 72], CAPES[51], AENEAS[8] and so on. Such frameworks monitor software performance and automatically adjust PerfConfs to ensure optimal operation despite unpredictable external changes. For example, AENEAS dynamically adjusts the GPS accuracy (`gpsPrio`) and update interval `gpsUpdate` in Android to meet performance requirements (*e.g.* 20% per hour battery drain rate). By dynamically configuring the PerfConfs, such self-adaptive frameworks make software substantially more robust and performant than approaches that must stick with a single PerfConf setting for their lifetime

[85, 21, 54, 36, 8, 51, 74, 72, 81, 95].

However, these approaches do not eliminate all the burden of managing PerfConfs, as self-adaptive frameworks themselves expose configuration parameters that need to be set by users. We call these **AdapConfs** to distinguish the self-adaptive framework's parameters from the PerfConfs it should control.

For example, to use an self-adaptive framework based on control theory, users must to set an **explicit AdapConf** (the *p*ole of the controller's characteristic equation) [21, 54, 37, 36], which determines the self-adaptive frameworks' tradeoff between reaction time and noise sensitivity. If the *pole* is too small, then the self-adaptive is more sensitive to disturbances and it may crash the software system (see Tab. 5.2 in Sec. 5.3); if the *pole* is set too large, then the self-adaptive is slow to change the PerfConf, which leads to sub-optimal performance and negates the benefits of using the self-adaptive framework in the first place.

In addition to these explicit AdapConfs, there are **implicit** AdapConfs which are not directly set by users but arise from the fact that the framework requires users to provide training (or profiling) data that is used to train a model stored within the framework itself. For example, in a machine-learning-based self-adaptive framework, representative inputs are used to determine implicit AdapConfs (*e.g. weights* used in artificial neural networks)[95, 8, 81]. In fact, if the training data sets do not represent the behavior seen during deployment, the self-adaptive frameworks will not react appropriately and can crash or fail to deliver the required performance [16]. For example, if the training inputs for managing a webserver consist only of small-size requests, the system could potentially crash when the actual workload consists of much larger requests[85].

In summary, self-adaptive frameworks have the potential to automatically configure software systems, but they do not completely solve the configuration management problem because they typically have their own explicit and implict configurations. In other words, prior works proposing self-adaptive frameworks replace PerfConfs with AdapConfs, which

must still be set by users. And, much like PerfConfs, the optimal settings for these AdapConfs depend on the target software, the hardware platform, and the run-time environment. Like PerfConfs, those AdapConfs become a new source of bugs, either from setting the explicit AdapConf incorrectly, or providing insufficient or unrepresentative training data for the implicit AdapConf. Unsuitable AdapConfs could lead to sub-optimal performance, system instability, or even a system crash. Thus, a configuration-free (both PerfConfs and AdapConf free) self-adaptive framework for general software systems is highly desired.

In this work, we aim at **completely** eliminating any configurations that require pre-configuration in a big family of self-adaptive frameworks that function through control-theoretic techniques [85, 37, 36, 20, 21, 30, 74, 72, 73, 62]. We propose a novel self-adaptive control framework *AgileCtrl*, which extends general control-based self-adaptive systems by automatically adjusting its internal AdapConfs so that neither pre-configured explicit or implicit AdapConfs are needed. The design of *AgileCtrl* leverages the following insights. **First**, different from traditional self-adaptive frameworks that only adjust the PerfConfs of the target software based on how well the target software is performing(*e.g.* , request latency or memory consumption), *AgileCtrl* also adjusts the AdapConfs of the self-adaptive framework based on how well the framework itself is adapting. Specifically, we propose that both control-based and learning-based self-adaptive frameworks can evaluate the qualtiy of their own adaptations by predicting the future performance of the underlying software system and then evaluating the accuracy of those predictions. In other words, traditional self-adaptive frameworks aim at optimizing software performance, M_{software}, while *AgileCtrl* evaluates itself based on the difference between its predicted $M^{\text{predict}}_{\text{software}}$ and the actual software performance measured at run time $M^{\text{observe}}_{\text{software}}$. **Second**, for control-based self-adaptive frameworks, *AgileCtrl* further leverages a simplified MIT rule[1] to estimate AdapConfs [63]. Specifically, the feedback about how well the adaptation framework is performing enables

1. MIT rule was developed at Instrumentation Laboratory (now Draper Laboratory) at Massachusetts Institute of Technology.

AgileCtrl to gradually adjust the framework's AdapConfs so that the predicted software performance stays close to the observed software performance, which in turn allows the adaptation framework to effectively adjust the target software's configurations. Putting these together, *AgileCtrl* allows a large family of self-adaptation frameworks to dynamically adapt their own, internal adaptation logic to accommodate for different run time dynamics and compensate for a wide range of profiling deficiencies without any extra configuration requirements.

To demonstrate the problems with current state-of-the-art self-adaptive frameworks we first show how prior work (*SmartConf* [85]) can handle some environmental fluctuations, but suffers from performance degradation, oscillation, or crashes when those fluctuations become large enough (Section 5.3). We then compare *AgileCtrl*'s key ideas to *SmartConf* and other recent control-based self-adaptive frameworks. Across a number of case studies, we find that *AgileCtrl* not only avoids performance oscillation and system crashes but also improve the performance. Furthermore, *AgileCtrl* withstands errors setting AdapConfs by $10^6\times$ most of the time (Section 5.5), which greatly extend the system robustness.

To summarize, *AgileCtrl* makes the following contributions:

- Expose AdapConfs used in self-adaptive frameworks as a potential source of bugs and demonstrate the importance of properly setting AdapConfs.
- Propose *AgileCtrl* to eliminate both AdapConfs and PerfConfs from the software system by leveraging Model Reference Adaptive Controller used in control theory.
- Evaluate *AgileCtrl* against multiple state-of-the-art self-adaptive frameworks and demonstrate that *AgileCtrl* greatly enhance the system robustness with performance improvement.

5.2 Background

5.2.1 Self-Adaptive Framework

Modern software often faces the challenge of meeting non-functional requirements such as performance, energy consumption, and others [7], while facing unexpected changes at run time, such as resource contention and workload changes. Self-adaptive frameworks help tackle these challenges by automatically adjusting the software, like tuning the PerfConfs, based on the run time software status.

Self-adaptive frameworks are generally classified as control theory-based approaches [63, 85, 20, 37] and machine learning-based approaches [26, 8, 81, 95], with some representative ones listed in Table 5.1.

As shown in Table 5.1, all these systems, whether based on control theory or machine learning, contain two or more AdapConfs to help adjust the PerfConfs of the software under control. For *SmartConf*, *ADSS*, and *POET*, they use multiple AdapConfs to automate a single PerfConf, thus the total number of AdapConfs is actually larger than that of PerfConfs [85, 37, 20].

As indicated by the last column of Table 5.1, some AdapConfs are used to characterize the software model. For example, the α used in *SmartConf* and neural network weights W used in *OtterTune* reflect how PerfConfs affect the performance. Other AdapConfs improve the system's robustness to tolerate unseen workload, environment, or strict constraints. For instance, a proper pole p can avoid aggressive controller adjustment and withhold small disturbances; a suitable `learning rate (lr)` used in machine-learning avoids the model overfitting so that its predictions better generalize to unseen data. From the table, those AdapConfs are set either explicitly by an expert, which requires much domain knowledge, or indirectly through profiling or training.

Overall, those AdapConfs are common, important, and complicated. The importance of

setting suitable AdapConfs is underestimated, and the consequence of improper AdapConfs have not been thoroughly examined, which motivates our *AgileCtrl*.

5.2.2 Control-based Self-adaptive Frameworks

In this work, we focus on understanding and eliminating AdapConfs from a state-of-the-art control-based self-adaptive framework *SmartConf* [85], which represents a large family of control-based self-adaptive frameworks that are based on the Linear model Learned at Run time (LLR) combined with traditional Proportional-Integral-Derivative (PID) controller (as described in a recent survey paper on the use of control for self-adaptive software [70]). *SmartConf* applies PID control techniques to automatically adjust PerfConfs in distributed systems, while addressing several challenges unique and crucial to PerfConfs. *SmartConf* uses 3 AdapConfs to automate the setting of each single PerfConf c:

Coefficient (α) *approximates* how the *current performance* s_k (latency, energy consumption, etc) at time k reacts to the *PerfConf* value c_{k-1} (e.g., queue size, cpu frequency, etc) at time $k-1$ and b is the intercept. This value is typically set through offline profiling, where a linear regression model is built to quantify the effects as following:

$$s_k = \alpha \cdot c_{k-1} + b. \tag{5.1}$$

The α can be any real value. A positive α means increasing configuration improves the performance; a negative α means the opposite. α's value determines how aggressive the controller should react to the performance gap. The larger α's absolute value is, the more sensitive the system performance would react to any configuration changes, in which case the controller should be gradually tune the configuration otherwise performance will vary aggressively.

