AgileCtrl: A Self-Adaptive Framework for Configuration Tuning

Shu Wang
LinkedIn
Sunnyvale, California, USA
shuwang@uchicago.edu

Henry Hoffmann
University of Chicago
Chicago, Illinois, USA
hankhoffmann@cs.uchicago.edu

Shan Lu
University of Chicago
Chicago, Illinois, USA
shanlu@uchicago.edu

ABSTRACT

Software systems increasingly expose performance-sensitive configuration parameters, or PerfConfs, to users. Unfortunately, the right settings of these PerfConfs are difficult to decide and often change at run time. To address this problem, prior research has proposed *self-adaptive frameworks* that automatically monitor the software's behavior and dynamically tune configurations to provide the desired performance despite dynamic changes. However, these frameworks often require configuration themselves; sometimes explicitly in the form of additional parameters, sometimes implicitly in the form of training.

This paper proposes a new framework, *AgileCtrl*, that eliminates the need of configuration for a large family of control-based self-adaptive frameworks. *AgileCtrl*'s key insight is to not just monitor the original software, but additionally to monitor its adaptations and reconfigure itself when its internal adaptation mechanisms are not meeting software requirements. We evaluate *AgileCtrl* by comparing against recent control-based approaches to self-adaptation that require user configuration. Across a number of case studies, we find *AgileCtrl* withstands model errors up to $10^6\times$, saves the system from performance oscillation and crashes, and improves the performance up to 53%. It also auto-adjusts improper performance goals while improving the performance by 50%.

CCS CONCEPTS

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

KEYWORDS

Software Configuration, Performance, Distributed Systems, Self-Adaptive Control

ACM Reference Format:
Shu Wang, Henry Hoffmann, and Shan Lu. 2022. *AgileCtrl*: A Self-Adaptive Framework for Configuration Tuning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549136

1 INTRODUCTION

Modern software systems provide great flexibility to users 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, and many other aspects of system operation [38, 67, 68, 71]. Many of these configurations can greatly affect system performance metrics, such as request latency, job throughput, memory and disk consumption, and others. Unfortunately, these performance-sensitive configurations, *PerfConfs* for short, are typically numerical variables, whose optimal settings are difficult to determine and vary based on run-time situations. Their mis-configuration easily leads to performance degradation or even crashes [20, 67]. Indeed, a recent study finds that 65% of configuration-related issues in 4 distributed systems (Cassandra, HBase, HDFS, MapReduce) involve performance concerns [67].

To ensure that software is configured optimally despite dynamic changes in operating environment, prior works have proposed self-adaptive frameworks, including, but not limited to *SmartConf* [67], DAC [69], OtterTune [66], POET [27] , SimCA [60, 62], CAPES [39], AENEAS [7], JouleGuard [23], and CALOREE [46]. Such frameworks monitor software performance and automatically adjust PerfConfs to ensure optimal operation despite unpredictable external changes. For example, AENEAS dynamically adjusts Android's GPS accuracy (`gpsPrio`) and update interval `gpsUpdate` 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 than approaches that must stick with a single PerfConf setting for their lifetime [7, 17, 26, 39, 42, 60, 62, 66, 67, 69].

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 parameters of a self-adaptive framework from the PerfConfs of the systems it should control.

For example, to use a self-adaptive framework based on control theory, users must set an **explicit** AdapConf—called the *pole*—that determines the tradeoff between adaptation's reaction time and noise sensitivity [17, 42]. If the *pole* is too small, the self-adaptive framework is more sensitive to disturbances and it may crash the software; if the *pole* is too large, the framework 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 are computed based on the training or profiling data that is provided by users. For example, in a machine-learning-based self-adaptive framework, training data determine implicit AdapConfs (*e.g. weights* used in neural networks) [7, 66, 69]. When the training inputs and environment do not match that at deployment, the self-adaptive frameworks will not react appropriately and can crash the system under control or fail to deliver the required performance [11]. For example, if the training inputs for a web server consist of only small requests, the system could crash when processing larger requests [67].

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 implicit configuration parameters. 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, the run-time workload and environment. Like PerfConfs, these AdapConfs become a new source of bugs, either from setting the explicit AdapConf incorrectly, or through providing insufficient or unrepresentative training data for the implicit AdapConf. Unsuitable AdapConfs lead to sub-optimal performance, system instability, or even system crashes. Thus, a configuration-free (both PerfConfs and AdapConf free) self-adaptive framework for general software systems is highly desired.

In this work, we propose a novel self-adaptive control framework, *AgileCtrl*, which extends general control-based self-adaptive systems [16, 17, 21, 26, 27, 41, 49, 56, 60–62, 67, 72] by automatically adjusting its internal AdapConfs so that explicit AdapConfs can be **completely** eliminated and much larger errors in implicit AdapConfs' training can be tolerated.

To better motivate *AgileCtrl*, we first experimentally show how a state of the art self-adaptive framework, *SmartConf* [67], can handle some training-deployment mismatches and yet greatly suffers from performance degradation, performance oscillation, and even crashes when mismatch becomes large; i.e., when the error in any explicit or implicit AdapConf settings becomes high (Section 3).

To help self-adaptive frameworks, *AgileCtrl*'s design leverages two insights:

- First, *AgileCtrl* adjusts its own AdapConfs based on how well its own internal adaptations are performing, measured by how accurately it predicts future performance. This is in contrast to prior work that adjusts only PerfConfs based on how closely the system meets its pre-defined goal. *AgileCtrl* adjusts both the PerfConfs—like prior work using the difference between the performance goal and the measured performance—and its internal AdapConfs—using the difference between the predicted and measured performance.
- Second, *AgileCtrl* leverages a simplified MIT rule [1] [2] to adjust AdapConfs, so that the predicted software performance stays close to the observed software performance, to ensure that *AgileCtrl*'s AdapConf adjustments always drive the software system to its overall performance goal.

Putting these two insights together, *AgileCtrl* allows a large family of self-adaptation frameworks—specifically those based on control theory—to dynamically adapt their own adaptation logic, accommodating different run time dynamics and a wide range of training or profiling deficiencies without any extra configuration requirements for users. The details are in Section 4.

Finally, we apply the *AgileCtrl* to 3 widely-used open-source distributed systems (Cassandra, HBase, and HDFS) in Section 5. Without introducing any additional AdapConf, *AgileCtrl* greatly enhances the system through the model and robustness self-adjustment. For model self-adjustment, *AgileCtrl* withstands errors up to a factor of $10^6\times$ and saves the system from performance oscillation and crashes. For uncrashed systems, *AgileCtrl* can further improve the performance up to 53%. For robustness self-adjustment, *AgileCtrl* can automatically reset the goal requirement while improving the performance by 50%.

To summarize, *AgileCtrl* makes the following contributions:

- Exposing AdapConfs used in self-adaptive frameworks as a potential source of bugs and demonstrate the importance of properly setting AdapConfs.
- Proposing that self-adaptive frameworks can be designed to reduce these potential bugs by constructing them to observe their own behavior and automatically modify their own internal AdapConfs.
- Proposing *AgileCtrl* to show how to use self-monitoring principle in self-adaptive frameworks based on control theory.
- Evaluating *AgileCtrl* against multiple advanced self-adaptive frameworks and demonstrate that *AgileCtrl* greatly enhance the system robustness with performance improvement.

