WASL: Harmonizing Uncoordinated Adaptive Modules in Multi-Tenant Cloud Systems

Ahsan Pervaiz*
The University of Chicago
USA

Anwesha Das
The University of Chicago
USA

Vedant Kodagi
The University of Chicago
USA

Muhammad Husni Santriaji
Universitas Gadjah Mada
Indonesia

Henry Hoffmann†
The University of Chicago
USA

Abstract

Modern cloud applications increasingly rely on adaptive control modules, such as dynamic resource tuning or system reconfiguration, to meet strict quality-of-service (QoS) objectives. However, when multiple independently developed adaptation modules are colocated on a shared infrastructure, their uncoordinated behavior causes interference leading to QoS violations. Existing approaches require centralized control or inter-module communication, violating modularity and limiting adoption in multi-tenant environments.

We present *WASL*, a modular runtime coordination technique, that enables colocated adaptive workloads to operate harmoniously without information sharing or control coupling. WASL estimates the deviation between expected and observed behavior using only local feedback and dynamically adjusts each module's adaptation rate to reduce interference. It acts as a lightweight plug-in with constant-time overhead and can be integrated into diverse adaptation strategies without requiring any changes to control logic.

We implement WASL across five latency-sensitive applications from the TailBench suite, incorporating three adaptation paradigms. Across single- and multi-application scenarios, WASL reduces tail latency by up to *84%* compared to uncoordinated adaptation and achieves performance comparable to centralized coordination approaches, while preserving modularity, avoiding information leakage, and aligning with resource isolation policies. WASL provides a general and practical solution for runtime coordination in adaptive cloud systems, enabling scalable deployment of independently managed services without compromising QoS.

CCS Concepts

• **General and reference → Performance**; **Estimation**; • **Computer systems organization → Cloud computing**; • **Computing methodologies →** *Modeling and simulation.*

Keywords

Clouds; Multi-Tenancy; Performance Interference; Cross-Layer Adaptation; Control Theory; Runtime Coordination

ACM Reference Format:
Ahsan Pervaiz, Anwesha Das, Vedant Kodagi, Muhammad Husni Santriaji, and Henry Hoffmann. 2026. WASL: Harmonizing Uncoordinated Adaptive Modules in Multi-Tenant Cloud Systems. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3777884.3797009

1 Introduction

Modern cloud applications increasingly rely on adaptive control mechanisms to meet strict QoS goals, especially for *latency-sensitive* (LS) applications [6, 46]. Such adaptation modules dynamically tune internal parameters—at the application- or system-level—to ensure low latency in response to dynamic workload changes [13, 15, 40]. However, when these adaptation modules are colocated on a shared cloud infrastructure, their uncoordinated behavior leads to performance degradation due to *destructive interference* [8]. This problem is especially acute in multi-tenant environments where independent stakeholders deploy applications with limited visibility into the behavior of other applications [6, 16, 42].

Limitations of prior adaptation methods: Existing solutions mitigate this interference using either (1) centralized, monolithic control frameworks [3, 13, 45] or (2) decentralized coordination mechanisms that share control variables across adaptation modules [14, 20, 39]. While effective in single-stakeholder settings, these methods introduce tight coupling between adaptation modules, undermining core properties of multi-tenant cloud systems, namely, *modularity* and *isolation* [24, 25]. Furthermore, practical challenges such as unknown workload compositions, dynamic scheduling, and proprietary adaptation mechanisms make such approaches brittle and difficult to deploy. Even small changes to one module may require coordinated updates across others, creating maintenance overhead and reducing flexibility. These issues make cross-module coordination impractical at scale in modern cloud deployments.

Key Insight: A major source of interference between colocated adaptive modules is not just resource contention, but the *mismatch in the rate* at which each module adapts to its environment. When multiple modules adjust their parameters rapidly and independently, they create a turbulent operating condition where no single module can accurately attribute cause and effect to its own actions. This results in oscillatory behavior and repeated QoS violations amidst applications, even if each module performs well in isolation.

[*] This work was done while the author was at the University of Chicago.
[†] ACM Corresponding Author.

ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797009

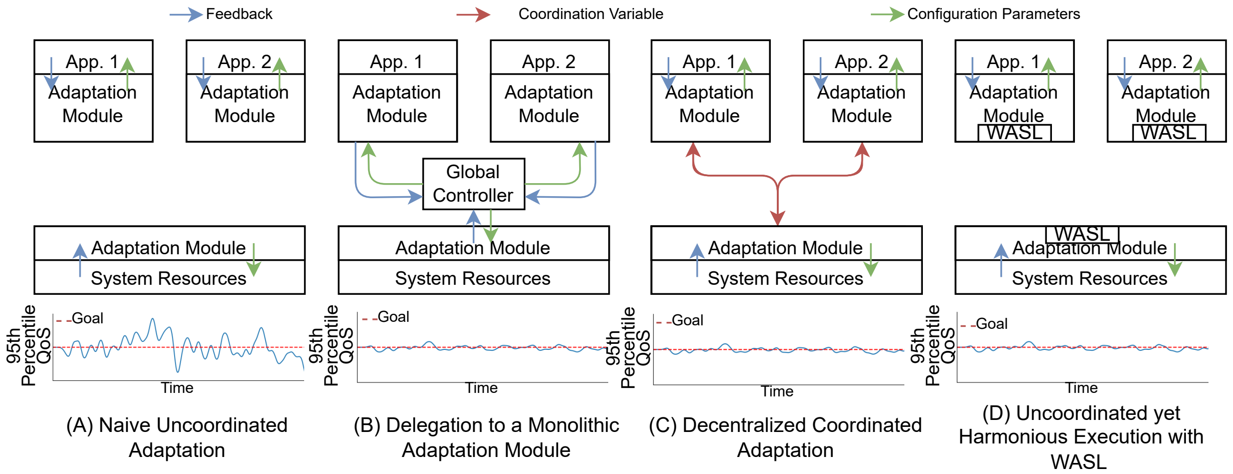

Feedback
Coordination Variable
Configuration Parameters
App. 1
App. 2
Adaptation Module
Global Controller
WASL
Adaptation Module
System Resources
95th Percentile QoS
Goal
Time
(A) Naive Uncoordinated Adaptation
(B) Delegation to a Monolithic Adaptation Module
(C) Decentralized Coordinated Adaptation
(D) Uncoordinated yet Harmonious Execution with WASL

Figure 1: Comparison of coordination strategies: (a) uncoordinated, (b) centralized control, (c) decentralized information exchange, and (d) WASL 's modular coordination, reduces oscillatory nature preserving deployability in multi-tenant clouds.

Our approach: WASL[1] addresses this challenge by introducing a lightweight runtime coordination technique that regulates the *rate of adaptation* preventing aggressive adjustments. Adaptive modules usually adjust based on an expected (i.e., a *target* or *desired* value) and measured (i.e., an observation in *current* runtime environment) behaviour. Instead of requiring modules to communicate or share control state, WASL monitors local feedback signals—such as tail latency—and quantifies the deviation between expected and measured behaviour. If this deviation exceeds a configurable threshold, WASL adjusts the adaptation rate via the *detour* metric, allowing the module to *adjust more conservatively*. This dynamic throttling ensures that each module has enough time to assess the effects of its own actions, thereby, reducing adaptive multi-tenant interference, and enabling more stable system-wide behavior. WASL is designed as an independent module requiring no coordination logic that can be invoked by an adaptive module at runtime via an API call.

We use three distinct adaptation modules based on control-theory and reinforcement-learning (RL) proposed by prior works [19, 33, 50] with five TailBench applications [26]. We make each application adaptive by using one of the three considered adaptation modules. We then colocate these adaptive applications with a system-level adaptive resource manager that allocates resources to these applications. We conduct experiments to examine the impact of interference caused by resource conflicts, and WASL's ability to overcome such interference. WASL imposes a negligible, constant $O(1)$ performance and memory overhead. We compare WASL to both naive uncoordinated and centralized adaptation methods. While our centralized coordination method is not an oracle, it represents the best practical baseline.

Contributions: Our work makes the following contributions:

(1) We show how colocated adaptive modules interfere with one another, degrading QoS despite being independently effective. This motivates the need for dynamic runtime coordination in multi-tenant clouds.
(2) We propose **WASL**, a modular coordination technique that dynamically regulates adaptation rates using only local feedback, with no shared control logic, information exchange, or code refactoring, improving QoS.
(3) We evaluate our runtime system with WASL across five latency-sensitive applications using three adaptation methods. WASL reduces tail latency by up to 84% over naive uncoordinated method, while matching the performance of centralized control, but without sacrificing modularity or deployability as in centralized coordination schemes.
(4) We open-source our runtime library at https://github.com/adaptsyslearn/AdaptationWithWASL to enable further studies on multi-module scalable adaptive coordination.

2 Background

We discuss the problem of multi-tenant interference (§ 2.1), followed by latency-sensitive applications and existing multi-module adaptation methods (§ 2.2) to motivate the need for WASL.