Pole (p) is a key parameter for PID controller which determines how aggressively the following controller reacts to the current performance error e_k, where c is a PerfConf and c_k

Table 5.1: Partial AdapConfs used in self-adaptive frameworks (**E**: explicit AdapConf and **I**: implicit AdapConf)

System	Category	AdapConf	Type	How to set ?	Role
SmartConf [85]	Control	$pole$	**I**	Profiling	Robustness
		α	**I**	Profiling	Model
		vg	**I**	Profiling	Robustness
ADSS [20]		$pole$	**E**	Expert	Robustness
		α	**I**	Profiling	Model
POET [37]		$pole$	**E**	Expert	Robustness
		q_b	**E**	Expert	Model
		m_v	**E**	Expert	Model
SimCA [74, 72]		$pole$	**E**	Expert	Robustness
		α	**I**	Profiling	Model
Brownout [45]		$pole$	**I**	Profiling	Robustness
		α	**I**	Profiling	Model
AENEAS [8]	Machine Learning	δ	**E**	Expert	Model
		σ	**E**	Expert	Model
		ϵ	**E**	Expert	Model
OtterTune [81]		W	**I**	Profiling	Model
		ds	**E**	Expert	Model
		lr	**E**	Expert	Robustness
		do	**E**	Expert	Robustness
DAC [95]		nt	**I**	Expert	Model
		tc	**I**	Expert	Model
		lr	**I**	Expert	Robustness
ACTGAN [4]		W	**I**	Profiling	Model
		k_d^h	**E**	Expert	Model
		lr	**E**	Expert	Robustness
RFHOC [5]		ps	**E**	Expert	Model
		mp	**E**	Expert	Robustness
		cp	**E**	Expert	Robustness

is the value of that PerfConf at time k:

$$c_{k+1} = c_k + \frac{1-p}{\alpha} e_k. \tag{5.2}$$

In *SmartConf*, p is set based on a profiling measurement of how (un)stable the software under control is—the more stable the software is, the smaller p is and hence the controller would react more aggressively. Specifically, *SmartConf* computes an (un)statbility metric Δ as $\Delta = 1 + \frac{1}{N}\sum_1^N \frac{3\sigma_i}{m_i'}$, 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. Then, *SmartConf* sets p to be $max(0, 1 - 2/\Delta)$.

Virtual Goal Ratio (vg) is brought in when there is a hard constraint of performance (like never overshooting memory limits). The vg is a real number between 0 to 1. It reserves some potential performance gain as cushion space to trade for robustness to system instability (like environment or workload changes). The more unstable the system is, the larger the cushion space needs to be reserved. Specifically, the virtual goal ratio $vg := 1 - \frac{1}{N}\sum_1^N \frac{\sigma_i}{m_i}$, where σ_i and m_i are the standard deviation and mean of the performance measured under the i-th sampled configuration value based on offline profiling. Then, Virtual Goal can be calculated by $vg * \tilde{s}$, where $\tilde{s}$ is the desired performance goal.

Note that, we will evaluate our tool on *SmartConf*, but the idea applies to other self-adaptive frameworks in this PID-controller based big family, which all contain AdapConfs similar to α, p and vg (as shown in the top half of Table 5.1).

5.3 Motivating Example

To motivate our tool, we investigate how *SmartConf* responds to the environment discrepancy between the offline-profiling stage and online-tuning stage and understand why tuning AdapConfs is important for system performance and stability. We take two benchmarks

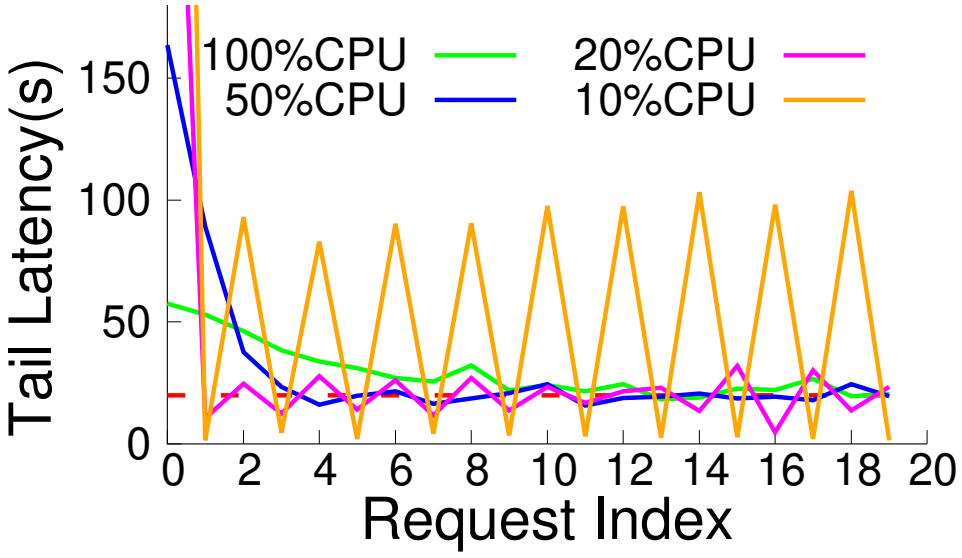

Figure 5.1: HD4995 under different CPU resources

from *SmartConf* [85] (HD4995 and HB3813) as examples, and the corresponding experiment settings and findings are as follows.

5.3.1 Different Run-time Resources

Our first example HD4995 reveals, through *SmartConf* tolerate some run-time resources changes, *SmartConf* can malfunction under dramatic changes.

Here, the PerfConf `content-summary.limit` limits the number of files that are traversed before `du`, a command for estimating file space usage, has to release a highly contested lock. If this value is too large, write blocked for long; Too small, `du` latency hurts.

We use the same profiling workload used in *SmartConf*. Specifically, we use the distributed file system benchmark TestDFSIO[34]. Based on the profiling workload, *SmartConf* indirectly learns AdapConfs ¡α, p, vg ¿ as ¡0.00005s, 0.53, 1.00¿, and α indicates average latency to traverse a single file or directory is around 0.00005s. At the runtime, we gradually decrease the CPU allocated to the program from 100% to 10%.

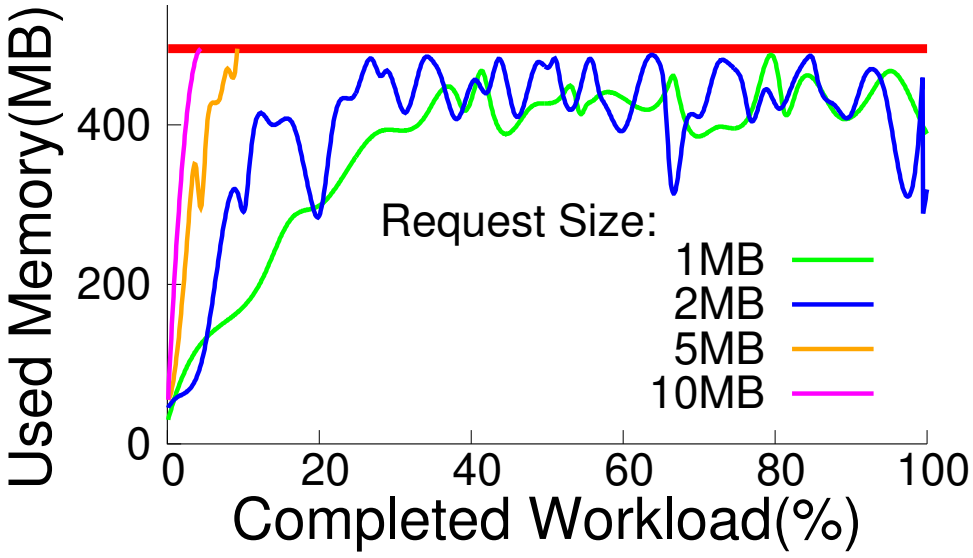

Figure 5.2: HB3813 under different workloads(1MB, 2MB, 5MB, and 10MB)

SmartConf provides some robustness when the online environment is a little bit different from offline profiling. As shown in Figure 5.1, under full CPU resources, *SmartConf* gradually reduces the tail latency to the requirement goal (20s). When HDFS allocated with 50% CPU resources, *SmartConf* can tolerate such environment changes and reduce the tail latency to the goal.

However, with 20% CPU resources, the *SmartConf* still acts aggressively, thus the tail latency oscillates a lot around the goal. If only 10% CPU resources are allocated to HDFS, then the system fails to converge and tail latency oscillates between 1s and 90s.

The reason behind this is our profiling stage utilizes the full CPU resource, so processing every file/directory (α) is fast. With lesser CPU resources, the processing becomes slower. Thus, system takes more time to process than *SmartConf* expected. As result, the tail latency oscillates though controlled by *SmartConf*.

5.3.2 Different Run-time Workloads

Our second example HB3813 shows, under different run-time workloads, the *SmartConf* can even overshot the hard constraint and result in crashes.

The PerfConf `max.queue.size` determines the largest size for an RPC queue used in Hbase. A large queue can lead to an out-of-memory (OOM) when under memory pressure, while a small queue reduces RPC throughput.