2 BACKGROUND

We first discuss the common properties of self-adaptive frameworks in general. We then describe a large class of such frameworks distinguished by their use of control theory.

2.1 The Benefits of Self-Adaptive Frameworks

Modern software must deliver non-functional requirements such as performance, energy consumption, and others [6], while facing unexpected changes at run time, such as resource contention and workload fluctuations [70, 74]. Self-Adaptive frameworks help developers meet these requirements by automatically adjusting the software's PerfConfs, based on observed run time behavior.

Self-Adaptive frameworks can be generally classified as either control theory-based [2, 16, 27, 67] or machine learning-based approaches [7, 19, 66, 69], with some representative ones listed in Table 1. As the table shows, all these systems contain two or more AdapConfs that users must set, either explicitly or implicitly. Note that, *SmartConf*, *ADSS*, and *POET* all use multiple AdapConfs to automate a single PerfConf, thus the total number of AdapConfs is actually larger than that of PerfConfs [16, 27, 67].

As indicated by the last column of Table 1, these parameters can affect the framework's *internal model* or its *robustness to error*. Intuitively, the accuracy of the internal model affects the average performance achieved while the robustness to error affects the ability to tolerate variance in the underlying system. All frameworks have internal models that are used to predict how a change in a PerfConf will affect the observed performance. For example, both *SmartConf*'s α and *OtterTune*'s neural network weights W are essential for each framework to predict the performance that could be achieved with a specific PerfConf setting. Other parameters affect

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

Table 1: Partial AdapConfs used in self-adaptive frameworks (E: explicit AdapConf and I: implicit AdapConf)

System	Category	AdapConf	Type	How to set ?	Role
SmartConf [67]	Control	$pole$	**I**	Profiling	Robustness
		α	**I**	Profiling	Model
		vg	**I**	Profiling	Robustness
ADSS [16]		$pole$	**E**	Expert	Robustness
		α	**I**	Profiling	Model
POET [27]		$pole$	**E**	Expert	Robustness
		α	**I**	Profiling	Model
		q_b	**E**	Expert	Model
		m_v	**E**	Expert	Model
SimCA [60, 62]		$pole$	**E**	Expert	Robustness
		α	**I**	Profiling	Model
Brownout [34]		$pole$	**I**	Profiling	Robustness
		α	**I**	Profiling	Model
AENEAS [7]	Machine Learning	δ	**E**	Expert	Model
		σ	**E**	Expert	Model
		ϵ	**E**	Expert	Model
OtterTune [66]		W	**I**	Profiling	Model
		ds	**E**	Expert	Model
		lr	**E**	Expert	Robustness
		do	**E**	Expert	Robustness
DAC [69]		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

the framework's robustness to tolerate some error or uncertainty in terms of the unexpected workloads, environments, or constraints. For instance, in control based approaches setting the pole p properly avoids aggressive adaptation and makes the software stable during transient disturbances; in learning-based approaches setting a suitable `learning rate (lr)` avoids model over-fitting so the framework better generalizes to unseen data.

Overall, 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—this paper illustrates some problems with AdapConfs in the *SmartConf* framework in Section 3.

2.2 Control-Based Self-Adaptive Frameworks

Control theory is an increasingly popular set of techniques for implementing self-adaptive frameworks. The benefit of using control theory is that it supports formal analysis of the self-adaptive framework [18]: implementers can reason about the conditions under which the software system will or will not meet its goals. The drawback is that control-based approaches often require some expert knowledge to deploy effectively. In other words, many such approaches expose explicit AdapConfs—e.g., ADSS, POET, and SimCA from Table 1—that must be set by users.

To alleviate this burden (and reduce the total number of parameters exposed to users, including both PerfConfs and AdapConfs), some approaches have eliminated explicit AdapConfs from their interface, leaving only implicit ones that are set through profiling. SmartConf [67], Brownout [34], and DAC [69] are all examples of this idea (see Table 1). While eliminating the explicit AdapConfs makes it easier for non-experts to deploy these systems, the implicit AdapConfs must still be properly set to avoid bugs (as we demonstrate in the next section).

To further understand the challenges with implicit AdapConfs we focus on those from a state-of-the-art self-adaptive framework *SmartConf* [67] that applies proportional-integral-derivative (PID) control techniques to automatically adjust PerfConfs in distributed systems. *SmartConf* represents a large family of self-adaptive frameworks that combine a linear model with traditional PID controller [59]. Other examples of this approach include POET [27], ADSS [16], ACMA [42], Sthira [52], Tangram[53], SAMA [43], ControlVAE[58], Brownout [34], SimCA [60, 62], CALOREE [46], JouleGuard [23], and others.

SmartConf uses 3 AdapConfs for each PerfConf: model coefficient (α), pole (p) and virtual goal ratio (vg). The former two are also used in many other control-based self-adaptive frameworks, as shown in Table 1.

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

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

A positive α means increasing configuration increases the performance metric; a negative α means the opposite. The larger α's absolute value is, the more sensitive the system performance is to any configuration changes.

Because this coefficient is a key parameter of many control systems, the recent MoD2 framework automatically detects when the workload has drifted outside of a valid model and adapts this coefficient dynamically [65]. This approach uses a Kalman filter to perform this dynamic adjustment, which provides greater robustness. Unfortunately, the Kalman filter itself requires additional AdapConfs and Kalman filter-based solutions can still lead to catastrophic failures when the model error is sufficiently high (see our evaluation in Section 5).

Pole (p) is a key parameter for a PID controller as the pole determines how aggressively the controller reacts to the current performance error e_k, where c_k is the PerfConf at time k:

$$c_{k+1} = c_k + \frac{1-p}{\alpha} e_k. \tag{2}$$

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

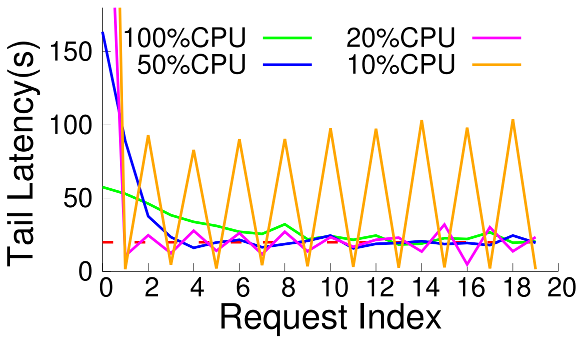

Figure 1: HD4995 under different CPU resources

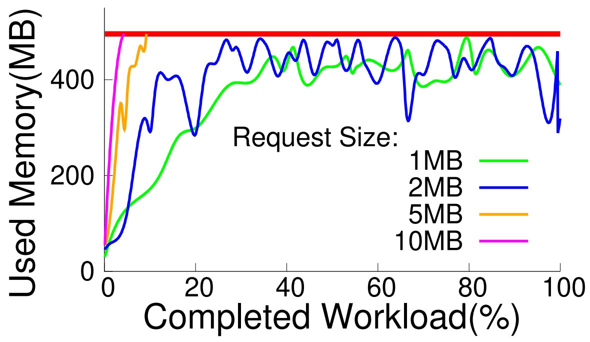

Figure 2: HB3813 under different workloads.