2.1 Interference and QoS violations

Modern cloud systems colocate LS applications to maximize resource utilization. These applications often operate under strict QoS constraints, such as tail latency requirements, and rely on adaptive mechanisms to tune application [1, 23, 42] or system [2, 37, 41] parameters in response to changing workloads or environmental conditions. For example, consider an application that adapts by adjusting thread count to its workload, running on a system that adjusts CPU frequency to utilization. If such independently developed adaptive modules are colocated (i.e., both adaptive applications and adaptive system managers), they may interfere with each other's control loops. This happens because each module adapts based on local feedback, unaware that its actions may influence or be influenced by the adaptation of others. Using the example above, if the application uses more threads to speedup at the same time the system allocates a higher frequency, both the application and

[1] *WASL* in *Urdu* means union of friends denoting *harmony* amongst different colocated cloud applications when they execute together

Table 1: Qualitative Features of Cross-Layer Multi-Module Adaptation

Studies	Decentralized?	Core Technique	No Sharing?	Content Shared	Set-up Overhead	Performance Overhead
SimCA [45]	×	Simplex Optimization, Control Theory	×	Sensor data, Speed	Redo/Update Controller Synthesis	O(n), n→number of controllers
SOSA [9]	×	ML, Control Theory	×	Control decisions, Sensor data	Re-write Supervisor (i.e., global controller)	O(n), n→number of low-level controllers
DDPC [43]	×	Control Theory	×	Allocated power, Number of requests/sec	Redo Controller Generation	O(n), n→number of app-level controllers
Yukta [39]	✓	Robust Control Theory	×	System/Appl. parameters, Meta-data	Re-write Multi-layer Controller	O(n), n→number of controllers
CoADAPT [10]	✓	Constraint Optimization	×	Constraints, Control decisions, Shared variables	Re-write adaptation planning strategy	$O(n)/O(n^2)$ latency, n→message/graph size
coAdapt [20]	✓	Control Theory	×	Appl. Configurations, Control decision	Re-write Runtime Controller	O(n), n→number of controllers
WASL	**✓**	**Rate Adaptation**	**✓**	N/A	**WASL (API) Call**	**O(1) memory and time**

system will see lower than expected latency. Consequently, both the application and system will immediately pull back (threads and clockspeed, respectively) for which both will experience lower than expected performance. This destructive interference often leads to instability, performance oscillations, and violation of QoS goals [9].

This interference is especially pronounced in multi-tenant environments, where different stakeholders deploy and manage applications and system resource managers independently. In such settings, cloud providers cannot assume knowledge of colocated adaptive applications (which workloads will be colocated at a specific time is not known ahead) or access their internal adaptation and control logic (data access privacy policies are in place for vendors and stakeholders [35]). Existing coordination mechanisms, such as centralized controllers or shared feedback signals [21, 39, 45], violate modularity and isolation principles, making them impractical for real-world deployment. As a result, even individually well-behaved adaptation modules can degrade system-wide performance when deployed together without any coordination [20].

2.2 LS Applications and Adaptation Modules

Data centers are moving from batch jobs to LS applications [6]. Also, LS workloads often have higher priority over best-effort applications, if any [34, 37, 55]. LS applications operate under strict tail latency constraints while optimizing secondary objectives like throughput, accuracy, or energy efficiency [37, 41, 43], e.g., web services, machine learning (ML) inference, and transactional databases. To manage these tradeoffs in dynamic runtime environments, both application and system developers incorporate adaptation modules that tune internal or system-level parameters at runtime, e.g., thread counts, DVFS states, or model precision levels [23, 43]. Recent research has proposed increasingly sophisticated adaptation strategies using control-theoretic techniques [14, 31, 39], ML-based methods (including RL) [8, 9, 47], and combinations of both [21, 33]. These methods adjust application or system behavior based on real-time feedback. As adaptation has become more popular for enabling systems to meet constraints, some studies integrate both application- and system-level adaptation to satisfy latency requirements while optimizing other goals [9, 20, 38].

However, these approaches break modularity and isolation by either relying on centralized controllers or decentralized coordination, where application(s) and system share information about what can be adapted either with a single adaptive module that jointly optimizes all adaptation [9], or via some priority or hierarchy amongst multiple adaptive modules [10, 39]. Fig. 1 shows these differences: colocating uncoordinated adaptation modules leads to unstable performance and poor tail latency (a), while both centralized (b) and decentralized coordinated adaptation (c) create brittle deployment dependencies; even a minor update to one module may require reciprocal changes to others. Additionally, in multi-tenant clouds, stakeholders are often unwilling or unable to share internal logic or performance data, due to (resource/tenant) isolation and other policy constraints (data, privacy etc.) [5, 6, 15]. Workload compositions are dynamic and colocation is unpredictable, making prior coordination methods difficult to apply in practice.

These limitations motivate the design of WASL as in Fig. 1d, a coordination approach that is agnostic to the internal logic of adaptation modules, requires no inter-module communication, and respects module boundaries. By focusing on regulating the *rate* of adaptation rather than the policy, WASL enables *harmonious* execution in dynamic, colocated environments without violating resource isolation or requiring any shared state.

3 Related Work

WASL relates to prior efforts in three broad areas: (1) coordination strategies for adaptive modules, (2) cloud-centric adaptation methods, and (3) system-level techniques for interference mitigation. We contrast them with WASL's modular, communication-free design.

Coordination Strategies for Adaptation: Several systems coordinate adaptation using centralized or decentralized control mechanisms. Centralized schemes rely on individual components to allow some central manager or monolithic module to control their adaptive components across system layers, as shown in Fig. 1(b) [9, 31, 43, 45, 51]. These methods can be optimal in single-stakeholder environments [31]. However, their centralized nature makes coordination difficult when adaptive modules are developed or deployed independently by different stakeholders. In other words, the operational reuse of these strategies involve updates to individual local components and/or the central control logic [44, 48].

Decentralized approaches aim to reduce this coupling by relying on shared signals, such as performance metrics or local adaptation information, as shown in Fig. 1(c) [10, 11, 14, 39, 50]. While these methods provide better modularity and isolation than a fully centralized approach, they still require inter-module communication or agreement on shared semantics, which is impractical in multi-tenant or multi-stakeholder systems. Besides, they can incur high set-up and/or performance overhead [10].

Tab. 1 lists some prior works that coordinate multiple adaptive modules in a centralized or decentralized manner involving information exchange comprising control decisions, sensor data, feedback signal, and application configurations (col.#5). Many studies build on principles of control theory, ML or optimization heuristics. Centralized methods perform near optimally but require global control [31, 45], while decentralized methods are more modular but require interactions that can break isolation. Our approach requires an adaptation module to only invoke WASL without major re-writes to the control logic. WASL differs from prior work by avoiding shared state or control entirely: it coordinates colocated modules *indirectly*, through local rate regulation based on behavior observed locally within each adaptation module, as in Fig. 1(d).

Cloud-centric Adaptation: Many studies [3, 5, 6, 8, 18, 22, 28, 37, 41] focus on reducing QoS violations in shared multi-tenant clouds. Techniques include QoS-aware resource scheduling, throttling and sparsification techniques to adjust parameters, workload classification with respect to heterogeneity and interference, using inter-service dynamics for vertical scaling, policy adaptation with performance learning, and kernel instrumentation. The core principles to adapt are at times adopted by application-level adaptation. However, most of these system-level solutions do not consider application-level adaptation, and thus does not require cross-layer multi-module coordination. These are complementary to our work.

System-level Adaptation: A separate body of work addresses interference in colocated applications using system-level scheduling, prediction, or isolation techniques. Cilantro [3], USHER [47], FIRM [41], Optum [29], and Imbres [30] focus on global scheduling or resource partitioning to manage performance under varying load. Others rely on profiling, QoS prediction, or ML to guide decisions about workload placement or configuration [5, 6, 8, 22, 28, 33, 37]. TopFull [36], Rajomon [52], and ScalaTap [7] employ decentralized rate-limiting involving communication amongst tenants, without considering individual application-level adaptation.

All of these techniques are complementary to WASL. While they aim to optimize global behavior at the infrastructure level, WASL operates alongside an adaptive module, requiring no support from the operating system (OS), scheduler, or other colocated services. WASL enables runtime coordination without shared control logic, model access, or communication between modules. This makes WASL uniquely suited to modern cloud environments where *modularity*, *deployability*, and *tenant isolation* are critical.

4 Approach

We first identify the qualitative features suitable for colocating multiple adaptive modules (§ 4.1), then demonstrate interference caused by cross-layer multi-module adaptation (§ 4.2), followed by a description of our overall runtime approach using WASL (§ 4.3).

4.1 Qualities for Colocated Adaptation

The following features if enabled for colocated multi-module adaptation can aid harmonious execution of multi-tenant applications:

(1) **No information sharing**: There should be no or minimal information exchange amongst the adaptive modules as per any existing isolation and data privacy policies [6, 22, 24].
(2) **No apriori requirements**: The core principle of multi-module adaptation should not rely on any additional model(s), or external control variables [31, 41]. Model-based adaptation can create dependencies outside an application, and may cease to be efficient with a *change* in colocated applications.
(3) **Application-agnostic**: The coordination method should not heavily rely on the intrinsics of application-level adaptation (e.g., *utility* of a learning method), so that it operates irrespective of what colocated applications are running [3, 41].
(4) **Low overhead**: With dynamically fluctuating workload conditions, the runtime overhead of the method should remain low leading to coordination efficiency [10, 39].
(5) **Flexible invocation**: Independent runtime invocation of the method over enmeshing the implementation intricately with an adaptation logic provides flexible reuse similar to API calls or tools proposed by past runtime systems [27, 32].