We use YCSB [17] workload-A with 1MB request size and 50-50 read-write ratio. Under this profiling workload, *SmartConf* indirectly determines three important AdapConfs ¡α, p, vg ¿ as ¡1.25MB, 0.45, 0.91¿. Specifically, α characterizes the average request size inside the queue. At the runtime, we gradually increase the workload from 1MB to 10MB.

As shown in Figure 5.2, for 1MB workload, *SmartConf* works as expected. For a slightly larger request size (2MB), *SmartConf* can still efficiently utilize the memory as *SmartConf* has some ability to accommodate different workload as well.

When request size becomes $5MB$, the system exceeds the memory limits after finishing around 10% of the total workload and then crashed. This is due to *SmartConf* still treat each request is the same as offline profiling. However, the run-time request size becomes larger than expected. Unsurprisingly, the Hbase also crashed with $10MB$ request size with only 5% workload finished.

In fact, the *SmartConf* synthesized with $1MB$ workload can still operate well under a different runtime workload as long as the request size approximates from $0.34MB$ to $4.54MB$ with 84% probability[85]. However, for $5MB$ and $10MB$ which are above this threshold, so *SmartConf* cannot save the system from crash.

Our two motivation experiments confirm that AdapConfs are important. In Sec. 5.5, we identify several representative scenarios that could result in similar consequences. When there is a large mismatch between the offline system used for controller synthesis and run-time system controlled by *SmartConf*, the system performances suffered became unstable, and

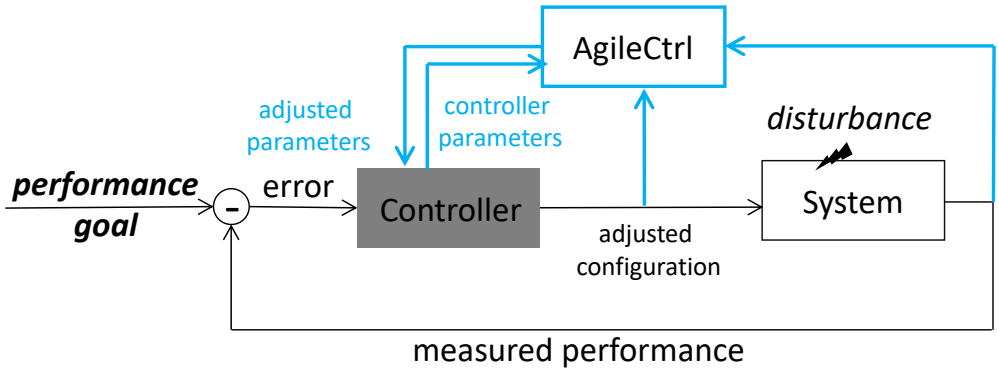

Figure 5.3: Agile Controller

even system crashed. Thus, an automatic adaption for all AdapConfs without introducing any new configuration is highly desirable. In other words, removing those explicit or implicit AdapConfs from self-adaptive framework is critical for a better self-adaptive system.

5.4 *AgileCtrl* Design

Motivated by the experiment in the previous section, we propose *AgileCtrl* to auto-adjust the AdapConfs in additional to existing *SmartConf* which automatically tunes PerfConfs.

To combat unexpected environment or workload changes, one could monitor them and adjust AdapConfs accordingly. Such a solution could be beneficial for revealing the root cause leading to system instability or crash. Yet, some of those, such as interference from different programs, are inevitable and cannot be ruled out, and identifying those the root cause itself cannot save the system. Moreover, the cost of monitoring **all** possible changes is usually inevitable and huge. Finally, how to adjust AdapConfs could be different for a different types of changes.

Theoretically, how well the system performance is converged towards the goal is guaranteed by *SmartConf*. Any performance deviation from performance expectation indicates the mismatch, thus AdapConfs are required to be adjusted. Therefore, *AgileCtrl* directly moni-

tors the quality of its adaptation regardless of the changes' presences or types. More specifically, shown in Figure 5.3, *AgileCtrl* extends traditional self-adaptive framework through (1) estimating the expected system performance based on current PerfConfs and AdapConfs and (2) automatically adjust AdapConfs as well as PerfConfs when there is a mismatch between expected and actual system performance. As result, *AgileCtrl* can successfully mitigate performance degradation or recover from possible system crashes.

Particularly, we aim at ensuring two representative AdapConfs (α and vg) mentioned in the Sec. 5.3 are correctly set. Pole p does not need to be considered separately in *AgileCtrl*, since (1) it was introduced to tolerate the inappropriate α and (2) α and p are correlated and together constitute a coefficient that determines how aggressively the controller reacts to the performance error. *AgileCtrl* is designed to fix the inappropriate α during the runtime, thus pole p is no longer needed (*e.g.* $p = 0$ is used in *AgileCtrl*).

Therefore, in the following subsections, we will discuss how to detect and fix (1) wrong α sign (2) wrong α value, and (3) wrong vg.

5.4.1 α Sign Module

The α sign determines the configuration tuning direction–increasing or decreasing the PerfConfs. Ideally, the expected performance should converge to the goal gradually. However, if the sign is wrong, then the performance will deviate away from the performance goal. Most importantly, the α sign is the **only** AdapConf that captures the trend – positive or negative correlation of performance-configuration and thus determines the tuning configuration bigger or smaller. Neither improper α nor vg results in wrong tuning direction. In other words, as long as the tuning direction is different from the expected trend, the α sign must be wrong. Thus, we can detect the wrong α sign based on tracking the trend.

Specifically, we can calculate the performance error e_k as the current performance $w.r.t$ the performance goal. Ideally, the traditional controller is asymptotically stable, which

means $|e_k|$ should decrease while gradually reduce to 0[64]. Therefore, we check the trend of e_k by comparing consecutive errors–whether the last i errors are decreasing or not. The noise could occasionally affect the temporary trend, but long-term trend remains the same. Specifically, we will use the following Algorithm 1 to detect sign changes.

Algorithm 1: Wrong α sign detection

Input : $\mathbb{G}$ – Performance Goal
C – Current Performance
i – Last i samples
Output: α – Updated alpha value

Calculate current error $e_k = |\mathbb{G} - C_k$—
if $e_k > e_{k-1} > \cdots > e_{k-i+1}$ **then**
    `/* incorrect` e_k `trend, sign flip is needed */`
    $\alpha_{k+1} = -\alpha_k$
end

5.4.2 α Value Module

Besides the α sign, its value determines how aggressive the controller reacts to the performance gap. Specifically, if the α used by the controller is too big than expected, then the controller might not be able to react to the system changes fast enough, resulting in performance degradation. On the other hand, if the controller's α is too small then the system becomes instability, causing performance oscillation or even crash.

In fact, the α is the key component of the system model that characterizes the relation between configuration and performance. Prior works mainly solve the α auto-adjusting through *Online Linear Regression* or *Kalman Filter*[38, 80, 20]. We analyze those two representative approaches, then propose our *AgileCtrl* support for α auto-adjusting.

Online Linear Regression (OLR) simply applies the same offline linear regression algorithm to the runtime [20]. However, some of PerfConfs change *continuously* (*e.g.* , configuration queue size can only increase or decrease by 1 each time.)[85]. The most recent PerfConfs are likely within a small range and cannot be representative. Statistically, for the

same data, a broader range of the independent variable usually results in a higher coefficient of determination ($\mathbb{R}^2$) and indicates a better regression fit [59]. In fact, from our evaluation in Sec. 5.5, the **OLR** approach generates fluctuating α, and sometime the α sign is wrong as well. This suggests some of PerfConfs used in a software system are inherently hard to use online linear regression.

Kalman Filter (KF) is a recursive filter that estimates the internal parameters of a system given the noisy measurements[42]. Specifically, existing works use the Kalman Filter to reduce the noisy α obtained from online linear regression [20, 38]. Aside from **OLR**'s drawbacks, the Kalman Filter also suffers from following factors: (1). It assumes Gaussian noise, but the software performance's noise often does not follow a normal distribution[57]. (2) More importantly, it will introduce two additional parameters (process noise, and observation noise) to the system and those parameters are hard to be set correctly without a strong control background. Therefore, Kalman Filter is less performant and user-friendly.

AgileCtrl approaches the ideal α gradually with fewer assumptions (like normal distribution) as well as introducing one extra or none to AdapConf. We leverage three insights to guide our *AgileCtrl* design for dynamic α adjustment. **First**, *AgileCtrl* separates α value tuning from α sign correction, since the sign determines convergence/divergence, while the value determines the rate of convergence (or settling time from the control aspect). **Second**, it is much easier and more robust to determine enlarging or reducing α value iteratively than acquiring optimal α value directly. In fact, as shown in Figure. 5.4, the possibility of enlarging or reducing α correctly is over 80% while the possibility that directly generates a proper α based on *OLR* (off by at most 5× than the optimal) is only around 1/3. Not to mention that 1/3 of directly generated α from *OLR* is too conservative/aggressive, and the remaining 1/3 produces the opposite α sign. **Third**, we leverage Model reference Adaptive Control [63] used in control theory to build another feedback loop to evaluate the controller's performance and propose the simple adaptation rule to estimate α.