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. 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.

In *SmartConf*, like other systems that use implicit AdapConfs, these parameters are set through profiling runs. Users provide representative workloads and the framework collects statistics from those workloads to set the AdapConfs. The next section demonstrates some problems that can arise from this approach.

3 MOTIVATING EXAMPLE

To better motivate *AgileCtrl*, we investigate how implicit AdapConfs are used in *SmartConf* [67], which sets its AdapConfs during training then uses these fixed values throughout run-time. We show these fixed values can lead to system performance degradation or crashes when the deployment environment differs significantly from the training environment. We take two benchmarks from *SmartConf* (HD4995 and HB3813) as examples. While this specific example uses *SmartConf* to illustrate the points, the general problems and behaviors are shared by all self-adaptive frameworks that use offline profiling data to set implicit AdapConfs.

3.1 Run-Time Resource Mismatch

Our first example HD4995 reveals that, although *SmartConf* can tolerate some training-deployment mismatch in terms of run-time resources, it severely malfunctions when the mismatch increases.

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

We use the same training workload used in *SmartConf* with 100% CPU resources, which is based on the distributed file system benchmark TestDFSIO [25]. Through training, *SmartConf* computes AdapConfs <α, p, vg > as <0.00005s, 0.53, 1.00>, with α indicating the average latency to traverse a single file to be 0.00005s. In the deployment runs, we gradually decrease the CPU allocated to HDFS from 100% to 10%.

As shown in Figure 1, *SmartConf* provides some robustness when the environment is a little bit different from training: under both 100% CPU resources (green curve) and 50% CPU resources (blue curve), *SmartConf* gradually reduces the tail latency to the required goal (20s).

However, when the CPU resource drops to 20%, the *SmartConf* acts too aggressively, with the tail latency oscillating around the goal (pink curve). Even worse, with 10% CPU allocated to HDFS, the system fails to converge, with the tail latency jumping between 1s and 90s (orange curve).

The rationale is that the HDFS file/directory processing speed slows down when the CPU resources drop, causing the ideal setting of the AdapConf α to increase. Without adjusting the controller's α setting at run time, the whole system's performance oscillates.

3.2 Run-Time Workload Mismatch

Our second example HB3813 shows, when the run-time workloads differ from the training workloads, the *SmartConf* may overshoot 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 [12] workload-A with 1MB request size and 50-50 read-write ratio as the training workload, under which *SmartConf* computes three AdapConfs <α, p, vg > as <1.25MB, 0.45, 0.91>. Specifically, α characterizes the average request size inside the queue. At runtime, we gradually increase the request size from 1MB to 10MB.

As shown in Figure 2, for 1MB workload (green curve), *SmartConf* works as expected, keeping the memory consumption under the specified constraint (the red horizontal line). For a slightly larger request size 2MB (blue curve), *SmartConf* can still efficiently utilize the memory as *SmartConf* has some ability to accommodate for training-deployment workload mismatch.

However, when the request size increases to $5MB$, the system exceeds the memory limits after finishing around 10% of the total workload and then crashed. Unsurprisingly, Hbase also crashed with $10MB$ request size with only 5% workload finished.

The rationale is that the ideal setting of the AdapConf α increases with the Hbase request size. Without adjusting its α setting at run time, *SmartConf* underestimates the memory impact of the RPC queue, causing out of memory failures.

Our two motivational experiments reveal that fixed AdapConfs are sources of system under-performance or crashes. While existing self-adaptive frameworks like *SmartConf* eliminates explicit AdapConfs to avoid direct human misconfiguration, it still has implicit AdapConfs set based on user-designed training data. As our experiments show, when the training data does not match with the run-time characteristics (resources or workloads), the AdapConfs settings become problematic. Therefore, eliminating any explicit AdapConf and self-adjusting any implicit AdapConf are highly desirable for a robust self-adaptive system.

4 *AGILECTRL* DESIGN

The previous section shows how self-adaptive frameworks may fail to meet software requirements if their operating environment diverges significantly from the profiling environment. Specifically, if the environment does diverge, implicit AdapConfs representing either the framework's internal model or robustness to error might be set incorrectly, leading to performance oscillation (i.e., failure to meet the requirements) or even system crashes. The key insight of this paper is to augment such self-adaptive frameworks with an ability to observe themselves and adjust their AdapConfs to prevent this bad behavior.

We note that one approach could be to monitor different aspects of the environment (e.g., resource availability or workload properties). The problem with this approach is that there could be any number of environmental factors to observe and it is not clear ahead of time which of these factors matter or even which can be easily monitored. Thus, we propose that self-adaptive frameworks should be modified to observe both (1) how well they are meeting the performance requirements and (2) the same metrics that were used to set any implicit AdapConfs during profiling. This modification can be done internally to the framework with no change required from users. It is also robust to any environmental factor that affects the framework's ability to meet goals, and it does not require collecting any new information beyond what the framework already collects during profiling. Specifically, as AdapConfs affect either the framework's internal model or its robustness to errors, we propose to monitor (1) how well the self-adaptive framework predicts future performance on average and (2) how close the framework comes to its goal (by both mean and standard deviation). The first of these allows our approach to make dynamic adjustments to the model while the second allows adaptive adjustments to the AdapConfs that affect robustness.

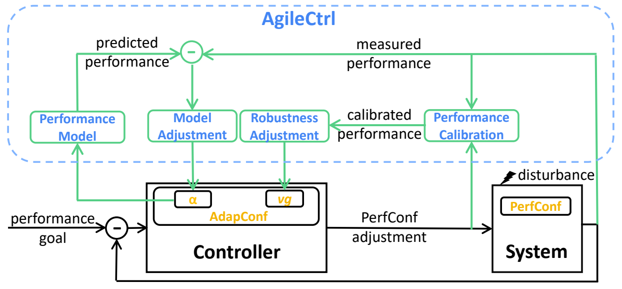

Figure 3: The overall *AgileCtrl* which extends a generic control-based self-adaptive framework.

4.1 Overview

We apply the above insights to *SmartConf*, a state-of-the-art and general self-adaptive framework. Specifically, shown in Figure 3, *AgileCtrl* collects both the measured performance from the target system and the predicted performance based on current controller status for model adjustment (α). Moreover, *AgileCtrl* leverages the stability of measured performance *w.r.t* to the goal for robustness adjustment (vg and $pole$). Together, *AgileCtrl* enhances the system with model and robustness self-adjustment during the run-time.