Features 1, 2 and 3 enable the ability to *seamlessly integrate* such a multi-module adaptive method to existing adaptation modules without making significant changes to the overall implementation. This makes an approach *general-purpose* in nature. Features 4 and 5 imply that the method's *ability to meet QoS goals outweighs its performance overhead*, making the cost-benefit trade-off worthwhile, encouraging the method's adoption in practice. We design our runtime library with the above factors in mind, achieving performance comparable to prior works, as indicated in Tab. 1.

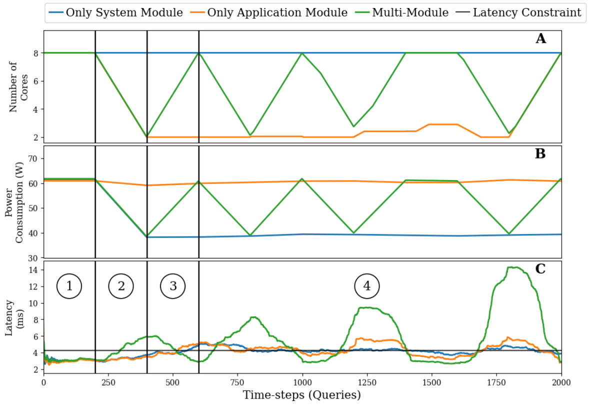


Figure 2: Colocated Adaptation

4.2 Motivational Example of Interference

To demonstrate destructive interference caused by multiple adaptation modules, we use *Xapian*, an indexing search engine application from TailBench [26], as a representative example. Our overall findings hold for other applications too (not shown due to space limits). We add adaptation modules to both the application and host system using control theoretic techniques from prior work [33]. Xapian's adaptation module adjusts control variables like *hyperthreading* and *core utilization* to meet a QoS goal, i.e., 95^{th} %-ile tail latency, while minimizing cost in terms of the *number of cores used*. System-level adaptation adjusts the *core* and *uncore frequency* with a QoS goal that ensures the *application meets its QoS goal and optimization objective* while minimizing *power consumption*. Our system-level QoS goal ensures that the hosted application(s) meet their QoS goal, similar to prior studies [6, 16]. When an application- or system-level adaptive module runs in isolation, they meet their respective QoS goals as per the provable guarantees of control theory.

Fig. 2 shows core utilization (*A*), i.e., number of CPU cores used, power consumption (*B*), and request latency (*C*) over some time-steps for different input queries of Xapian across three scenarios namely, a) only system-level adaptation is active, b) only application-level adaptation is active, and c) multiple cross-layer adaptation is active (i.e., *Multi-Module*). The latency variation shown in Fig. 2 is with respect to the QoS goal, i.e., our *latency constraint*.

Unlike application-only or system-only adaptation, multiple colocated adaptation causes interference resulting in increased constraint violations, e.g., [1750 - 2000] time-steps in Fig. 2. During time window ①, the measured latency meets the latency constraint. This enables the adaptive modules to reduce resource usage to minimize costs. Time window ② has QoS goal violation as the measured latency exceeds the constraint when both adaptation modules try to adapt simultaneously. To cope with this violation, the adaptive modules increase their resource usage that decreases latency, as seen in time window ③. However, resource conflict caused interference leads to goal violation again as the measured latency exceeds the latency constraint in time window ④. This *oscillatory* nature repeats itself as modules try to adapt with changing workloads. We find a 35% increase in measured tail latency for the *multi-module* scenario compared to scenarios that use a single adaptation module.

Observation 1: *Performance degradation can be higher with multiple colocated adaptation modules over a single-level or single adaptation module leading to increased QoS goal violations.*

We find that by reducing the CPU core usage (i.e., *A* in Fig. 2) to $\frac{1}{2}$ (from 8 to 4) instead of $\frac{1}{4}$ (8⟶2) of the previous value, with similar adjustments in other control variables, QoS goal violations could be prevented during window ②. Thus, our key intuition for coordinating adaptation is to minimize the deviation between *measured* and *expected* behaviour. This reduces the oscillatory nature leading to better performance in subsequent windows.

Observation 2: *The deviation between measured and expected application behaviour can be reduced by identifying a suitable rate of change by which necessary control variables should be adapted. Appropriate adjustments in adaptive modules can lead to stable system-wide performance lowering QoS goal violations.*

Obs. 2 implies that even if an expected behaviour is met in the *near future*, *drastic* changes in control variables can lead to oscillatory nature over time. In presence of multi-module adaptation, certain control variables have to be adjusted *moderately* to meet QoS goals leading to *long-term* stable performance. In practice, when multiple tenants adapt simultaneously, aggressive changes even in a single tenant can lead to a more chaotic operating environment. *Slowing down adjustments, even if can take longer time to stabilize, is key in reducing oscillatory behaviour for extended periods of time.*

The *rate of change* by itself is application-agnostic, independent of the specifics of an adaptation method. This gives rise to two questions: a) *what kind of control variables* should be dynamically adjusted by an adaptation module?, and b) *at what rate* should those variables change? Most principled adaptation methods use control variables that govern the *rate of change* in application behaviour, such as weights assigned to cost functions, call graphs, or reward functions [38, 41, 46, 51]. Some studies [23, 33] adjust this *rate* by updating the *pole* variable that decides how fast a system should react to changes in operating conditions. Other adaptive methods [50] adjust the Kalman gain variable that influences the *rate*. Thus, *adjusting these (often scalar) control variables related to the rate of change can help adapt application behaviour in a way that reduces its overall deviation from expected behaviour.* Every adaptation method can adjust its relevant control variable(s) as per its identified rate of change value.

We define the *detour* metric to indicate this *rate of change*. To determine *detour*, we need adaptation methods to measure runtime application behaviour, e.g., latency. Most adaptation methods based on feedback control or RL-based reward functions [33, 50] measure runtime behaviour as feedback, or for executing specific application configuration parameters. Thus, the measured behaviour is generally stored in adaptation modules that can be leveraged for multi-module coordination, requiring no extra communication, external to an application. Reducing the magnitude of *detour*, i.e., lowering the deviation between measured and expected behaviour, can lead to relatively better performance.

Observation 3: *Individual adaptation methods often have control variables that govern the rate of change in application behaviour. Besides, they have information about the measured and expected runtime behaviour. These together can help regulate deviation in application behaviour, eliminating the need for any external interaction.*

Obs. 3 enables tenant isolation preserving data privacy by not sharing data with other modules, suitable for multi-tenant clouds. Based on these observations, we design our runtime library using WASL that leverages the *detour* metric to coordinate adaptation. WASL can be used by any adaptation module that dynamically decides the *extent of adjustments* needed to stabilize performance in face of changing operating environment.

4.3 System Overview

Fig. 3 shows our overall framework. Our runtime system consists of colocated application(s) and system with their respective adaptation modules. The individual adaptation modules have the necessary control variables besides historically measured runtime behaviour, e.g., latency. WASL is a software technique to determine the *detour* metric that can be independently invoked from any adaptive module. An adaptive module uses the *detour* metric to adjust its control variable(s). Thus, applications do not require external interactions with one other or any central controller. WASL's presence in all

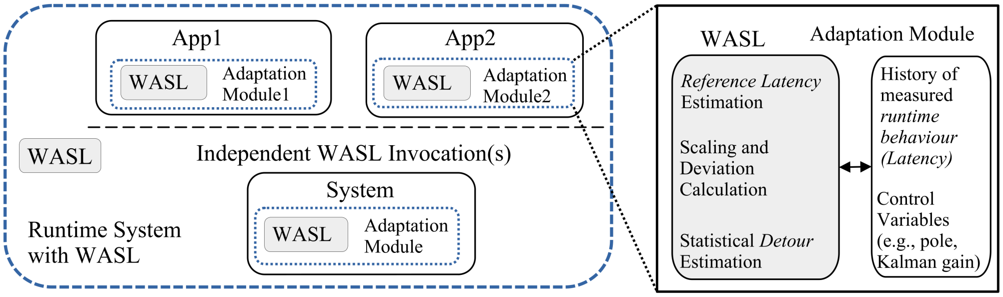

App1
WASL
Adaptation Module1
App2
WASL
Adaptation Module2
WASL
Independent WASL Invocation(s)
System
WASL
Adaptation Module
Runtime System with WASL
WASL
Adaptation Module
Reference Latency Estimation
Scaling and Deviation Calculation
Statistical Detour Estimation
History of measured runtime behaviour (Latency)
Control Variables (e.g., pole, Kalman gain)