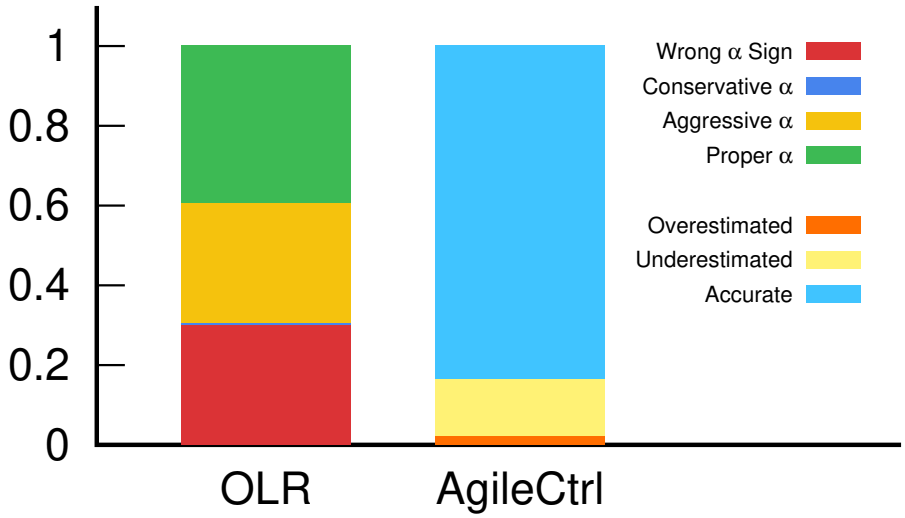

Figure 5.4: The statistics difference between OLR and *AgileCtrl*. For OLR, we checked the percentage of getting an α with opposite sign, a conservative α (x5 larger than optimal), an aggressive α (x5 smaller than optimal), and remaining as a proper α. For *AgileCtrl*, we checked the percentage of accurate estimate/overestimate/underestimate predicted α against the optimal.

Specifically, during the runtime, we measure the current performance and predict the performance based on the current model. We expect the predicted performance is close to actual measurement if such a system model is accurate enough. Then, we can compare the predicted performance against the actual to determine whether current α is too large or too small, and adjust the α value accordingly till α reflects the actual relationship between configuration and performance. Essentially, it is an another feedback controller that adjusts α gradually.

In control theory, the *MIT Rule* propose a self-adjustable parameter θ as a function of the loss function, which minimize the prediction error and the actual measurement error [56]. Specifically, for our performance model, the most straightforward strategy is setting parameter θ to the ratio between prediction error and the actual measurement error to update α. The intuition is that if the prediction error is far away from the actual measurement error, then the current α is also far away from the actual α as well. To avoid unnecessary large tuning when the current performance is close to the goal, we regulate the above ratio using $|\frac{\mathbb{G}-C_k}{\mathbb{G}}|$ as the power. Therefore, we will use the following equation to update the α:

$$\alpha_{k+1} = \alpha_k \cdot \{|\frac{C_k - C_{k-1}}{P_k - C_{k-1}}|\}^{|\frac{\mathbb{G}-C_k}{\mathbb{G}}|}, \tag{5.3}$$

where C_k is current performance, C_{k-1} is the previous performance, P_k is predicted performance and $\mathbb{G}$ is the performance goal. Specifically, we will update the alpha based on the following Algorithm 2. As long as the ratio $\frac{C_k-C_{k-1}}{P_k-C_{k-1}}$ is close $\frac{\alpha_a}{\alpha_k}$, where α_a is the actual alpha based on performance impacts, then α_k will converge to α_a.

Algorithm 2: Dynamic α with dynamic step

Input : $\mathbb{G}$ – Performance Goal
Output: α – Updated alpha value

1 Measure current performance C_k at time k
2 Predict further performance P_{k+1} at time $k+1$ based on C_k
3 Update $\alpha_{k+1} = \alpha_k \cdot \{|\frac{C_k-C_{k-1}}{P_k-C_{k-1}}|\}^{|\frac{\mathbb{G}-C_k}{\mathbb{G}}|}$

5.4.3 Virtual Goal (vg) Module

We need to adjust the virtual goal (vg) during the runtime as well, especially when the environment becomes more chaotic. The vg should only reflect the inherent system performance noise rather than performance variation caused by different configurations. Therefore, we need to have the same configurations when calculating vg. However, It is non-trivial to compute vg during the runtime. Recent performances are usually measured under different configurations since (1) as discussed in **OLR** for dynamic α adjustment , some PerfConfs change passively, and (2) the controller also adjusts all kinds of PerfConfs continuously as long as the goal is not reached. Ignoring this challenge will lead to a large error in a virtual goal, which could either impede the performance or even leading to unreliability.

To deal with this problem, we leverage the system configuration-performance model. Specifically, we can **calibrate** the performance by compensating the configuration difference. Specifically, we take i pairs of ¡performance, configuration¿ for virtual goal adjustment, e.g. $< p_{k-i+1}, c_{k-i+1} >, \cdots, < p_{k-1}, c_{k-1} >, < p_k, c_k >$. We calibrate all performance as if they are measured with configuration c_k based on the system model, and result in a adjusted pairs $< p'_{k-i+1}, c_k >, \cdots, < p'_{k-1}, c_k >, < p_k, c_k >$. Then, we can simply recalculate the σ_i and m_i based on adjusted performance $p'_{k-i+1}, \cdots, p'_{k-1}, p_k$, and recalculate the virtual goal as usual.

5.5 Evaluation

5.5.1 Evaluation Methodology

Table 5.2: Benchmark suite and runt-time setting for module evaluation. In issue description, the main constraint that users complain about is put earlier, the trade-off description is later, and finally the metrics of constraint performance (goal) and trade-off performance.

ID	Issue Description	Run-time Setting		
		α sign	α value	vg
HD4995	`content-summary.limit` limits #files traversed before `du` releases big lock. Too big, write blocked for long; Too small, `du` latency hurts. Write latency; `du` latency	Manually flip α sign	Limit CPU usage to 100%, 50%, 20%, 10%	Reduce vg to 20%, 40% 60%, 80%
HB2149	`global.memstore.lowerLimit` decides how much memstore data is flushed. Too big, write blocked for too long; Too small, write blocked too often. Tail write latency; The number of violated write latency			
HB3813	`ipc.server.max.queue.size` limits RPC-call queue size. Too big, OOM; Too small, read/write throughput hurts. Used memory; Write throughput		Increase request size to 1MB, 2MB, 5MB, 10MB	
HB6728	`ipc.server.response.queue.maxsize` limits RPC-response queue size. Too big, OOM; Too small, read/write throughput hurts. Used memory; Write throughput			
CA6059	`memtable_total_space_in_mb` limits the memtable size. Too big, OOM; Too small, write latency hurts. Used memory; Write latency			

Machines We have used the Chameleon Cloud [44] for our experiments. Each server has 2 12-core Intel Xeon E5-2670 v3 CPU with 128GB RAM. Ubuntu 16.04, JVM 1.7, and JVM 1.8 (compatible with CA6059) are installed.

Baseline and Benchmarks We compare *AgileCtrl* with *SmartConf* from three aspects: α sign, α value and vg. We evaluate all benchmarks used in *SmartConf* expect 2820 [2]. *AgileCtrl* evaluate 5 benchmarks from Cassandra, HBase, and HDFS, as shown in Tab. 5.2. Among those benchmarks, HD4995 and HB2419 have constraint on latency, and the other benchmarks (HB3813, HB6728 and CA6059) have hard-limit constraint on memory usage to avoid the out-of-memory failures.

Workload and Run-time For database-related benchmarks Hbase and Cassandra (HB3813, HB6728, HB2149 and CA6059), standard performance testing framework YCSB[17] is used, while we use TestDFSIO[34] for file system related benchmark HDFS (HD4995). As shown in Table 5.2, we consider a separated run-time settings for evaluating α sign, α value, and vg modules. Furthermore, for benchmarks (HB2149 and HD4995), which the main constraint is about latency, so we limit CPU resources by a factor of $\times 1$, $\times 2$, $\times 5$ and $\times 10$ (namely, limit CPU usage to %100, %50, %20, and %10). For benchmarks (HB3813, HB6728 and CA6059), as the main constraint is memory usage, we increase the every request size by the same factor (from 1MB up to 10MB).

We present a detailed module evaluation on *AgileCtrl* for α sign correction, α value adjustment, and vg adjustment. Then, we take HB3813 as an example to analyze how all three modules work as a whole as well as the contribution of each module.

2. For 2820, the main goal to restrict the maximum OOD exceptions within one job smaller than the threshold to avoid job failure, and the exception is limited by the number of machines the job tried. *AgileCtrl* is expected to work well for large cluster. However, given the small cluster size in our experiment, *AgileCtrl* failed to correct the improper AdapConfs fast enough. Further discussions on *AgileCtrl* limitation are in Section 5.5.4.