Our proposal is to make online adjustments to AdapConfs within a self-adaptive framework. In other words, we want to estimate the best setting for these AdapConfs based on the software's current operating conditions. In deciding how to estimate these values, we face a design choice: they can be estimated directly or indirectly based on the properties of each AdapConf [2]. For direct estimation, AdapConfs use design equations to reparameterize the model. This approach has a shorter response time [65], but it assumes that there is low noise in the samples and that inaccurate estimation will not lead to catastrophic failure. Indirect estimation, recursively estimates (i.e., slowly approaches the best value) for each AdapConf as it collects feedback. This approach is much more resistant to transient noise because a single erroneous (or outlier) observation will not change the overall trajectory in which the AdapConf setting is moving [2, 64].

The underlying ideas of *AgileCtrl* can be applied to other self-adaptive frameworks besides *SmartConf*. *AgileCtrl* can be applied with almost no changes to the increasingly large number of control based self-adaptive frameworks (e.g., [16, 27, 34, 60, 62]), which all have one or more parameters that capture the model relating PerfConf settings to performance (analogous to α in this paper). Similarly, while not all control methods use a virtual goal, all have some parameter relating to robustness to error and that parameter can be tuned following the same methodology shown below. For Multiple-Input Multiple-Output (MIMO) systems, the correlation between configuration parameters and performance metrics can be captured as a matrix instead of a scalar. *AgileCtrl* can leverage a vector of performance metrics to update the model matrix based on the difference between measured and predicted performance metrics. In other words, the principle of updating the model based on observed behavior still applies.

Interesting future work could investigate applying the same principles to tune machine learning based models, as well, although the application of the proposed methods to such models is not straightforward. Learning methods train an internal model relating PerfConfs to performance. Updating that model online would require retraining as the learning-based framework evaluates its dynamic behavior, so it is not clear that retraining cost would be worthwhile. It is possible a learning-based system would also benefit from observing the dynamic volatility in the environment and adjusting its own robustness-related AdapConfs (e.g., the learning rate) to tailor online behavior to the actual environment rather than assuming the run time will be the same as the profiling, but applying this approach would require a careful understanding of the relationship between ML hyperparameters and the desired performance.

4.2 Tuning AdapConfs Related to the Model

An accurate performance model is important for self-adaptive frameworks to provide theoretical guarantees [16]; for instance, that the software will converge to the desired performance. As shown in Sec. 3, model deviation threatens both these formal guarantees and even system availability.

For control-based self-adaptive frameworks, coefficient α (or analogous parameter) is the most important AdapConf for characterizing the system model. It has the following properties:

(1) Its sign is the primary determinant of whether the software converges to the goal, since the sign determines whether to increase/decrease the underlying PerfConf, while its magnitude determines how aggressively the adjustment moves in the direction indicated by the sign.
(2) Its magnitude usually has a wide range, as designers want meaningful parameters and prior studies show that 90% of the configuration are either integer or floating point values [67].
(3) Its performance is sensitive to α (Changing α from 0.01 to 0.001 could result in PerfConf changed by 10 times based on Equation 2).

For all these reasons, we use the most conservative self-adaptive strategy for α tuning. Furthermore, we make the design choice that it is preferable to reduce convergence speed while reducing the chance for divergence. We therefore split the process of model adjustment into two parts: determining the sign (or direction) of α and then determining the magnitude once the sign is established.

Coefficient α sign , as mentioned above, is the key for performance convergence or divergence. Moreover, it is the **only** AdapConf that captures the trend—positive or negative correlation of performance-configuration—and thus determines whether the underlying PerfConf should be bigger or smaller (Equation 1). It is also important to note that none of other AdapConfs (α's magnitude, pole p nor vg) in Sec 2.2 have the property that incorrect setting would result in tuning in the wrong direction. In other words, the trend remains the same even all other AdapConfs are wrong. Thus, we can indirectly detect the wrong α sign based on tracking the trend of how well the software system is meeting its goal.

Specifically, we can calculate the performance error e_k as the current performance *w.r.t* the performance goal (Note that all control-based approaches already track this value). Ideally, the traditional controller is asymptotically stable, which means $|e_k|$ should decrease while gradually reducing to zero [51]. Therefore, we check the trend of e_k by comparing consecutive errors—whether the last i [2] errors are decreasing or not. The noise could occasionally affect the temporary trend, but the long-term trend remains the same. Specifically, we flip the sign when the last i errors are in ascending order (Algorithm 1); i.e., the performance is diverging further and further away from the goal.

```
Algorithm 1: Wrong $\alpha$ sign detection
  Input  : $\mathbb{G}$ – Performance Goal
           $C_k$ – Current performance measured at time $k$
           $i$ – Last $i$ samples
  Output : $\alpha$ – Updated alpha sign
1 Calculate current error $e_k = |\mathbb{G} - C_k|$
2 if $e_k > e_{k-1} > \cdots > e_{k-i+1}$ then
      /* incorrect $e_k$ trend, flip the sign */
3     $\alpha_{k+1} = -\alpha_k$
4 end
```

Coefficient α magnitude determines how the magnitude in change for the underlying PerfConf. For example, if the PerfConf controls the maximum size of a software data structure, then $\alpha's$ magnitude determines how much that size might be changed at one time. Specifically, if α is too big, the controller might not be able to react to the system changes fast enough, resulting in performance degradation. Conversely, if α is too small then the system becomes unstable, causing performance oscillation or even crashes. *AgileCtrl* approaches the ideal α gradually without introducing extra AdapConfs through indirect estimation. In fact, it is much easier and more robust to determine to enlarge or reduce α magnitude iteratively than acquiring optimal α magnitude directly [1, 2, 29]. Specifically, we leverage Model Reference Adaptive Control [2, 13, 37, 50] used in control theory to build another feedback loop to evaluate the controller's performance and propose a simple adaptation rule to estimate α's magnitude.

Specifically, we make α a time varying quantity: α_k. We then determine a ratio θ, where $0 < \theta < 1$, so that we can write $\alpha_{k+1} = \theta\alpha_k$. In control theory, the *MIT Rule* proposes to adjust θ such that the quadratic loss function [44]: $J = \frac{1}{2}(P-C)^2 = \frac{1}{2}(\theta\alpha_k - \alpha_{true})^2 c^2$ is minimized. Here P and C are the predicted and current performance, c is current configuration, and α_{true} represents the actual coefficient of the system (which cannot be measured). Essentially, the loss function J represents the error between the predicted and current performance—we expect the predicted performance to be the same as the actual measurement if α_k is accurate. Thus, we can approximate α_{true} by noting that if the predicted performance is far from the current performance, then the current α_k must also be far from α_{true} as well. Specifically, the true value for θ is $\frac{\alpha_{true}}{\alpha_k}$, but since we cannot measure α_{true}, we approximate the ratio of the true to current α_k as $\frac{\alpha_{true}}{\alpha_k} = \frac{(\alpha_{true}*c_k+b)-(\alpha_{true}*c_{k-1}+b)}{(\alpha_k*c_k+b)-(\alpha_k*c_{k-1}+b)} = \frac{C_k-C_{k-1}}{P_k-(\alpha_k*c_{k-1}+b)} \approx \frac{C_k-C_{k-1}}{P_k-C_{k-1}}$. Of course, we assume the system is subject to noise so we regularize this approximation using $|\frac{\mathbb{G}-C_k}{\mathbb{G}}|$