Figure 3: High-level Runtime System Overview for Multi-Module Adaptation

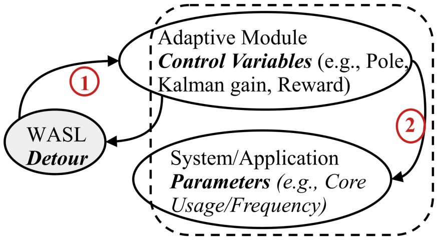

Adaptive Module
Control Variables (e.g., Pole, Kalman gain, Reward)
1
WASL
Detour
2
System/Application
Parameters (e.g., Core Usage/Frequency)

Figure 4: Variable and Parameter Updates

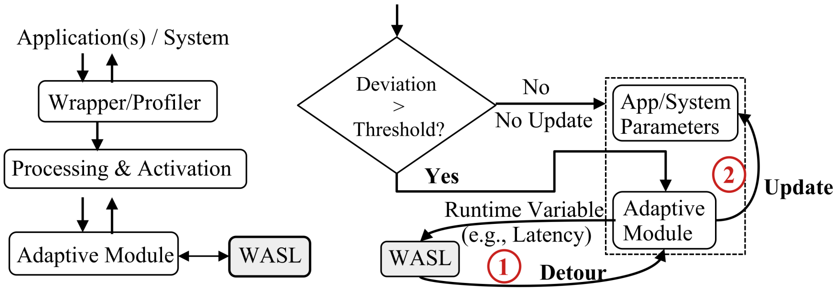


Figure 5: Detour-based Multi-Module Adaptation

running applications and system helps mitigate interference without any *explicit* coordination amongst multiple adaptation modules.

Our prototyped WASL-based runtime system including different adaptation methods can be used as a complete library to adapt in multi-module adaptation settings. This can facilitate further studies on cross-layer adaptation for various workloads and QoS goals.

Algorithm 1 Overall Runtime Coordination

```
Require: QoS Goals (goals), Optimization Objectives (Opt.)
Ensure: Tail Latency, Stability
 1: procedure Runtime Coordination(goals, Opt.)
 2:     for each Application do
 3:         [Measurements, Configs] ⟵ Profile Appl. (Available Resources)
 4:         ProcData ⟵ Activation (Measurements, Configs, goals, Opt.)
 5:         Local Adaptation ⟵ Adaptive Module (ProcData, CVar)

 6:     Adaptation Module Activation of Application(s) and System
 7:     WASL Invocation from all colocated Adaptation Modules
 8:     for each Adaptation Window do
 9:         detour ⟵ WASL (History, measured, expected, $\gamma$)    ▷ Algo. 2
10:         $\mathrm{CVar}_{new}$ ⟵ CVar · detour    ▷ Control Variable Update
11:         Parameter Update(s) ⟵ Adaptive Module (ProcData, $\mathrm{CVar}_{new}$)
12:         Runtime environment change
13:     Compute Tail latency, Stability
14: return Tail Latency, Stability
```

Algo. 1 shows our overall approach. Available system resources are extracted and applications are *profiled* for necessary measurements. A few such measured parameters are shown in Tab. 2. A set of distinct configurations are created for running an application based on the available resources, e.g., CPU core and uncore frequency, number of cores to be used etc. as shown in Tab. 3.

The collected raw data is processed for further aggregations as needed, e.g. computing *average* energy usage over a window and so on. The *activation* layer organizes the processed parameters, necessary control variables, specific QoS goals and optimization objectives to enable adaptation. This enables local adaptation at the system- or application-level. During system operation, adaptation modules of all running application(s) and system are activated. To enable cross-layer multi-module adaptation, every adaptation module invokes WASL to obtain its *detour* value.

Table 2: Measurements

Parameter	Description
Latency	Application Latency
Energy	Energy Usage
Core	CPU Core Usage

Table 3: Configurations

#	CoreFreq.	UncoreFreq.	#Cores
1	12	1200	8
2	16	2800	6
3	24	2000	2

Respective control variables of adaptive modules are updated by a factor of their *detour* values (#10). An adaptive module then uses the updated control variable to adjust system- or application-level parameter(s), as shown in Fig. 4. For e.g., an RL-based adaptation method [17] can update the *reward* variable based on its *detour* that then adjusts the IPC (instructions per cycle) parameter. *The translation of control variable to parameter update is intrinsic to an adaptive module* (i.e., ② in Fig. 4). We do not discuss details about the underlying technique of any adaptive module as that is not our core contribution. Those are adopted from prior studies [33, 50]. We analyze *tail latency* and *stability* to assess our multi-module adaptation approach. Fig. 5 further illustrates our modular design with interaction between WASL and an adaptive module.

Algo. 2 shows our *detour* estimation method called WASL, using measured and expected runtime behaviour in terms of latency. *Expected* refers to some known *target* or *desired* value, e.g., latency when the parameters were last updated, while *measured* is a *current* observation. We obtain a reference latency representative of a nominal configuration ($\text{ref}_{\text{latency}}$) based on latency values available in the adaptation module from prior adaptation steps, i.e., *history* values. Instead of using raw latency values, we compute ratios of measured and expected latency with respect to the $\text{ref}_{\text{latency}}$. These scaled values suggest the *slowdown* factor. We find *deviation* between the scaled latency values for statistical estimation. If the *estimated* deviation ($\text{estimated}_{\text{dev}}$) exceeds a *threshold* (γ), a new rate, i.e., *detour*, is computed using the ratio of γ and $\text{estimated}_{\text{dev}}$, else the previous rate is retained. Thus, detour always ranges between 0 and 1. *Detour*-based updated control variable is then used by an adaptation module for parameter adjustments that triggers a runtime environment change. With every threshold violation implying a change in the operating condition, *detour* is derived. For any adaptation window, *detour* varies across individual adaptation modules that helps mitigate multi-tenant interference.

Algorithm 2 WASL Multi-Module Coordination Method

```
Require: History, measured, expected, $\gamma$ (Threshold)
Ensure: detour                                              ▷ Determined rate
procedure WASL(History, measured, expected, $\gamma$)
    $\text{ref}_{\text{latency}} \leftarrow$ Obtain Reference Latency (History)
    $\text{measured}^{scale} \leftarrow$ measured / $\text{ref}_{\text{latency}}$
    $\text{expected}^{scale} \leftarrow$ expected / $\text{ref}_{\text{latency}}$
    deviation $\leftarrow$ ($\text{measured}^{scale}$ - $\text{expected}^{scale}$)   ▷ Scaled deviation
    $\text{estimated}_{\text{dev}} \leftarrow$ Statistical Estimator (deviation, History)
    if ($\text{estimated}_{\text{dev}} > \gamma$) then
        detour $\leftarrow \frac{\gamma}{estimated_{\text{dev}}}$                    ▷ Rate of Change
    else detour $\leftarrow$ Previously computed rate (i.e., no updates)
return detour
```

Choice of γ: The threshold depends on the performance dynamics of the runtime variable, irrespective of other colocated modules. In our experiments, γ ranges between 0.05 and 0.08. In general, a suitable γ can be empirically chosen based on the performance variation observed for a specific application without taking into account the shared runtime environment. The magnitude of measured and expected observations can help experimentally determine γ.

For WASL, it is necessary that individual adaptation modules are robust to noise in the absence of other adaptation modules. This is usually the case in practice [31, 50] as most adaptation methods use control theory or other optimization heuristics that are noise robust. In a multi-module adaptive setting, besides thresholding, γ helps to filter out noise, if any, due to colocated adaptation modules.

Reference Latency: Reference latency represents latency from a past runtime environment with different application configuration(s). The intuition is to use reference of a known highly performant configuration. If such latency data of an *optimal* configuration or a *stable* runtime environment is available to an adaptive module, that value can be directly used in Algo. 2, else, we can estimate $ref_{latency}$ with a Kalman filter using *directly available* historical latency values, as in prior work [2, 23, 50]. Eq. 1 shows standard Kalman filter based estimation using two previously available latency values, i.e., $\text{latency}^{\text{prev1}}$ and $\text{latency}^{\text{prev2}}$, where K is the Kalman gain variable, and μ is the mean. In a continuous running environment, at any time-step, the estimated mean (μ^{cur}) is based on the prior mean (μ^{prev1}) and the ratio of prior latency values (y^{cur}). This estimated latency can be used as $\text{ref}_{\text{latency}}$ for *detour* estimation. Eq. 1 takes constant time O(1) for an adaptation step.

$$\mu^{\text{cur}} \longrightarrow (1 - \text{K}) \cdot \mu^{\text{prev1}} + \text{K} \cdot \text{y}^{\text{cur}}, \text{where } \text{y}^{\text{cur}} \longrightarrow \frac{\text{latency}^{\text{prev1}}}{\text{latency}^{\text{prev2}}}$$
$$\text{ref}_{\text{latency}} \longrightarrow \mu^{\text{cur}} \tag{1}$$

Statistical Estimation: Some historical latency values may be needed for statistical estimation. To determine a suitable estimator, we analyze the following well known estimation methods:

1. Linear: Instantaneous rate of change via derivative [19].

$$\text{estimated}_{\text{dev}} \longrightarrow F(\text{deviation}, \text{deviation}_{\text{prev1}}, \text{deviation}_{\text{prev2}}) \tag{2}$$

2. EWMA (α): Exponential Weighted Moving Average, where α is the assigned weight for the historical average.

3. AR (p): Autoregressive model of order *p*.

4. ARMA (p, q): Autoregressive Moving Average model, where *p* and *q* are orders of *autoregressive* and *moving average* parts.