5.5.2 Module Evaluation

In this section, we consider how each module of *AgileCtrl* improves system reliability and performance. The evaluation of α Sign and Virtual Goal (vg) Modules are shown in Tab. 5.2, and that of α Value Module is presented in Tab. 5.4 and Tab. 5.5.

α Sign Module

The wrong α sign mostly comes from insufficient profiling or human mistake based on our experiment rather than workload or run-time resources changes. Therefore, we flip the initial α to the opposite sign to simulate those situations resulting in wrong α sign and compare *SmartConf* and *AgileCtrl*. We consider both functionality (can we detect α sign is wrong or not) and performance (can we reach the desired goal while improve the trade-off performance due to wrong α sign) aspects.

As shown in Table.5.3, *AgileCtrl* can successfully correct the wrong initial α value, and the constraint performance better approaches the ideal goal while improve the trade-off performance across all benchmarks. As discussed, wrong α sign will cause the constraint performance deviating from goal. *AgileCtrl* keep tracking of the moving direction of the constraint performance, and auto-reverse α sign when direction is wrong. As result, *AgileCtrl* improve the constraint performance towards expected goal while boosting the trade-off performance.

α Value Module

We consider the *AgileCtrl* from two aspects: (1)how *AgileCtrl* improves the system performance and keeps constrain satisfied while combating against run-time changes in workload and environment. (2) what is the tolerable limits of α that *AgileCtrl* can fix without causing

Table 5.3: α Sign Module Evaluation (**Goal**: the normalized main constraint performance *w.r.t* the goal (close to 1 means better). **Tradeoff**: the trade-off performance speedup *w.r.t SmartConf* (the high, the better).)

Benchmark	*SmartConf*		*AgileCtrl*	
	Goal	Tradeoff	Goal	Tradeoff
HD4995	0.13		1.08	1.12
HB2149	0.15		0.99	1.42
HB3813	0.62	1.00	0.86	1.16
HB6728	0.53		0.70	2.67
CA6059	0.08		0.81	1.28

system instability (*e.g.* , crash or performance oscillation [3]).

Table 5.4 shows how *AgileCtrl* and other alternative approaches react to different levels of workload and environment changes. In general, our baseline *SmartConf* can tolerate some run-time workload and environment changes as expected. **Online Linear Regression (OLR)** performs the worst across different benchmarks compared with all other approaches. Specifically, **OLR** works well only when measurement noise is relatively low, thus estimated alpha from **OLR** is accurate, e.g., HD4995. We also carefully tune two additional AdapConfs used in Kalman filter (**KF**) by exhaustively searching, and it can achieve similar results as *AgileCtrl*. In the contrast, ***AgileCtrl*** achieves the best for meeting performance goal without oscillations or crashes, and better trade-off performance without introducing any new AdapConfs. A detailed explanation will be provided in the next case study section.

Furthermore, we quantify the **robustness** of the *AgileCtrl* against other alternative solutions by calculating the **error tolerance**. Specifically, we will *first* calculate the ideal alpha α_{sys} based on the offline profiling. Then, we can vary the α_{ctrl} value (only its absolute value, not its sign) and find the lowest and highest boundary alpha (α_{lowest_bound} and $\alpha_{highest\_bound}$) under the same workload that system is above to instability (crash or oscillation). Therefore, the **error tolerance** (ET_l and ET_h) for the particular benchmark

3. There is no uniform definition on oscillation. In this paper, performance oscillation means the standard deviation of final performance is twice larger than that of system internal noise.

and strategy can be defined as:

$$ET_l = \frac{\alpha_{sys}}{\alpha_{lowest\_bound}}, \quad ET_h = \frac{\alpha_{highest\_bound}}{\alpha_{sys}} \tag{5.4}$$

By above definition, both ET_l and ET_h are greater than 1. The larger ET_l or ET_h is, the better ability to tolerate the errors in alpha the system has, thus the better system robustness is. In other words, the system performance is stable if α_{ctrl} is within $[\frac{\alpha_{sys}}{ET_l}, \alpha_{sys}ET_h]$. Moreover, if none of α can save the system from crash or oscillation, then we set $ET = 0$ to indicates such a solution can not be used for adjusting the alpha value.

Table 5.5 shows *AgileCtrl* can greatly extend both the lowest bound and highest bound compared with all alternative approaches, which means it can tolerate a larger range of wrong α_{ctrl} used by the controller. Kalman filter approaches achieve slightly worse performance, and online linear regression failed to provide any **robustness** in 3 out of all 5 cases.

Virtual Goal (vg) Module

For virtual goal module evaluation, we investigate how much trade-off performance speedup if the initial virtual goal is only 20%, 40%, 60% and 80% of the ideal virtual goal obtained from the profiling. As shown in Tab. 5.6, across all benchmarks, *AgileCtrl* can successfully not only meet the ideal performance goal but also improve the trade-off performance by 50% on average.

5.5.3 Case Study

We take a close look at how *AgileCtrl* handles one representative case HB3813. The overall evaluation of *AgileCtrl* considers an extreme buggy scenario (None of α sign, α value, and vg are right.) that required all modules to work together. Then, we will analyze how each

Table 5.4: Overall trade-off performance speedup *w.r.t* to ACif constraint performance converged to goal(The higher, the better.). Otherwise, system crashed(**C**) or constraint performance oscillate around the goal(**O**).

Benchmarks	Change	Level	SC	OLR	KF	AC
HD4995	Resource	×1	0.93	0.91	1.00	1.00
		×2	0.93	0.91	0.87	1.00
		×5	**O**	0.97	0.91	1.00
		×10	**O**	0.92	0.83	1.00
HB2419		×1	0.86	**O**	0.87	1.00
		×2	0.80	**O**	0.88	1.00
		×5	**O**	**O**	0.81	1.00
		×10	**O**	**O**	**O**	1.00
HB3813	Workload	×1	1.00	**C**	0.91	1.00
		×2	0.96	**C**	0.86	1.00
		×5	**C**	**C**	0.81	1.00
		×10	**C**	**C**	**C**	1.00
HB6728		×1	0.98	**C**	1.01	1.00
		×2	0.65	**C**	1.02	1.00
		×5	**C**	**C**	1.00	1.00
		×10	**C**	**C**	**C**	**C**
CA6059		×1	0.97	**C**	0.95	1.00
		×2	0.95	**C**	1.01	1.00
		×5	**C**	**C**	0.89	1.00
		×10	**C**	**C**	0.99	1.00

module functions in this representative case.

Again, in HB3813, the PerfConf `max.queue.size` limits the largest size for an Hbase RPC queue. Out-of-memory (OOM) is more likely to happen with a large queue, while RPC throughput is reduced with a small queue. The *SmartConf* alleviated the HB3813 issues to accommodate different workloads, maintain the memory consumption without OOM, and improve the system throughput. However, *SmartConf* introduces two representative AdapConfs (α and vg) to the system, where α represents the size of the average request, and vg reserved a portion of memory for safety. In fact, HB3813 contains the following challenges that are unique to self-adaptive for software configuration tuning: (1) it requires online α tuning since online request size could be much larger or smaller than offline. (2) it has a hard constraint on the performance, e.g. the memory limit cannot be violated, (3) the

Table 5.5: Overall Error Tolerance for *AgileCtrl* compared with alternative approaches (ET_l/ET_h: the lowest/highest alpha that corresponding approach can correct without causing performance oscillations or system crashes. **0**: No such alpha without causing performance oscillations or system crashes)

Ben -ch	SC		OLR		KF		AC	
	ET_l	ET_h	ET_l	ET_h	ET_l	ET_h	ET_l	ET_h
HD 4995	2	4	10^6	10^6	$5 * 10^5$	10^4	10^6	10^6
HB 2419	2	10^3	**0**	**0**	10	2	10^6	10^6
HB 3813	3	10^5	**0**	**0**	10^6	30	10^6	10^6
HB 6728	2	10^5	**0**	**0**	1.5	40	10^6	10^6
CA 6059	5	10^3	2	400	2	10^6	10^6	10^6

Table 5.6: Virtual Goal(vg) Module Evaluation of trade-off performance speedup *w.r.t SmartConf*.

Benchmark	Trade-off Performance Speedup			
	20%	40%	60%	80%
HD4995	1.16	1.20	1.05	1.00
HB2149	1.65	1.42	1.38	1.19
HB3813	1.58	1.38	1.34	1.27
HB6728	2.10	2.39	2.26	2.13
CA6059	1.86	1.46	1.09	1.04

performance (memory usage) has large variations due to JAVA GC.

Overall Evaluation

For the overall evaluation, ideally, a full combination of different α's value, α's sign, and virtual goal variations should be tested thoroughly. However, the target system is more likely to suffer from performance degradation or crash when attributes α or virtual goal deviate from the ideal setting. Therefore, we consider the following situation, where both initial α's value and initial virtual goal are 10x different from ideal, and α's sign is also flipped compared to ideal sign (Specifically, $\alpha_{initial} = -0.25$ and $vg_{initial} = 0.09$). As shown in Fig.