[2] For *AgileCtrl*, we set i = 4. See Sec. 5.4 for more discussions.

as an exponent on this ratio. When the current performance already meets the goal, then the exponent is close to 0 and α remains the same. When the current performance is far away from the goal, the exponent is close to 1 which allows maximum changes on α. Putting this all together, we indirectly estimate α_k as:

$$\alpha_{k+1} = \alpha_k \cdot \{|\frac{C_k - C_{k-1}}{P_k - C_{k-1}}|\}^{|\frac{\mathbb{G}-C_k}{\mathbb{G}}|}, \tag{3}$$

where C_{k-1} and C_k are the previous and current performance, P_k is the predicted performance and $\mathbb{G}$ is the performance goal. Specifically, we update α based on Algorithm 2. As long as the ratio $\frac{C_k - C_{k-1}}{P_k - C_{k-1}}$ is close $\frac{\alpha_{true}}{\alpha_k}$, then we prove that $a_0 < \cdots < \alpha_{k-1} < \alpha_k < \cdots < \alpha_{true}$ by induction, which means α_k will converge to α_{true}.

Algorithm 2: Dynamic α Adjustment

```
Input  : 𝔾 – Performance Goal
         C_k – Current performance measured at time k
Output : α – Updated alpha magnitude
1 Predict further performance $P_k$ at time $k$ based on $C_{k-1}$
    using Equation 1
2 Update $\alpha_{k+1} = \alpha_k \cdot \{|\frac{C_k - C_{k-1}}{P_k - C_{k-1}}|\}^{|\frac{\mathbb{G}-C_k}{\mathbb{G}}|}$
```

SmartConf assumes a linear model (Equation 1) with bounded errors for PerfConf, and the linearity assumption can cause failures shown in Section 3. *AgileCtrl* relaxes such assumption through dynamically updating the approximation (Algorithm 2).

4.3 Tuning AdapConfs Related to Robustness

Besides model accuracy, robustness is another important property of any self-adaptive system. The robustness means the system can accommodate unexpected changes in workload or environment.

For control-based self-adaptive framework, robustness is captured by both the pole p and the virtual goal, vg. For *AgileCtrl*, the pole p does not need to be adjusted dynamically, since (1) it is set to tolerate erros in α which are already addressed in Sec. 4.2 and (2) α and p are correlated and together constitute a coefficient that determines how aggressively the controller reacts to the performance error. Therefore, $p = 0$ is used in *AgileCtrl*.

Recent works [63, 67] introduce a virtual goal (vg) that is smaller than actual goal to avoid overshooting, and it can be calculated based on inherent system noise (Sec. 2). Unlike α, vg is in the range $[0, 1]$. Also, vg itself characterizes the system noise, which can be statistically estimated. Therefore, vg can be self-adjusted through direct estimation at run time.

To re-calculate vg, one needs to compensate for performance variation caused by different configurations, since PerfConfs are continuously changed by self-adaptive framework. To solve it, we leverage the system configuration-performance model. Specifically, we **calibrate** the performance by compensating the measured performance difference with the configuration differences, as shown in Algorithm 3. We take N [3] pairs of <performance, configuration> for virtual goal adjustment. We calibrate performance P as if it is measured with configuration C_k based on the system model (i.e., based on α_k as computed above) and obtain calibrated performance P' (line 2-5). Then, we can simply recalculate the σ_k and m_k based on calibrated performance and recalculate the virtual goal as usual.

[3]For *AgileCtrl*, we set N = 40 suggested by sample-size rule-of-thumb [24]. See Sec. 5.4 for more discussions.

Algorithm 3: Virtual Goal Self-Adjustment

```
Input  : P – Current Performance
         C – Current Configuration
         N – Last N samples
Output : vg_k – Updated virtual goal
1 s = 0
2 while s < N do
3     Calibrate $P_{k-s}$ to $P'_{k-s} = P_{k-s} - \alpha_k(C_{k-s} - C_k)$
4     s++
5 end
6 Calculate standard deviation $\sigma_k$ and mean $m_k$ of $P'$
7 Update $vg_k = 1 - \frac{1}{N}\sum_1^N \frac{\sigma_k}{m_k}$
```

4.4 Putting It All Together

All above-mentioned components are integrated into *AgileCtrl* and can function seamlessly. In fact, α sign correction is independent of others since it does not require α value or vg to be accurate. The α magnitude adjustment will continuously update the performance model to match the run-time, and it does not depend on either the α sign or vg adjustment. The vg adjustment only relies on an accurate performance model. There is no circular dependency among all those components. As result, all components in *AgileCtrl* operate together to achieve a self-adaptive system.

5 EVALUATION

5.1 Evaluation Methodology

Machines We used the Chameleon Cloud [33] for our experiments. Each server has 2 12-core Intel Xeon E5-2670v3 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 magnitude and vg. We evaluate all benchmarks used in *SmartConf* except for MR2820[4], as shown in Table 2. Among those benchmarks, HD4995 and HB2419 have a constraint on latency, and the other three benchmarks have hard-limit constraints on memory usage to avoid out-of-memory failures.

Workload For database-related benchmarks Hbase and Cassandra (HB3813, HB6728, HB2149, and CA6059), standard performance testing framework YCSB [12] is used, while we use TestDFSIO [25] for file system related benchmark HDFS (HD4995).

Run-time As shown in Table 2, we consider a separated run-time settings for evaluating model self-adjustment (α sign and magnitude adjustment) and robustness self-adjustment (vg adjustment).

[4]For MR2820, the main goal is 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 a 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 5.4.

Table 2: Benchmark suite and run-time setting for evaluation.

ID	Issue Description (the primary constraint is put earlier; the trade-off constraint is later)	Metrics		Run-time Setting		
		Primary	Secondary	α sign	α magnitude	vg
HD 4995	`content-summary.limit` limits #files traversed before du releases locks. 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%
HB 2149	`global.memstore.lowerLimit` decides how much memstore is flushed. Too big, write blocked for too long; Too small, write blocked too often.	Tail Latency	# Violated Latency			
HB 3813	`ipc.server.max.queue.size` limits RPC-call queue size. Too big, OOM; Too small, read/write throughput hurts.	Memory	Throughput		Increase size to 1/2/5/10 MB	
HB 6728	`ipc.server.response.queue.maxsize` limits RPC-response queue size. Too big, OOM; Too small, read/write throughput hurts.	Memory	Throughput			
CA 6059	`memtable_total_space_in_mb` limits the memtable size. Too big, OOM; Too small, write latency hurts.	Memory	Latency			

Table 3: α Sign Evaluation (Primary: the normalized primary performance *w.r.t* the goal; the closes to 1 the better. Secondary: the secondary trade-off performance speedup *w.r.t* *SmartConf*; the larger, the better.)

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

For α sign: The wrong sign mostly comes from insufficient profiling or human mistakes, which we simulate by flipping the sign of the initial α, rather than workload or run-time resources changes.