A statistical estimator uses the current and historical values to estimate future values. For *linear* estimation, we use two previously measured latency values to calculate prior deviations as per Algo. 2, i.e., $\text{deviation}_{\text{prev1}}$ and $\text{deviation}_{\text{prev2}}$ in Eq. 2. The prior deviations along with the current deviation (*deviation*) is used to obtain $estimated_{dev}$ as in Eq. 2. Similarly, other estimators use suitable history to obtain the estimated deviation.

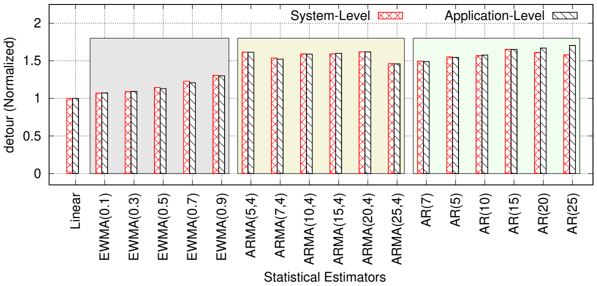


Figure 6: Detour Estimation (Linear, EWMA, ARMA, AR)

We show experiments with Xapian as a representative example as per our setting in §4.2, and derive *detour* using the aforementioned estimators. Similar findings hold for other applications too (not shown for brevity). Fig. 6 shows results normalized by the result from linear estimator for both system- and application-level executions. Overall, linear estimation has comparatively lower detour. Detour increases when larger weights are assigned to historical terms and parameters are updated. EWMA, AR, ARMA have larger detour as they retain a fraction of the estimated change from prior adaptation. The latter is usually addressed by an adaptation module, thus need not be re-considered for multi-module coordination. As α increases from 1 to 9 for EWMA in Fig. 6, detour increases indicating that *larger amount of history need not facilitate in reducing deviation.* Infact, with a dynamically changing environment, prior history becomes obsolete gradually for future estimation. In general, an estimator that does not rely *heavily* on past history is suitable for WASL. Thus, we use the *linear* estimator for our experiments.

As evident, WASL operates without sharing control state or message passing, does not rely on additional modeling or parameters that are not directly controllable, e.g., average threads/non-idle core, and harnesses variables already available to adaptive modules requiring no extra communication costs. WASL can be flexibly invoked in isolation without requiring module rewrites. Thus, WASL adheres to the desired qualities for colocated adaptation as per § 4.1.

Can WASL speedup adaptation? WASL is designed to work alongside an adaptation module that has its own pace of adjustments. As per Algo. 2, *detour* ∈ [0,1] indicates that WASL does not speedup adaptation beyond the maximum rate of adaptation inherent to an adaptive module.

Even if one adaptive module becomes aggressive (i.e., speeds up) in the absence of WASL, the other adaptive modules (with WASL) will begin to adjust more drastically leading to goal violations. As slowdown generally helps to meet QoS goals, WASL has not been developed to speedup adaptation.

Can WASL handle diverse runtime variables? WASL functions based on changes in the operating environment without having a global view of what runtime variables are used by individual adaptive modules. Besides, WASL uses the estimated deviation irrespective of the specific variable being monitored to derive a scalar ratio. Thus, WASL is effective even if adaptation modules use diverse runtime variables, e.g., latency, energy, or throughput.

5 Evaluation

We first describe the specific applications and adaptation modules used in our experiments (§ 5.1), followed by a discussion of our experimental results using the developed runtime library (§ 5.2).

5.1 Application, System, and Adaptive Module

Application: Tab. 4 lists the studied representative LS applications of different domains from TailBench [26] suite. TailBench [5, 22, 55] has been effective for studies on LS workloads and performance interference. Our application-level adaptation module uses parameters like hyperthreading, and allocated CPU cores to *minimize core usage* as the QoS goal, while satisfying a tail latency constraint. WASL's qualitative performance is similar for other LS workloads.

Table 4: Latency-Sensitive (LS) Applications

Application	Domain
Xapian	Online Search Engine
Moses	Statistical Machine Translation (SMT) System
Masstree	Scalable Key-Value Store
Silo	In-memory Transactional Database
DNN	OpenCV-based Handwriting Recognition

System: Our system-level adaptation uses core and uncore frequency parameters to meet a QoS goal while ensuring that all colocated applications meet their respective QoS goals. Our system-level QoS goal is *energy usage minimization.* As mentioned in Sec. 4.2, a system-level QoS goal often subsumes application-level goal(s) for which the system has relevant insights about the running application(s) [6], e.g., latency constraints and measured tail latencies.

Adaptation Modules: We use the following three adaptation methods from prior work:

1. Adaptive Control (AC) Module: A learning-based general adaptive controller to satisfy latency constraints [33].

2. PI Module: A discrete-time Proportional-Integral (PI) controller based on feedback loop, widely used for adaptation [19].

3. RL Module: A probabilistic RL design principle, using feedback based estimation and a slowdown factor for adjustments [50].

We consider these methods as their underlying techniques using classical control theory, ML, optimization etc., encompass major insights adopted by adaptation methods in general with wide and diverse usage [33, 43]. We refer to the reader to the papers for further details. In our experiments, an adaptation module, be it system-level or application-level, uses one of the above methods.

Baselines: We compare our approach with two multi-module adaptation baselines, similar to prior studies [10, 39, 41]:

(1) **Uncoordinated**: This is the *naive* case where there is no coordination (explicit or implicit) between application- and system-level adaptation modules.
(2) **Monolithic**: We adopt prior technique based on control theory that uses a model predictive controller for multiple QoS goals [31] as a centralized monolithic controller.

Monolithic is not an oracular baseline [3], however, our baselines are mostly optimal. Baseline comparison suggests WASL's ability to reduce QoS violations that would have been higher, otherwise.

Why no decentralized method(s)? We do not compare WASL with decentralized methods [10, 11] as they involve communication due to their reliance on shared state, as seen from col.#5 in Tab. 1, that conflicts with our communication-free design. They also violate deployment boundaries and have higher developmental costs (cols. #6-7 in Tab. 1) compared to our API call-based approach. *Even the best decentralized method has no better performance than our optimal monolithic method involving information sharing for which additional comparison yields no major insight.* Also, independent stakeholders often run their applications on the same shared cloud host. Sharing control variables across such *unrelated* applications can violate the tenant isolation principle making some of the decentralized methods unrealistic for multi-tenant clouds [20, 39].

5.2 Experimental Results

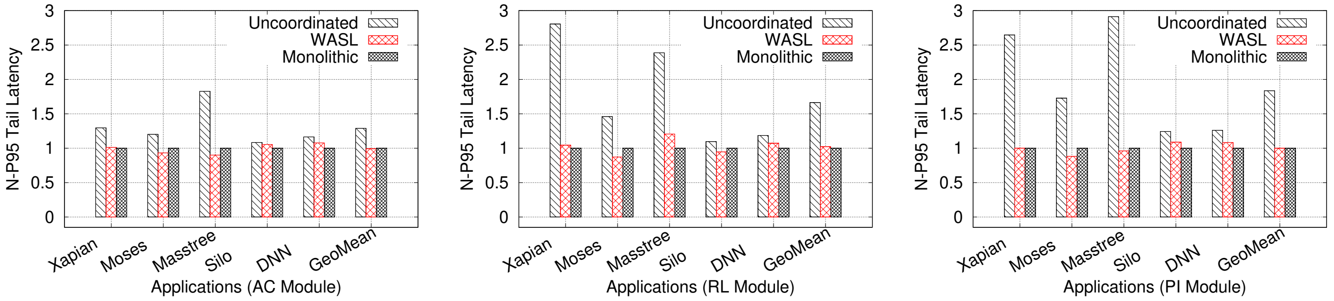
Set-up: We implement our runtime library in RUST. We conduct experiments on a compute node from a well used cloud platform running Ubuntu 18.04 on GNU/Linux 5.4 kernel having 192 GB RAM, 12 physical threads, 24 hyperthreads and Intel Xeon 6126 Gold processor with TurboBoost technology enabled.

Experiments: We evaluate WASL's performance for multi-module cross-layer adaptation in the following two contexts:

(1) A *single application* with different adaptation methods
(2) *Multiple applications* with different adaptation methods
(3) Impact on the adaptive multi-module *runtime environment*

We set tail latency constraints as per the realistic needs of LS applications running in cloud environments, i.e., 95th percentile latency, referred to as *P95 tail latency*, as in prior studies [22, 26, 37, 55]. We analyze WASL's performance with respect to the considered baselines. *For all experiments, we report the normalized P95 tail latency (in milliseconds), indicated by N-P95 tail latency in the plots.*

WASL's stability with different adaptation methods brings generality, while its ability to efficiently function in presence of multiple diverse adaptation methods simultaneously makes it robust. Thus, we experimentally evaluate both *generality* and *robustness* of WASL.