5.5, though the initial α and vg are both wrong from the beginning, *AgileCtrl* can quickly adjust the virtual goal back to the ideal setting (90% of max memory), and α sign is flipped to the positive and quickly converge to ideal alpha value. This demonstrates that *AgileCtrl*'s feasibility of fixing multiple errors at the same time.

α sign module is independent of other modules, since its algorithm (Alg. 1) does not require α value or vg to be accurate. The α value module will continuously update the performance model to match the real runtime scenario, and it does not depends on either the α sign or vg module. The vg module only relies on an accurate performance model from the α value module, and not the other way around. So, there is no circular dependency among all those modules. As result, all three modules are expected to function simultaneously till both α and vg converge.

Module Evaluation

Unlike the overall evaluation that all AdapConfs are incorrect at once, we only consider incorrect AdapConfs separately and take a close look at how *AgileCtrl* auto-adjust AdapConfs.

α **Sign Module**: The α sign has significant impact on controller correctness. To verify how *AgileCtrl* reverses wrong α sign, we consider two cases: (1). the α sign is opposite from the beginning and (2). we flip the α sign in the middle of the workload. The results are shown in Fig. 5.5 and 5.6 correspondingly. In both cases, *AgileCtrl* realizes that the performance is moving away from the performance goal, and is able to flip the α back.

α **Value Module**: We consider the following scenario for α value evaluation: the offline profiling workload has a request size of 1MB, while the testing workload has a request size of 10Mb. Since the online workload is x10 different from the offline one, our *SmartConf* controller failed to retain the memory usage within the limits, and the system crashed as result. We add Online Linear Regression (**OLR**), Kalman Filter (**KF**), and *AgileCtrl* (**AC**) to the *SmartConf* to adjust its AdapConf α, and the comparison results are listed as following

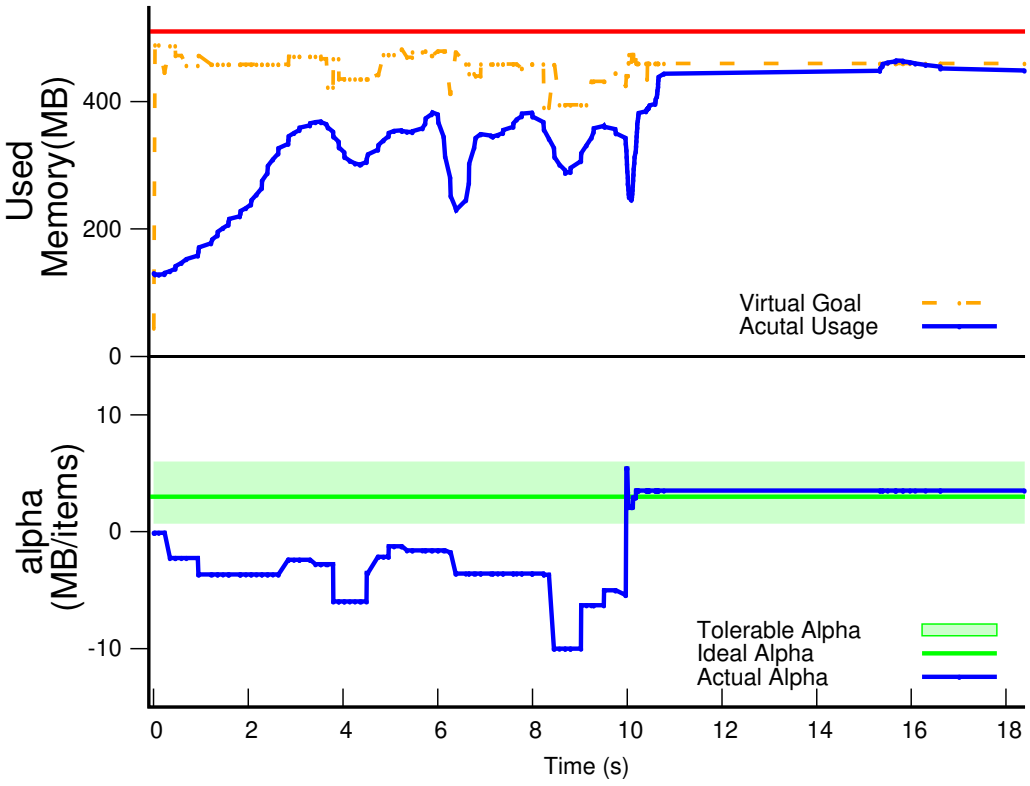

Figure 5.5: Both Initial α's value and initial virtual goal are 10x different from ideal setting, and initial α's sign is also flipped. All modules of *AgileCtrl* are enabled and able to fix wrong α and virtual goal for HB3813

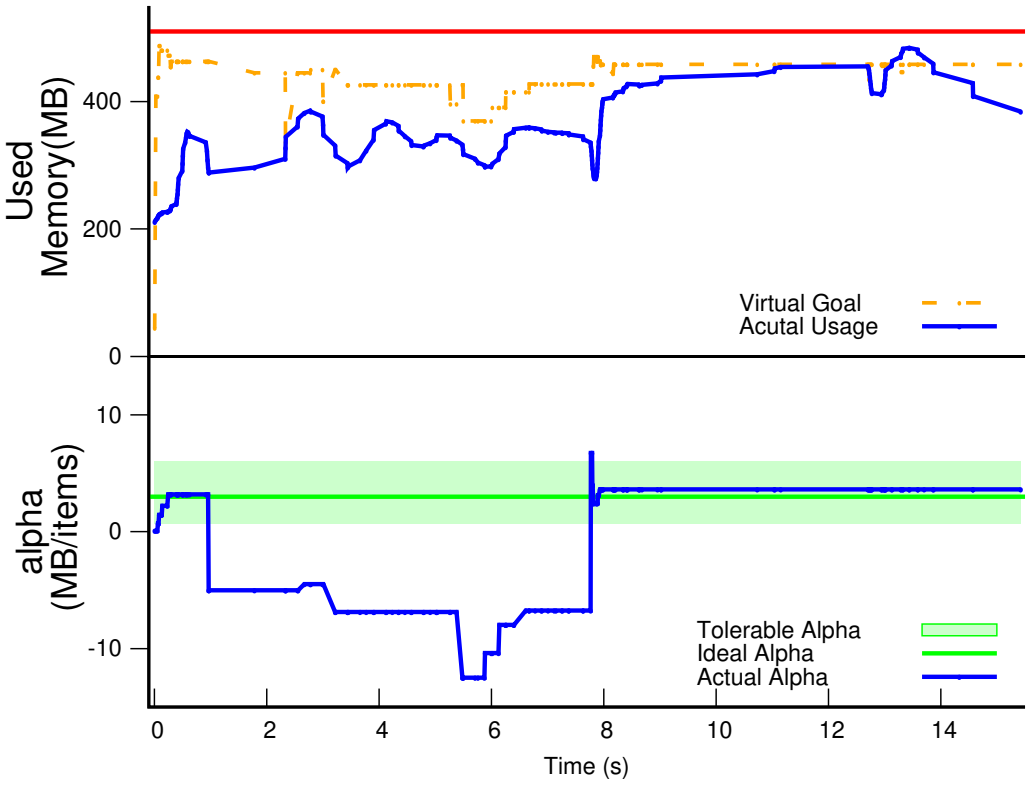

Figure 5.6: Both Initial α's value and initial virtual goal are 10x different from ideal setting, however the α's value and sign are both flipped around time 1 second. All modules of *AgileCtrl* are enabled and able to fix wrong α and virtual goal for HB3813

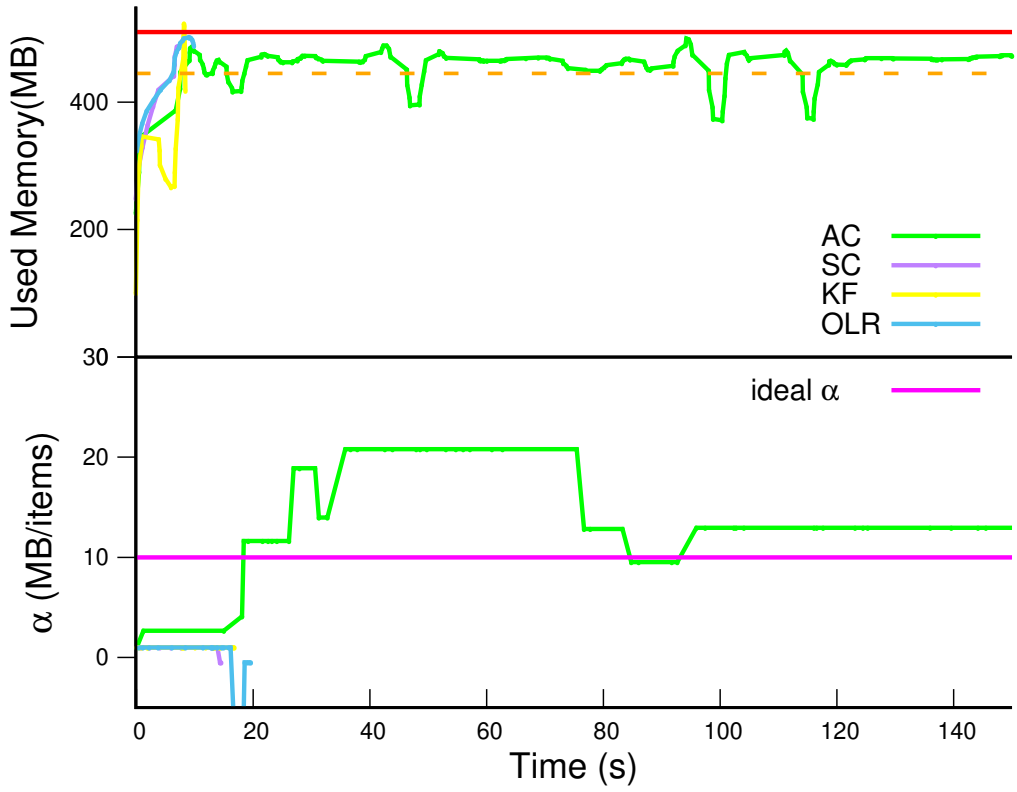

Figure 5.7: HB3813 with workload of 10MB, and a comparison on different approaches in α Value Module

and shown in Fig. 5.7.