For α magnitude: α magnitude are usually affected by different types of runtime settings. For benchmarks (HB2149 and HD4995), their primary performance metric is about latency which is directly affected by CPU resources. Therefore, we limit CPU resources by a factor of ×1, ×2, ×5, and ×10 (namely, limit CPU usage to 100%, 50%, 20%, and 10%) with Linux CPULimit Tool. For benchmarks (HB3813, HB6728 and CA6059), their primary metric is memory usage, which is affected by workload. Therefore, we increase every request size by the same factor (from 1MB up to 10MB).

For vg: Since vg reflects the chaos of the runtime environment and an improper vg means a mismatch between offline and online virtual goal setting, we reduce the initial virtual goal by 20%, 40%, 60%, and 80%, and compare *SmartConf* and *AgileCtrl*.

5.2 *AgileCtrl* Evaluation

5.2.1 Model (α) Self-Adjustment: Our experiment with the wrong α sign shows that *AgileCtrl* performs much better than the baseline *SmartConf*. As shown in Table 3, *SmartConf* can achieve only 13–62% of the primary performance goal. On the contrary, *AgileCtrl* keeps tracking of the moving direction of the primary performance, and hence can auto-correct the sign of α and achieves mostly more than 80% of the primary performance goal even with an incorrectly initialized α sign. Moreover, *AgileCtrl* also improves the secondary performance compared with *SmartConf*.

Table 4: α Magnitude Evaluation. (C: the system crashes. O: primary performance oscillates around the goal. Neither C nor O: the normalized secondary performance speedup with *AgileCtrl* being 1; the higher, the better.)

Bench	Change	Level	SC	OLR	KF	*AgileCtrl*
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

Our experiment with the wrong α magnitude compares *AgileCtrl* with not only *SmartConf* but also two prior techniques that adjust α through *Online Linear Regression (OLR)* [16] or *Kalman Filter (KF)* [27, 65]. As shown in Table 4, online linear regression (OLR) performed the worst, causing system crashes (mainly due to out-of-memory problems) or severe performance oscillation in all but one benchmark. This result shows that directly re-setting the value of α using the same offline linear regression algorithm used during profiling, as in OLR, does not work. *SmartConf* (SC) is just slightly better than OLR. Kalman Filter (KF) can eliminate most of the crashes and oscillations encountered by OLR and SC, but still performs significantly worse than *AgileCtrl*. It also introduces extra configuration tuning, which we explain below. In contrast, *AgileCtrl* only failed in one extreme case of HB6728, where other strategies also failed. *AgileCtrl* performs the best for meeting primary performance goals without oscillations or crashes, and enabling better secondary performance without introducing any new AdapConfs.

Kalman Filter is a recursive filter that indirectly estimates the internal parameters of a system given noisy measurements [32]. In general, it suffers from two limitations compared with *AgileCtrl*. (1) It assumes Gaussian noise, but software performance's noise often does not follow a normal distribution [45]. (2) It contains two additional parameters, process noise and observation noise, that are hard to be set correctly. In our experiment, we actually tuned these two additional parameters by exhaustively search.

Finally, we quantify the model robustness of *AgileCtrl* against other alternative solutions by calculating their **error tolerance**. Specifically, we *first* calculate the ideal alpha α_{sys} based on the offline profiling. Then, we vary the α_{ctrl} magnitude and find the lowest and highest boundary alpha (α_{lowest_bound} and $\alpha_{highest\_bound}$) under the same workload that system is about to crash or oscillate. Therefore, the **error tolerance** (ET_l and ET_h) for the particular benchmark and strategy can be defined as:

$$ET_l = \frac{\alpha_{sys}}{\alpha_{lowest\_bound}}, \quad ET_h = \frac{\alpha_{highest\_bound}}{\alpha_{sys}} \tag{4}$$

By 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 α the system has, thus the better system robustness is. In other words, the system is stable if α_{ctrl} is within $[\frac{\alpha_{sys}}{ET_l}, \alpha_{sys}ET_h]$. If none of α can save the system from crash or oscillation, we set $ET = 0$ to indicate such a solution can not be used for adjusting the alpha magnitude.

Table 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 crashes)

Benchmark	SC		OLR		KF		*AgileCtrl*	
	ET_l	ET_h	ET_l	ET_h	ET_l	ET_h	ET_l	ET_h
HD4995	2	4	10^6	10^6	$5 * 10^5$	10^4	10^6	10^6
HB2419	2	10^3	**0**	**0**	10	2	10^6	10^6
HB3813	3	10^5	**0**	**0**	10^6	30	10^6	10^6
HB6728	2	10^5	**0**	**0**	1.5	40	10^6	10^6
CA6059	5	10^3	2	400	2	10^6	10^6	10^6

Table 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. Well-tuned Kalman filter approach achieves slightly worse performance, and online linear regression failed to provide any **robustness** in 3 out of all 5 cases. The baseline solution *SmartConf* can only provide only limited tolerance but it is more stable than online linear regression.

5.2.2 Robustness (virtual goal) Self-Adjustment: Specifically, we investigate how much secondary performance improved if the initial virtual goal is only 20%, 40%, 60%, and 80% of the ideal virtual goal obtained from the profiling. Our experiment shows, across all benchmarks, *SmartConf* fails to correct the wrong initial vg, which

Table 6: Virtual Goal (vg) Evaluation: the speedup on the secondary performance metric *w.r.t SmartConf* under different initial virtual goal ratios.

Benchmark	Initial Virtual Goal Ratio			
	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

leads to poor secondary performance as well. *AgileCtrl* resets the wrong initial goal requirement so that the primary performance meets the ideal goal, while improving secondary performance by 50% on average, as shown in Table 6.

5.3 Case Study

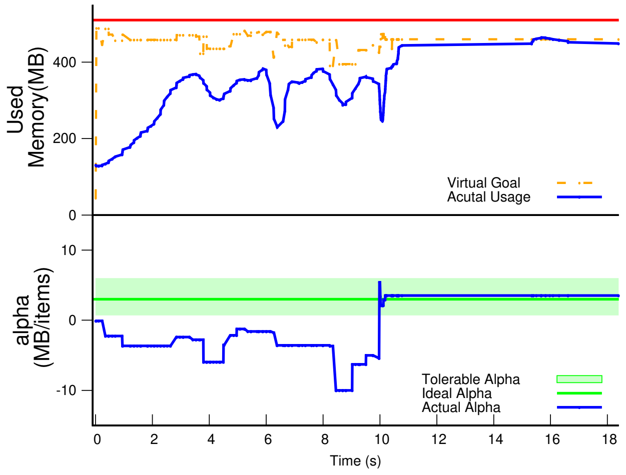

Figure 4: Both Initial α's magnitude and initial virtual goal are 10x different from the ideal setting, and the initial α's sign is also flipped. All modules of *AgileCtrl* are enabled and able to fix wrong α and virtual goal for HB3813

We take a close look at how *AgileCtrl* handles one representative case HB3813. We considers an extreme buggy scenario. None of α sign, α magnitude, and vg are right, and it required all components to work together. Then, we analyze how each component 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, 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 performance (memory usage) has large variations due to JAVA GC.