(a) AC Module

(b) RL Module

(c) PI Module

Figure 7: Single Application Performance with Different Adaptation Modules (Scaled w.r.t. Monolithic)

Single Application: How does WASL perform *when an application and system use the same adaptation module*? e.g., both Xapian and the system use the RL module. Fig. 7 shows results of the studied applications in terms of *N-P95 tail latency* scaled in relation to *Monolithic* performance. The main findings are as follows:

1. *Uncoordinated* approach has ≈1.3× to ≈1.8× higher tail latency compared to *Monolithic*, leading to increased QoS goal violations. WASL reduces the oscillatory nature, bringing performance closer to *Monolithic* by suitably adjusting *detour*.

2. Generally, WASL’s performance nearly matches *Monolithic* indicating that it is a *close to optimal* solution. WASL improves performance significantly for certain applications like, Xapian and Moses, by as much as 2.68× compared to *Uncoordinated*.

3. WASL’s performance can be slightly lower than that of *Monolithic* for rare cases, e.g., *Moses* with the RL or PI module. This is due to the stochastic nature of RL module, and the potential non-linearities that can lead to sub-optimal performance of the PI module. However, this is not a concern as with adequate iterations, *Monolithic* still tends to be optimal.

4. As seen from the geometric mean of Fig. 7c, WASL on average *reduces tail latency by 84%* compared to *Uncoordinated* that can benefit significantly in colocated cloud environments.

WASL helps to adapt in right proportion that aids to meet QoS goals thereby improving application performance, preserving cloud isolation principle(s), which is violated in Monolithic approaches.

Multiple Applications: We experiment with multiple colocated applications to examine WASL for two scenarios:

(1) **Symmetric**: Both applications and system use the same underlying adaptation method.
(2) **Asymmetric**: Applications and system use different adaptation methods that is more realistic in multi-tenant clouds.

It is practically infeasible to use *Monolithic* for multi-application scenario as what set of applications will be colocated at any time instance is not known ahead in dynamic clouds. Besides, colocated applications may change during an adaptation window. Thus, we compare WASL with *Uncoordinated* only, as in prior work [11, 20].

We consider colocated applications and report *N-P95 tail latency* scaled to WASL’s performance, similar to past studies [6, 24, 27, 33, 50]. *As our method adapts in relation to the changing operating environment, increasing the number of colocated applications does not make a fundamental difference to WASL’s performance.*

Fig. 8 shows results for the *symmetric* case. Our key takeaways are as follows:

(1) *Uncoordinated* degrades performance during multi-module adaptation similar to Fig. 7. WASL improves performance by 1.36× to 3.6× by suitably adjusting *detour*, and as much as ~30× for certain cases, helping reduce QoS goal violations.

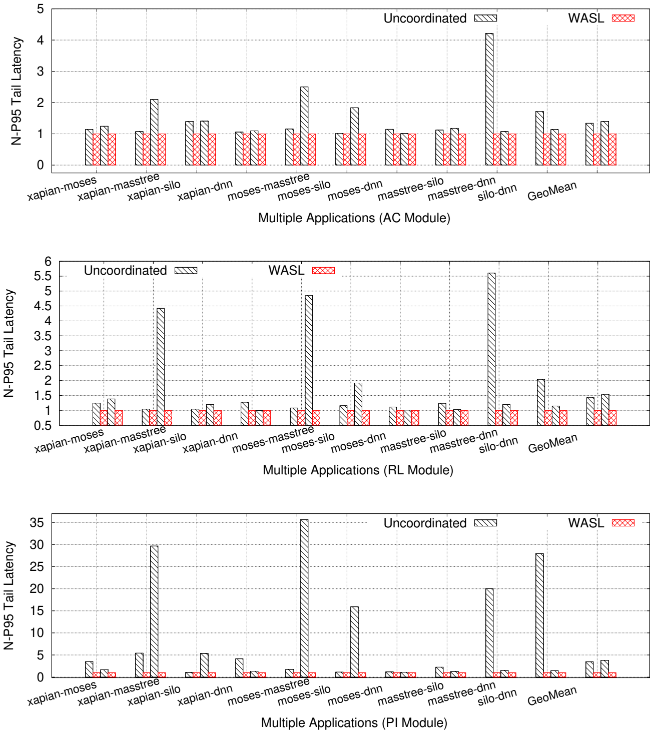


Figure 8: Multiple Application Performance with Symmetric Adaptation (Scaled w.r.t. WASL)

(2) Performance with PI module is relatively poorer than AC or RL module for *Uncoordinated* indicating PI-based adaptation’s subpar performance in certain settings. Learning-based method can be used in such cases.

(3) WASL’s performance benefits can be dramatic in some cases, e.g., over 29× for *Xapian-Masstree* pair with PI module. This helps to identify environmental settings when WASL can

be *more effective* over others. *When to invoke WASL* during runtime coordination can be decided based on such insights.

(4) Some colocated applications can exhibit symmetry in performance across adaptation modules, e.g., Masstree and DNN. In other words, their performance with respect to *Uncoordinated* is similar across diverse individual adaptation methods. For predictable performance in cloud environments, such characteristics can *guide runtime collocation decisions*.

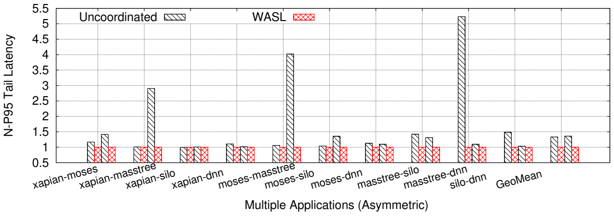


Figure 9: Multi-Application Performance with Asymmetric Adaptation (Scaled w.r.t. WASL)

Fig. 9 shows results for the *asymmetric* case similar to production data centers, where applications and system often use different adaptation methods. In our experiments, *the first and second applications use the AC and PI modules, respectively, while the system uses the RL module.* The results are normalized to WASL's performance. We have the following main findings:

(1) What applications are colocated impacts the extent of performance degradation possible. *Uncoordinated* degrades performance considerably for specific colocated scenarios, e.g., when Masstree colocates with Xapian or Moses. WASL improves performance by as much as ≈1.3×.

(2) Asymmetric adaptation scenarios can perform better than certain symmetric cases (e.g. using PI module only) indicating the power of asymmetry. WASL's ability to reduce destructive interference irrespective of what adaptation modules are used is beneficial in such scenarios.

Besides, reducing tail latency compared to *Uncoordinated*, WASL meets latency constraints for applications *in absolute terms that is an essential requirement in clouds.* WASL's performance in terms of the mean absolute percentage error (MAPE) is below 5%.

In summary, Figs. 8 and 9 show that some applications can colocate relatively more harmoniously than others, e.g., Moses and DNN. WASL-like cross-layer multi-module adaptation can also provide information to help decide which applications could be colocated together for more stability, or what adaptation modules are more compatible to meet QoS goals besides improving performance.

Detour generally varies for different adaptive modules with changing operating environment based on the estimated deviation.

Impact on Operating Environments: There is an assumption that operating conditions do not change frequently, and if they do, the duration of change is sufficient enough to recover from any performance disruptions. However, this assumption does not hold in colocated multi-module environments. Hence, we want our runtime system to cope with unprecedented changes. If WASL is enabled, we conform to this assumption once again as we successfully recover from QoS violations. Aside from improved application performance, WASL brings overall stability to the runtime environment.

To *quantify the environmental stability contributed by WASL*, we devise a metric based on *standard (std.) deviation* (as there is no direct measure of stability). We obtain the tail latency related to a highly performant application configuration (similar to $\text{ref}_{\text{latency}}$ in Algo. 2). We then measure the std. deviation across the estimated values of tail latency when an application is executed in the operating environment. The intuition is to measure variation between a *close to optimal* environment and a changing adaptive environment. A lower std. deviation indicates relatively better stability.

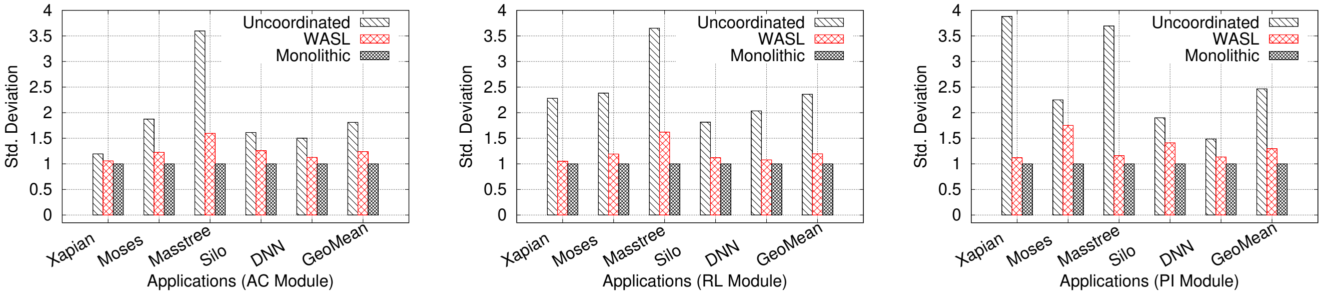
Fig. 10 shows results with a single application (i.e., one application and system using the same adaptation method) across the studied multi-module coordination methods. The reported std. deviation is normalized relative to *Monolithic. Uncoordinated* has much higher deviation leading to oscillatory behaviour in the system, irrespective of the adaptation method used. WASL reduces std. deviation by 46%, 95%, and 88%, respectively, across the three adaptive methods, leading to better stability. The ability to stabilize the runtime environment also facilitates the usage of controller correctness verifiers [49] that assume eventual stability in cluster environments.