For **OLR**, the updated α could be more than 20 times different from the ideal α. Most importantly, α becomes negative sometimes, which turns the original negative feedback into a dangerous positive feedback system. Especially, when the current memory is above the virtual goal, a negative α accelerates the memory crashes. For **KF**, it relies on α estimation from **OLR**. Given the large noises in **OLR**, **KF** also failed to bridge the gap between α used by the controller and α represented by the performance model. Also, **KF** also requires properly setting additional parameters related to the noise level. Finally, our *AgileCtrl* (**AC**) can quickly adjust the α to match the system model with the control model and then save the system from the out-of-memory crashes. From Fig. 5.8, the initial *alpha* could be either larger or smaller than ideal setting, **AC** are expected to function well in both situations.

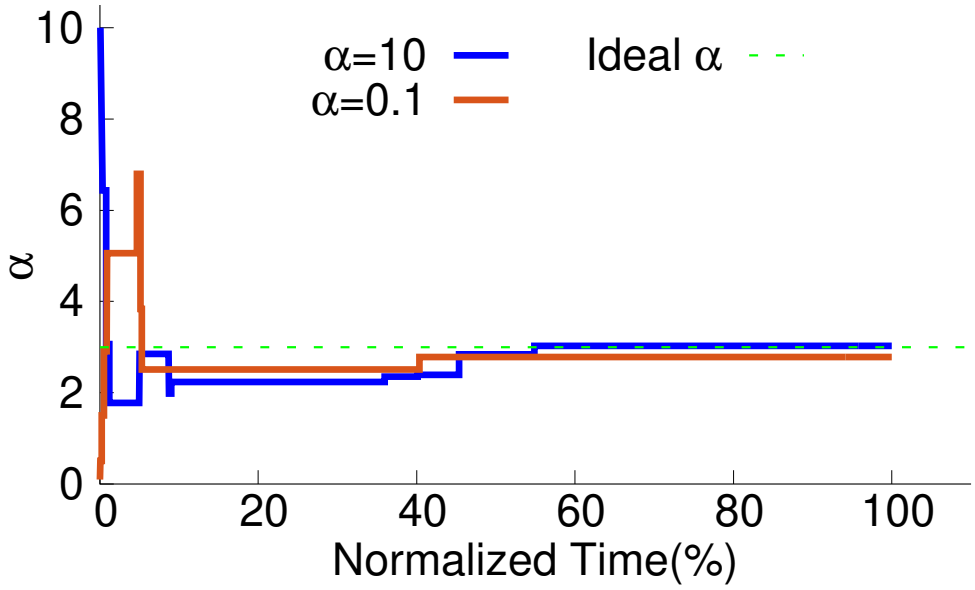

Figure 5.8: *AgileCtrl* with different initial alpha 0.1 and 10 while ideal alpha is 2.4

Virtual Goal Module: As shown in both Fig. 5.5 and 5.6, *AgileCtrl* can quickly fixed improper vg which reserves too much performance for safety purpose (too much memory reserved in HB3813). Although vg estimation depends on α estimation, our estimated vg converges while α is adjusting at the same time till both of them match the actual system model.

5.5.4 Limitations of AgileCtrl

AgileCtrl has its limitations. First, α Value Module depends on MIT rule, where itself is not globally convergent nor stable [56]. Compared with *SmartConf*, *AgileCtrl* sacrifices statistical guarantees provides by the traditional controller in return for system robustness. Though the statistical guarantees we gave up, as shown in the evaluation section, the *AgileCtrl* is expected to work empirically.

Second, the *AgileCtrl* is able to fix the both α and vg problem no matter its initial value. For example, in our evaluation, *AgileCtrl* can tolerate improper α by $10^6\times$ due to

human error. However, this error tolerance could vary in different scenarios that affect the system-performance model. For example, in HB3813, if the request size is enlarged from $1MB$ to $10^6MB \approx 1TB$, and any HBase with less than $1TB$ heap memory resources will crash directly. This is due to *AgileCtrl* is an asymptotic approach to bridge the gap between the system model and control model. It does require a certain response time to react to unexpected changes in the environment or workload. Yet, in the previous example, the memory was already used up before receiving the first full request, so there is no time for *AgileCtrl* to realize the workload changes and take any precautions. Besides response time, the computational precision should also take into consideration when we deal with the extreme AdapConfs values.

Third, *AgileCtrl* is designed to automate AdapConfs (α and vg) used in control-based self-adaptive framework. For machine learning-based self-adaptive system, they introduce a different set of AdapConfs (learning rate, weight, etc) which are not solved by *AgileCtrl*.

5.6 Conclusions

Self-adaptive frameworks have been successfully applied to automate configuration tuning with better performance. Those self-adaptive frameworks explicitly or implicitly introduce a set of AdapConfs to the system, and the proper AdapConfs setting depends not only on the understanding of AdapConfs but also complicated environment or workloads during the runtime. We argue that self-adaptive frameworks should automate not only PerfConfs but also AdapConfs. We proposed *AgileCtrl* to eliminate AdapConfs based on how well self-adaptive is performing. Our evaluation demonstrates that, compared with other well-tuned approaches, *AgileCtrl* can tolerate larger workload or environment changes while achieving a similar performance without introducing AdapConfs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Contributions

In conclusion, this dissertation makes the following three contributions.

1. Conduct a thoroughly empirical study about PerfConf problems based on developer-patches and user-posts. Specifically, 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 the dynamic workload, environmental factors, and performance tradeoffs. We further identify challenges in setting and adjusting PerfConfs: (1) about half of the PerfConfs threaten *hard* performance constraints like out-of-memory or out-of-disk problems; (2) about half of the PerfConfs affect performance *indirectly* through setting thresholds for other system variables; (3) more than half of the 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*.

2. Propose and evaluate a control-theoretic solution called *SmartConf* for automatic configuration tuning. This dissertation proposes a new configuration interface for modern software design. Specifically, *SmartConf* encourages developers to decide which PerfConf should be dynamically configured and allows users to specify the desired performance constraints. Furthermore, we explore a systematic and general control-theoretic solution to implement *SmartConf* library, so that *SmartConf* can automatically set and dynamically adjust performance-sensitive configurations to meet required operating constraints while optimizing other performance metrics.

3. Build a self-adaptive framework *AgileCtrl* to eliminate any configuration from the system. We first demonstrate that existing control-theoretic solutions could still suffer from different kinds of dynamics due to its internal attributes AdapConfs are statically determined. Then, we propose a framework—called *AgileCtrl*—by extensively modifying an existing control-theoretic framework for self-adaptive software. *AgileCtrl* monitors the quality of its adaptations and reconfigures its AdapConfs to provide even greater robustness in the face of user error or unexpectedly volatile environments.

6.2 Limitation and Future Work

The limitations and corresponding future research directions of this dissertation are:

- Both *SmartConf* and *AgileCtrl* do not work for configurations whose performance goals are about optimality instead of constrained optimality. This is due to control-theoretic solutions, like *SmartConf* or *AgileCtrl*, are designed for constrained optimization, and a proper performance goal is required. Recent works show that machine learning techniques would be a better fit in such a case. Moreover, our control-theory approaches (*SmartConf* and *AgileCtrl*) also out-perform the machine-learning based approach in terms of fast and fine-grained tuning, as well as the formal statistical guarantees. The machine-learning based approach is more suitable as regards scalability. The further work could explore the combination of machine-learning with control-theory for automatic configuration tuning.

- For *SmartConf*, some configurations might be inherently difficult to adjust dynamically, as the adjustment may cost huge code-refactoring effort or has other side-effects(*e.g.* security concerns or inconsistency). Currently, *SmartConf* does not provide a way to quantify code-refactoring effort or expected gain from the dynamical configuration. This is important for the developer to decide which configuration should be

enhanced first to be dynamically adjustable with less code-refactoring effort but more performance gain. Moreover, both *SmartConf* and *AgileCtrl* assume that the user knows that adjusting certain configurations will affect the system performance from several aspects. Improper configuration adjustment could potentially cause service outages or even cascading failure in the cloud computing era. However, such a relation sometimes is hard to be discovered even for the developer themselves. Further work could systematically explore how a configuration affects a system from any aspect including performance, security, scalability, reliability, usability, and *et al.*.