Ideally, a full combination of different α's magnitude, 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 magnitude and initial virtual goal are 10× different from ideal, and α's sign is also flipped compared to ideal sign (Specifically, $\alpha_{initial} = -0.125$ and $vg_{initial} = 0.091$). As shown in Fig. 4, 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 magnitude. This demonstrates that *AgileCtrl*'s feasibility of fixing multiple errors at the same time.

5.4 Limitations of *AgileCtrl*

AgileCtrl has its limitations. First, α magnitude adjustment depends on MIT rule, where itself is not globally convergent nor stable [44]. Compared with *SmartConf*, *AgileCtrl* sacrifices statistical guarantees provided by the traditional controller in return for system robustness. Though the statistical guarantees we gave up, as shown previously, the *AgileCtrl* outperforms *SmartConf* empirically.

Second, ideally, *AgileCtrl* can fix the both α and vg problem no matter its initial magnitude. For example, in our evaluation, *AgileCtrl* can tolerate improper α by $10^6\times$ due to human error. 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 $10^6 MB \approx 1TB$, 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, we introduce two parameters: i (Algorithm 1) and N (Algorithm 3). Unlike other AdapConfs, these two parameters provide statistical guarantees. For example, in HB3813, when the setting of i is smaller than the default setting 4, we observe that a large percentage of executions fail due to frequent sign flipping (90% for $i = 2$ and 60% for $i = 3$). When i is not smaller than the default setting, none of the executions fail. With a larger i, Algorithm 1 is less sensitive to the system noise but takes a longer time to detect the incorrect α sign. Similarly, for sample size N that is less than the default setting 40, like when N is 10 or 20, all executions would fail due to inaccurate vg estimation. In general, selecting a suitable sample size is covered by best practices in statistics [24, 30, 31, 48] and is out of the scope of this paper.

Fourth, *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 existing *AgileCtrl*.

6 RELATED WORK

Automatic Configuration Tuning Large modern software contains hundreds to thousands of configurations and those configurations are usually badly documented and are hard for both developer and user to set [67]. Great efforts have been made towards automatic configuration tuning in recent years. Specifically, existing approaches can be classified as, model-based tuning, search-based tuning, and learning-based tuning. Model-based tuning [3, 22] relies on the accurate performance model of the system. Such model synthesis usually requires domain-specific knowledge to abstract software with a mathematical model. The model is highly abstracted and very specific to the analyzed software. Search-based tuning [47, 73] treats the software as a black-box and uses a searching algorithm to find the optimal settings. However, those approaches suffer from exploration and exploitation problems and are not suitable for dynamic adjustment during the runtime. Learning-based tuning [10, 57, 67, 69] usually builds a performance model based on the profiling, and finds the best configuration during the runtime. The performance models could be regression model [57, 67] or machine learning model [10, 69].

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* [67], 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 specifically 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.

Machine Learning Machine learning has been widely used to learn the system performance model as the foundation for searching the optimal performance [10, 14, 15, 57, 66, 69]. In general, those approaches require a huge amount of effort on data collection, *e.g.* tens of hours for one workload [54, 69], let alone infinitely many disturbances, workloads, and environment. A limited amount of training data is not enough for the machine learning technique to work in dynamics. The ability to adjust the machine model itself based on dynamics during the runtime is needed. In contrast, control theory is designed for system dynamics with formal guarantee [21]. Empirical studies comparing control and learning solutions have observed that control techniques approach the goal with less error [40]. 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 [36]. Therefore, machine learning is not suitable for dealing with dynamics.

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 for the environment or workload changes.

Though various adaptive control techniques have been proposed, most of them are designed for specific systems. For example, it has been successfully applied to Aerial Vehicles [55], Engine Control [9], Distillation Column [35] and so on. However, the adaptive control is hard to be generalized to different applications because of different underlying system models. *AgileCtrl* is designed specifically to enhance general control frameworks [16], which approximates the system model as a linear model without taking the high-order correlations into consideration. *AgileCtrl* does not require additional assumptions other than the control framework. Moreover, AdapConf is adjusted based on its internal performance instead of analyzing the external system and environment. As demonstrated in the evaluation section, with different benchmarks, disturbances, and performance goals, *AgileCtrl* is robust enough across different deployment scenarios and different environmental settings that cause errors in AdapConfs. Thus, *AgileCtrl* itself is general with respect to different applications as well as different types of system disturbances.

Most importantly, adaptive controllers are inherently nonlinear and complex [8] and prior researches focus on establishing stability analysis of the adaptive control. However, most adaptive control system introduces additional parameters, which require the control expert to set. For non-expert, those parameters are hard to set and error-prone. Previous work, *CoPPer*, *POET*, and *MoD2* [27, 28, 65], took the first step for adjusting controller key parameters using Kalman filter. However, it requires setting two additional parameters, namely, process and observation noise with the assumption of Gaussian distribution. *AgileCtrl* aims at allowing non-expert to use without setting additional parameters.

7 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 automatically modify 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.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback and comments. This research is supported by NSF (grants CCF-2119184, CCF-2028427, CNS-1956180, CNS-1952050, CCF-1837120, CCF-1823032, CNS-1764039), ARO (grant W911NF1920321), DOE (grant DESC0014195 0003), DARPA (grant FA8750-16-2-0004), the CERES Center for Unstoppable Computing, the Marian and Stuart Rice Research Award, gifts from Microsoft and Facebook, and computation resources from Chameleon.

REFERENCES

[1] Karl J Åström. 1984. LQG SELF-TUNERS. In *Adaptive Systems in Control and Signal Processing 1983*. Elsevier, Amsterdam, Netherlands, 137–146.
[2] Karl J Åström and Björn Wittenmark. 2013. *Adaptive control*. Courier Corporation, Chelmsford, MA, USA.
[3] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30, 5 (2004), 295–310.
[4] Liang Bao, Xin Liu, Fangzheng Wang, and Baoyin Fang. 2019. Actgan: Automatic configuration tuning for software systems with generative adversarial networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Piscataway, NJ, USA, 465–476.
[5] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. 2015. RFHOC: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 5 (2015), 1470–1483.
[6] André B Bondi. 2015. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice*. Pearson Education, London, United Kingdom.
[7] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. 2018. 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 (ESEC/FSE)*. ACM, New York, NY, USA, 703–713.
[8] Rajeev Chandramohan and Anthony J Calise. 2013. Output Feedback Adaptive Control in the Presence of Unmodeled Dynamics. In *AIAA Guidance, Navigation, and Control (GNC) Conference*. AIAA, Reston, VA, USA, 4517.
[9] Anthony Siming Chen, Jing Na, Guido Herrmann, Richard Burke, and Chris Brace. 2017. 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)*. IEEE, Piscataway, NJ, USA, 1074–1079.
[10] Haifeng Chen, Wenxuan Zhang, and Guofei Jiang. 2010. Experience transfer for the configuration tuning in large-scale computing systems. *IEEE Transactions on Knowledge and Data Engineering* 23, 3 (2010), 388–401.
[11] Yeounoh Chung, Peter J. Haas, Eli Upfal, and Tim Kraska. 2019. Unknown Examples & Machine Learning Model Generalization. arXiv:1808.08294 [cs.LG]
[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 143–154.
[13] Aniruddha Datta and Petros A Ioannou. 1994. Performance analysis and improvement in model reference adaptive control. *IEEE Trans. Automat. Control* 39, 12 (1994), 2370–2387.
[14] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 39–52.
[15] Yi Ding, Ahsan Pervaiz, Michael Carbin, and Henry Hoffmann. 2021. Generalizable and Interpretable Learning for Configuration Extrapolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 728–740.
[16] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 299–310.
[17] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015*

10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). ACM, New York, NY, USA, 13–24.