Fig. 10 shows WASL's ability to reduce disturbances caused by interference in the environment considerably, enabling applications to execute harmoniously with the system, and meet their QoS goals.

Performance Overhead: As multi-module coordination is our core novelty, we discuss WASL's overhead excluding the individual adaptation methods and other implemented modules of our runtime library. WASL requires no set-up overhead. As per Algo. 2, using the measured and expected parameters has O(1) memory overhead. Obtaining reference latency and computing deviation takes O(1) time. Detour estimation can be done with O(1) complexity. Thus, the *overall performance overhead of WASL is constant in time and memory*, which can help when scaling workloads in clouds. Thus, WASL can be incorporated in an adaptive runtime system in a cost-effective manner. Contrary to WASL, adopting methods that involve centralized control, or significant message passing across adaptation modules incur higher set-up and/or performance overhead (Tab. 1).

Resource Efficiency: WASL strives to meet QoS goals via tail latency *without* improving efficiency in terms of minimizing energy or core usage. We observe that the energy usage using WASL is usually same or lower than the energy usage in the naive *Uncoordinated* case. However, there exists cases where resource usage can be higher than *Uncoordinated.* In those cases, the reduction in latency is often over 2× which is significant for LS applications, making this trade-off worthwhile. Besides, having lower latency can lead to lower resource usage. Some known adaptive approaches are not more efficient by being optimal [7, 17, 40]. Thus, WASL achieves scalability and stability *with resource efficiency no worse than some known adaptation methods, making it practically useful.*

Flexible Usability: One quality that distinguishes WASL from other cross-layer coordination methods is its flexibility in invocation. WASL is provided as an API call for runtime invocation. Such independent invocation is infeasible for methods using central controllers [39]. Decentralized methods have dependencies with



(a) Stability (AC Module) **(b) Stability (RL Module)** **(c) Stability (PI Module)**

Figure 10: Normalized Standard Deviation with a Single Application (Scaled w.r.t. Monolithic)

other modules for decision making that makes independent invocation difficult, e.g., using shared goals or conflicts that involve multiple adaptive modules [10, 11]. Our design enables existing adaptation modules to invoke WASL when needed during runtime without relying on any colocated modules. This provides *ease of reuse* and better programming interface for users that can benefit multi-tenant clouds. Further enhancements to transform this invocation to a generic tool using program abstraction techniques that may not require source code, or large language models (LLMs) [12] to automatically embed WASL in the right location of an existing adaptation module is subject to future work.

Discussion: WASL improves performance by as much as 84%, and at least 1.35×, at par with prior works that improve by 38% or 0.57× [3, 39] over baselines, but without information sharing. WASL's performance is as high as 3.6× in some scenarios, generally close to optimal, that can benefit LS workloads. WASL is not a microservice-specific solution [18, 36, 41], yet can be adapted for microservices with sub-millisecond scale latency constraints [25]. WASL brings stability to the runtime environment by using compatible rates of change, enabling applications to meet their constraints harmoniously without any explicit coordination between adaptation modules. For effective functioning of WASL, adaptive modules must be robust to the system and have measurable parameters related to the runtime environment. To the best of our knowledge, most principled adaptation methods possess the above qualities [9, 20], making WASL generally applicable.

As a use case, Micro-Armed Bandit (MAB) agent [17] is an RL-based adaptive module used for prefetching (data/instruction) in processors. When multiple MAB agents compete together, it can lead to memory contention due to aggressive adjustments. As each MAB agent is unaware of other's actions, uncoordinated adaptation causes interference resulting in suboptimal system-level performance, as shown in [4]. WASL can help coordinate multi-module adaptation in such scenarios without any communication to a central controller like μMAMA [4].

When is WASL not effective? WASL is not effective under two scenarios: a) the core technique of an adaptation method resembles *static* behaviour, e.g., comprises of fixed or periodic adaptation step size, and b) at least one of the colocated adaptive modules does not invoke WASL. Methods in a) are not influenced by the *extent of change* in adjustments, e.g., *binary* decisions like whether to scale uncore frequency or not [54], or static actions like setting core frequency to minimum [53]. These do not directly leverage runtime information that is needed for WASL to function. However, most dynamic adaptation methods use the value of runtime variables [37, 47], e.g., goodput or latency, for which WASL can be widely adopted. The level of flexibility available in invoking WASL depends on the intricate design specifics of the adaptation method.

Besides, even if one adaptive module does not invoke WASL, there can be incompatible adjustments in other colocated adaptive module(s) leading to increased overall QoS goal violations. Thus, *all colocated adaptive modules need to invoke WASL to mitigate global multi-module interference.*

6 Conclusion

We present a runtime library involving WASL, a software technique that enables harmonious colocation of application- and system-level adaptation modules. WASL's ability to meet QoS goals without explicit coordination with other adaptation modules is suitable for multi-tenant clouds with tenant isolation policies and data access restrictions in place. WASL achieves up to 84% reduction in tail latency and improves performance by over 29× that is comparable to prior approaches, but without any control state sharing. Our modular runtime coordination method can be seamlessly integrated into an adaptive runtime system, bringing stability to a dynamic environment, making it viable for adoption in practice.

Acknowledgments

We sincerely thank the anonymous reviewers for their helpful feedback that improved the paper and acknowledge the use of Chameleon Cloud testbed for compilation checks of the system. This project was supported by the National Science Foundation awards CCF-2119184, CNS-2313190, CCF-1822949, and CNS-1956180.

References

[1] Bilge Acun, Kavitha Chandrasekar, and Laxmikant V. Kalé. 2019. Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime System. In *Tenth International Green and Sustainable Computing Conference, IGSC.* IEEE, 1–8. https://doi.org/10.1109/IGSC48788.2019.8957174

[2] Cornel Barna, Marin Litoiu, Marios Fokaefs, Mark Shtern, and Joe Wigglesworth. 2018. Runtime Performance Management for Cloud Applications with Adaptive Controllers. In *ACM/SPEC International Conference on Performance Engineering, ICPE.* 176–183. https://doi.org/10.1145/3184407.3184438

[3] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *Symposium on Operating Systems Design and Implementation, OSDI.* USENIX Association, 623–643. https://www.usenix.org/conference/osdi23/presentation/bhardwaj

[4] Charles Block, Gerasimos Gerogiannis, and Josep Torrellas. 2025. Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching. In *IEEE/ACM International Symposium on Microarchitecture, MICRO*. 884–898.

[5] Quan Chen, Shuai Xue, Shang Zhao, Shanpei Chen, Yihao Wu, Yu Xu, Zhuo Song, Tao Ma, Yong Yang, and Minyi Guo. 2020. Alita: Comprehensive performance isolation through bias resource management for public clouds. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE/ACM, 32. https://doi.org/10.1109/SC41405.2020.00036

[6] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 107–120. https://doi.org/10.1145/3297858.3304005

[7] Zhongjie Chen, Yingchen Fan, Kun Qian, Qingkai Meng, Ran Shu, Xiaoyu Li, Yiran Zhang, Bo Wang, Wei Li, and Fengyuan Ren. 2025. ScalaTap: Scalable Outbound Rate Limiting in Public Cloud. In *IEEE Conference on Computer Communications, INFOCOM*. 1–10. https://doi.org/10.1109/INFOCOM55648.2025.11044509

[8] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 77–88. https://doi.org/10.1145/2451116.2451125

[9] Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil D. Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. In *International Symposium on Microarchitecture, MICRO*. ACM, 685–698. https://doi.org/10.1145/3352460.3358312

[10] Paul-Andrei Dragan, Andreas Metzger, and Klaus Pohl. 2023. Towards the decentralized coordination of multiple self-adaptive systems. In *International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS*. IEEE, 107–116. https://doi.org/10.1109/ACSOS58161.2023.00028

[11] Paul-Andrei Dragan, Andreas Metzger, and Klaus Pohl. 2025. Coordinated online reinforcement learning for self-adaptive systems using factored Q-Learning. In *International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS*. IEEE.