- Although *AgileCtrl* greatly extends the system reliability by eliminating any PerfConfs and AdapConfs. it is still not a panacea for all automatic configuration frameworks. *AgileCtrl* is still based on a linear regression model while its coefficient is dynamically adjustable. A more complicated model might be needed and more beneficial if we consider the interaction between different configurations and performance metrics. Further works address how to auto-adjust those AdapConfs used in complicated control models. Also, *AgileCtrl* automates AdapConfs (α and vg) used in control-based self-adaptive framework. For machine learning based self-adaptive system, they introduce a different set of AdapConfs (learning rate, weight, etc) which are not solved by *AgileCtrl*.

- *SmartConf* and *AgileCtrl* together extend the system robustness to tolerate unexpected changes in workloads or environments by continuously adjusting its own. Theoretically, they can accommodate any workload or environment changes as long as there is enough time for our framework to react since our framework is an asymptotic approach. However, in some extreme cases, there might not be enough time for our controller to react, and the system could still suffer from performance degradation or even crash. Therefore, in this situation, we need to switch to the most conservative controller model to ensure the basic functionality with less efficiency.

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] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, (5):295–310, 2004.

[4] Liang Bao, Xin Liu, Fangzheng Wang, and Baoyin Fang. Actgan: Automatic configuration tuning for software systems with generative adversarial networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 465–476. IEEE, 2019.

[5] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. Rfhoc: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1470–1483, 2015.

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

[7] André B Bondi. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice*. Pearson Education, 2015.

[8] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. Stochastic energy optimization for mobile gps applications. In *Proceedings of the 2018 26th ACM Joint Meeting*

on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 703–713, 2018.

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

[10] Rajeev Chandramohan and Anthony J Calise. Output feedback adaptive control in the presence of unmodeled dynamics. In *AIAA Guidance, Navigation, and Control (GNC) Conference*, page 4517, 2013.

[11] Anthony Siming Chen, Jing Na, Guido Herrmann, Richard Burke, and Chris Brace. Adaptive air-fuel ratio control for spark ignition engines with time-varying parameter estimation. In *2017 9th International Conference on Modelling, Identification and Control (ICMIC)*, pages 1074–1079. IEEE, 2017.

[12] 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.

[13] Haifeng Chen, Wenxuan Zhang, and Guofei Jiang. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):388–401, 2010.

[14] Yuxi Chen, Shu Wang, Shan Lu, and Karthikeyan Sankaralingam. Applying hardware transactional memory for concurrency-bug failure recovery in production runs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 837–850, 2018.

[15] Yuxi Chen, Shu Wang, Shan Lu, and Karthikeyan Sankaralingam. Applying transactional memory for concurrency-bug failure recovery in production runs. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):990–1006, 2018.

[16] Yeounoh Chung, Peter J. Haas, Eli Upfal, and Tim Kraska. Unknown examples & machine learning model generalization, 2019.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

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

[19] 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.

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

[21] 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, 2015.

[22] 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.

[23] 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):24:1–24:31, 2017.

[24] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[25] 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*, pages 561–566, 2004.

[26] Omid Gheibi, Danny Weyns, and Federico Quin. Applying machine learning in self-adaptive systems: A systematic literature review, 2021.

[27] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[28] 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.

[29] Joseph L Hellerstein. Challenges in control engineering of computing systems. In *American Control Conference, 2004. Proceedings of the 2004*, volume 3, pages 1970–1979. IEEE, 2004.

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

[31] 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.

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

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

[34] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[35] C. Imes, H. Zhang, K. Zhao, and H. Hoffmann. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 31–41, June 2019.

[36] Connor Imes and Henry Hoffmann. Bard: A unified framework for managing soft timing and power constraints. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 31–38. IEEE, 2016.

[37] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86. IEEE, 2015.

[38] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. Poet: A portable approach to minimizing energy under soft real-time constraints. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 75–86. IEEE, 2015.

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

[40] 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*, volume 40, pages 195–206. ACM, 2006.

[41] 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.

[42] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[43] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[44] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S Gunawi, Cody Hammock, et al. Lessons learned from the chameleon testbed. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 219–233, 2020.

[45] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711, 2014.

[46] Petia Koprinkova-Hristova, Yancho Todorov, Nicolae Paraschiv, Marius Olteanu, and Margarita Terziyska. Adaptive control of distillation column using adaptive critic design. In *2017 21st International Conference on Process Control (PC)*, pages 434–439. IEEE, 2017.

[47] Josua Krause, Adam Perer, and Enrico Bertini. Using visual analytics to interpret predictive machine learning models. *arXiv preprint arXiv:1606.05685*, 2016.

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

[49] Baochun Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9), 1999.

[50] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[51] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–14, 2017.

[52] 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. *IEEE TPDS*, 17(9):1014–1027, September 2006.

[53] 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):36:1–36:32, 2012.

[54] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Automated control of multiple software goals using multiple actuators. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 373–384, 2017.

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

[56] Iven MY Mareels, Brian DO Anderson, Robert R Bitmead, Marc Bodson, and Shankar S Sastry. Revisiting the mit rule for adaptive control. In *Adaptive Systems in Control and Signal Processing 1986*, pages 161–166. Elsevier, 1987.

[57] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 409–425, 2018.

[58] J. F. Martinez and E. Ipek. Dynamic multicore resource management: A machine learning approach. *IEEE Micro*, 29(5):8–17, Sept 2009.

[59] John H McDonald. *Handbook of Biological Statistics (3rd ed.)*, volume 2. Sparky House Publishing, Baltimore, Maryland, 2014.

[60] 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.

[61] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 257–267. ACM, 2017.

[62] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 289–302, 2007.

[63] B. Pasik-Duncan. Adaptive control [second edition, by karl j. astrom and bjorn wittenmark, addison wesley (1995)]. *IEEE Control Systems Magazine*, 16(2):87–, 1996.

[64] Alexey Pavlov, Nathan Van De Wouw, and Henk Nijmeijer. Convergent systems: analysis and synthesis. In *Control and observer design for nonlinear finite and infinite dimensional systems*, pages 131–146. Springer, 2005.

[65] Nirmit Prabhakar, Andrew Painter, Richard Prazenica, and Mark Balas. Trajectory-driven adaptive control of autonomous unmanned aerial vehicles with disturbance accommodation. *Journal of Guidance, Control, and Dynamics*, 41(9):1976–1989, 2018.

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

[67] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE software*, 30(4):88–94, 2013.

[68] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *CGO '05*.

[69] Atri Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. Cost-efficient sampling for performance prediction of configurable systems (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 342–352. IEEE, 2015.

[70] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. Control-theoretical software adaptation: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(8):784–810, 2017.

[71] Stepan Shevtsov and Danny Weyns. Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 229–241. ACM, 2016.

[72] Stepan Shevtsov, Danny Weyns, and Martina Maggio. Handling new and changing requirements with guarantees in self-adaptive systems using simca. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 12–23. IEEE, 2017.

[73] Stepan Shevtsov, Danny Weyns, and Martina Maggio. Self-adaptation of software using automatically generated control-theoretical solutions. In *Engineering Adaptive Software Systems*, pages 35–55. Springer, 2019.

[74] Stepan Shevtsov, Danny Weyns, and Martina Maggio. Simca* a control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(4):1–34, 2019.

[75] 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 multi-processors. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 41–50. IEEE, 2013.

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

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

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

[79] G. Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11, 2007.

[80] Yanxiang Tong, Yi Qin, Yanyan Jiang, Chang Xu, Chun Cao, and Xiaoxing Ma. Timely and accurate detection of model deviation in self-adaptive software-intensive systems.

In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 168–180, 2021.

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

[82] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[83] 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 '06: Proceedings of the seventh symposium on Operating systems Design & Implementation*, pages 117–130, 2006.

[84] Helen J. Wang, John Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Peerpressure: A statistical method for automatic misconfiguration troubleshooting.

[85] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 154–168, New York, NY, USA, 2018. ACM.

[86] Shu Wang, Zhuo Zhen, Jason Anderson, and Kate Keahey. Reproducibility as side ef-

fect. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18 Poster). IEEE Press*, 2018.

[87] 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):143–164, 2004.

[88] 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 *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Bergamo, Italy, August 2015.

[89] 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.

[90] 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.

[91] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS International conference on Measurement and modeling of computer systems*, pages 196–205, 2003.

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

[93] 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.

[94] 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.

[95] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *ACM SIGPLAN Notices*, volume 53, pages 564–577. ACM, 2018.

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

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

[98] 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.

[99] Ronghua Zhang, Chenyang Lu, Tarek F Abdelzaher, and John A Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 301–310. IEEE, 2002.

[100] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. An evolutionary study of configuration design and implementation in

cloud systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 188–200. IEEE, 2021.

[101] 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.