[18] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. 2017. Control strategies for self-adaptive software systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11, 4 (2017), 1–31.

[19] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. arXiv:2103.04112 [cs.NE]

[20] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 1–16.

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

[22] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*. CIDR, USA, 261–272.

[23] Henry Hoffmann. 2015. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM, New York, NY, USA, 198–214.

[24] Robert Vincent Hogg and Elliot A Tanis. 2009. *Probability and statistical inference*. Pearson Education, London, United Kingdom.

[25] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW)*. IEEE, Piscataway, NJ, USA, 41–51.

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

[27] Connor Imes, David HK Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Piscataway, NJ, USA, 75–86.

[28] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. 2019. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Piscataway, NJ, USA, 31–41.

[29] Rolf Isermann. 1982. Parameter adaptive control algorithms - A tutorial. *Automatica* 18, 5 (1982), 513–528.

[30] Glenn D Israel. 1992. Determining sample size.

[31] Prashant Kadam and Supriya Bhalerao. 2010. Sample size calculation. *International journal of Ayurveda research* 1, 1 (2010), 55.

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

[33] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, USA.

[34] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, New York, NY, USA, 700–711.

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

[36] Josua Krause, Adam Perer, and Enrico Bertini. 2016. Using visual analytics to interpret predictive machine learning models.

[37] Gerhard Kreisselmeier and Brian Anderson. 1986. Robust model reference adaptive control. *IEEE Trans. Automat. Control* 31, 2 (1986), 127–133.

[38] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, USA, 1–16.

[39] Yan Li, Kenneth Chang, Oceane Bel, Ethan L Miller, and Darrell DE Long. 2017. 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 (SC)*. ACM, New York, NY, USA, 1–14.

[40] Martina Maggio, Henry Hoffmann, Alessandro V Papadopoulos, Jacopo Panerati, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. 2012. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 7, 4 (2012), 1–32.

[41] Martina Maggio, Henry Hoffmann, Marco D Santambrogio, Anant Agarwal, and Alberto Leva. 2010. Controlling software applications via resource allocation within the heartbeats framework. In *49th IEEE Conference on Decision and Control (CDC)*. IEEE, Piscataway, NJ, USA, 3736–3741.

[42] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. 2017. Automated control of multiple software goals using multiple actuators. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering (ESEC/FSE)*. ACM, New York, NY, USA, 373–384.

[43] Biswadip Maity, Majid Shoushtari, Amir M Rahmani, and Nikil Dutt. 2019. Self-adaptive memory approximation: A formal control theory approach. *IEEE Embedded Systems Letters* 12, 2 (2019), 33–36.

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

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

[46] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 184–198.

[47] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 257–267.

[48] Marlies Noordzij, Giovanni Tripepi, Friedo W Dekker, Carmine Zoccali, Michael W Tanck, and Kitty J Jager. 2010. Sample size calculations: basic principles and common pitfalls. *Nephrology dialysis transplantation* 25, 5 (2010), 1388–1393.

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

[50] Patric Parks. 1966. Liapunov redesign of model reference adaptive control systems. *IEEE Trans. Automat. Control* 11, 3 (1966), 362–367.

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

[52] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Bhargava Gopireddy, and Josep Torrellas. 2017. Sthira: A formal approach to minimize voltage guardbands under variation in networks-on-chip for energy efficiency. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Piscataway, NJ, USA, 260–272.

[53] Raghavendra Pradyumna Pothukuchi, Joseph L Greathouse, Karthik Rao, Christopher Erb, Leonardo Piga, Petros G Voulgaris, and Josep Torrellas. 2019. Tangram: Integrated control of heterogeneous computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, New York, NY, USA, 384–398.

[54] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros G Voulgaris, and Josep Torrellas. 2020. Control Systems for Computing Systems: Making computers efficient with modular, coordinated, and robust control. *IEEE Control Systems Magazine* 40, 2 (2020), 30–55.

[55] Nirmit Prabhakar, Andrew Painter, Richard Prazenica, and Mark Balas. 2018. Trajectory-Driven Adaptive Control of Autonomous Unmanned Aerial Vehicles with Disturbance Accommodation. *Journal of Guidance, Control, and Dynamics* 41, 9 (2018), 1976–1989.

[56] Muhammad Husni Santriaji and Henry Hoffmann. 2016. Grape: Minimizing energy for gpu applications with performance requirements. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Piscataway, NJ, USA, 1–13.

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

[58] Huajie Shao, Shuochao Yao, Dachun Sun, Aston Zhang, Shengzhong Liu, Dongxin Liu, Jun Wang, and Tarek Abdelzaher. 2020. Controlvae: Controllable variational autoencoder. In *International Conference on Machine Learning (ICML)*. PMLR, USA, 8655–8664.

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

[60] Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2017. 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)*. IEEE, Piscataway, NJ, USA, 12–23.

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

[62] Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2019. SimCA* A Control-theoretic Approach to Handle Uncertainty in Self-adaptive Systems with Guarantees. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 13, 4 (2019), 1–34.

[63] Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni F Del Nero, Donatella Sciuto, and Marco D Santambrogio. 2013. ThermOS: System support for dynamic thermal management of chip multi-processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Piscataway, NJ, USA, 41–50.

[64] Torsten Söderström, Lennart Ljung, and Ivar Gustavsson. 1974. A comparative study of recursive identification methods.

[65] Yanxiang Tong, Yi Qin, Yanyan Jiang, Chang Xu, Chun Cao, and Xiaoxing Ma. 2021. 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 (ESEC/FSE)*. ACM, New York, NY, USA, 168–180.

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

[67] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. 2018. 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)*. ACM, New York, NY, USA, 154–168.

[68] Tao Ye and Shivkumar Kalyanaraman. 2003. 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 (SIGMETRICS)*. ACM, New York, NY, USA, 196–205.

[69] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 564–577.

[70] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload annotations: Bringing heterogeneous computing to existing libraries and workloads. In *2020 USENIX Annual Technical Conference (ATC)*. USENIX, Berkeley, CA, USA, 293–306.

[71] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. 2021. An Evolutionary Study of Configuration Design and Implementation in Cloud Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 188–200.

[72] Yanqi Zhou, Henry Hoffmann, and David Wentzlaff. 2016. CASH: Supporting IaaS Customers with a Sub-Core Configurable Architecture. *SIGARCH Comput. Archit. News* 44, 3 (jun 2016), 682–694. https://doi.org/10.1145/3007787.3001209

[73] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. BestConfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 338–350.

[74] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices* 45, 3 (2010), 129–142.