[12] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job?. In *USENIX Security Symposium, USENIX Security*. https://www.usenix.org/conference/usenixsecurity24/presentation/fang

[13] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *International Conference on Software Engineering, ICSE*. ACM, 299–310. https://doi.org/10.1145/2568225.2568272

[14] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated multi-objective control for self-adaptive software design. In *Foundations of Software Engineering, ESEC/FSE*. ACM, 13–24. https://doi.org/10.1145/2786805.2786833

[15] Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. 2025. TUNA: Tuning Unstable and Noisy Cloud Applications. In *European Conference on Computer Systems, EuroSys*. ACM, 954–973. https://doi.org/10.1145/3689031.3717480

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[17] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *IEEE/ACM International Symposium on Microarchitecture, MICRO*. 698–713. https://doi.org/10.1145/3613424.3623780

[18] Anyesha Ghosh, Neeraja J. Yadwadkar, and Mattan Erez. 2024. Fast and Efficient Scaling for Microservices with SurgeGuard. In *International Conference for High Performance Computing, Networking, Storage, and Analysis, SC*. IEEE.

[19] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. Wiley. https://doi.org/10.1002/047166880X

[20] Henry Hoffmann. 2014. CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems. In *Euromicro Conference on Real-Time Systems, ECRTS*. IEEE Computer Society, 223–232. https://doi.org/10.1109/ECRTS.2014.32

[21] Henry Hoffmann. 2015. JouleGuard: Energy guarantees for approximate applications. In *Symposium on Operating Systems Principles, SOSP*. ACM, 198–214. https://doi.org/10.1145/2815400.2815403

[22] Ziwei Huang, Mengyao Xie, Shibo Tang, Zihao Chang, Zhicheng Yao, Yungang Bao, and Sa Wang. 2024. INS: Identifying and Mitigating Performance Interference in Clouds via Interference-Sensitive Paths. In *ACM Symposium on Cloud Computing, SoCC*. ACM, 380–397. https://doi.org/10.1145/3698038.3698508

[23] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: A portable approach to minimizing energy under soft real-time constraints. In *Real-Time and Embedded Technology and Applications Symposium, RTAS*. IEEE Computer Society, 75–86. https://doi.org/10.1109/RTAS.2015.7108419

[24] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek R. Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX Annual Technical Conference, ATC*. 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[25] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 152–166. https://doi.org/10.1145/3445814.3446701

[26] Harshad Kasture and Daniel Sánchez. 2016. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In *International Symposium on Workload Characterization, IISWC*. IEEE Computer Society, 3–12. https://doi.org/10.1109/IISWC.2016.7581261

[27] Palden Lama, Shaoqi Wang, Xiaobo Zhou, and Dazhao Cheng. 2018. Performance Isolation of Data-Intensive Scale-out Applications in a Multi-tenant Cloud. In *International Parallel and Distributed Processing Symposium, IPDPS*. IEEE Computer Society, 85–94. https://doi.org/10.1109/IPDPS.2018.00019

[28] Jianshu Liu, Shungeng Zhang, and Qingyang Wang. 2023. μConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud. In *ACM Symposium on Cloud Computing, SoCC*. ACM, 427–442. https://doi.org/10.1145/3620678.3624980

[29] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2023. Understanding and Optimizing Workloads for Unified Resource Management in Large Cloud Platforms. In *European Conference on Computer Systems, EuroSys*. ACM, 416–432. https://doi.org/10.1145/3552326.3587437

[30] Shutian Luo, Jianxiong Liao, Chenyu Lin, Huanle Xu, Zhi Zhou, and Chengzhong Xu. 2025. Embracing Imbalance: Dynamic Load Shifting among Microservice Containers in Shared Clusters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 309–324. https://doi.org/10.1145/3676641.3716255

[31] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. 2017. Automated control of multiple software goals using multiple actuators. In *Foundations of Software Engineering, ESEC/FSE*. ACM, 373–384. https://doi.org/10.1145/3106237.3106247

[32] Nicolas Michael, Nitin Ramannavar, Yixiao Shen, Sheetal Patil, and Jan-Lung Sung. 2017. CloudPerf: A Performance Test Framework for Distributed and Dynamic Multi-Tenant Environments. In *International Conference on Performance Engineering, ICPE*. ACM, 189–200. https://doi.org/10.1145/3030207.3044530

[33] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. 2018. CALOREE: Learning Control for Predictable Latency and Low Energy. In *International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 184–198. https://doi.org/10.1145/3173162.3173184

[34] Pu Pang, Yaoxuan Li, Bo Liu, Quan Chen, Zhou Yu, Zhibin Yu, Deze Zeng, Jingwen Leng, Jieru Zhao, and Minyi Guo. 2023. PAC: Preference-Aware Co-location Scheduling on Heterogeneous NUMA Architectures To Improve Resource Utilization. In *International Conference on Supercomputing, ICS*. ACM, 75–86.

[35] Anjaly Parayil, Jue Zhang, Xiaoting Qin, Íñigo Goiri, Lexiang Huang, Timothy Zhu, and Chetan Bansal. 2025. Towards Workload-aware Cloud Efficiency: A Large-scale Empirical Study of Cloud Workload Characteristics. In *ACM/SPEC International Conference on Performance Engineering, ICPE*. 136–146. https://doi.org/10.1145/3676151.3722008

[36] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In *SIGCOMM*. ACM, 876–890. https://doi.org/10.1145/3651890.3672253

[37] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *International Symposium on High Performance Computer Architecture, HPCA*. IEEE, 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[38] Ahsan Pervaiz, Yao-Hsiang Yang, Adam Duracz, Ferenc A. Bartha, Ryuichi Sai, Connor Imes, Robert Cartwright, Krishna V. Palem, Shan Lu, and Henry Hoffmann. 2022. GOAL: Supporting General and Dynamic Adaptation in Computing Systems. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* 16–32. https://doi.org/10.1145/3563835.3567655

[39] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros G. Voulgaris, and Josep Torrellas. 2018. Yukta: Multilayer Resource Controllers to Maximize Efficiency. In *ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*. 505–518. https://doi.org/10.1109/ISCA.2018.00049

[40] Yi Qin, Yanxiang Tong, Yifei Xu, Chun Cao, and Xiaoxing Ma. 2024. Active Monitoring Mechanism for Control-Based Self-Adaptive Systems. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1841–1864. https://doi.org/10.1145/3660789

[41] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[42] Rohan Basu Roy, Vijay Gadepally, and Devesh Tiwari. 2025. DarwinGame: Playing Tournaments for Tuning Applications in Noisy Cloud Environments. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 264–279. https://doi.org/10.1145/3669940.3707259

[43] Mehmet Savasci, Ahmed Ali-Eldin, Johan Eker, Anders Robertsson, and Prashant J. Shenoy. 2023. DDPC: Automated Data-Driven Power-Performance Controller Design on-the-fly for Latency-sensitive Web Services. In *ACM Web Conference 2023, WWW*. 3067–3076. https://doi.org/10.1145/3543507.3583437

[44] Mehmet Savasci, Abel Souza, Li Wu, David E. Irwin, Ahmed Ali-Eldin, and Prashant J. Shenoy. 2024. SLO-Power: SLO and Power-aware Elastic Scaling for Web Services. In *IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid*. 136–147.

[45] Stepan Shevtsov and Danny Weyns. 2016. Keep it SIMPLEX: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*. 229–241. https://doi.org/10.1145/2950290.2950301

[46] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters. In *USENIX Annual Technical Conference, ATC*. 403–417. https://www.usenix.org/conference/atc23/presentation/shi

[47] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. USHER: Holistic Interference Avoidance for Resource Optimized ML Inference. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 947–964. https://www.usenix.org/conference/osdi24/presentation/shubha

[48] Bento R. Siqueira, Fabiano Cutigi Ferrari, and Rogério de Lemos. 2023. Design and Evaluation of Controllers based on Microservices. In *IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS*. 13–24.

[49] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[50] Chengcheng Wan, Muhammad Husni Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. 2020. ALERT: Accurate Learning for Energy and Timeliness. In *USENIX Annual Technical Conference, ATC*. 353–369. https://www.usenix.org/conference/atc20/presentation/wan

[51] Xiaorui Wang and Yefu Wang. 2011. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *IEEE Trans. Parallel Distributed Syst.* 22, 2 (2011), 245–259. https://doi.org/10.1109/TPDS.2010.91

[52] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. 2025. Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices. In *USENIX Symposium on Networked Systems Design and Implementation, NSDI*. 21–36. https://www.usenix.org/conference/nsdi25/presentation/xing

[53] Hongjian Zhang, Akira Nukada, and Qiucheng Liao. 2024. FCUFS: Core-Level Frequency Tuning for Energy Optimization on Intel Processors. In *IEEE International Conference on Cluster Computing, CLUSTER*. 214–225.

[54] Zhong Zheng, Seyfal Sultanov, Michael E. Papka, and Zhiling Lan. 2025. Minimizing Power Waste in Heterogenous Computing via Adaptive Uncore Scaling. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. ACM, 505–518.

[55] Liren Zhu, Liujia Li, Jianyu Wu, Yiming Yao, Zhan Shi, Jie Zhang, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Diyu Zhou. 2025. Criticality-Aware Instruction-Centric Bandwidth Partitioning for Data Center Applications. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*. IEEE, 442–457. https://doi.org/10.1109/HPCA61900.2025.00042