THE UNIVERSITY OF CHICAGO

FRAMEWORKS FOR GENERAL-PURPOSE ADAPTATION IN COMPUTING SYSTEMS

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

DEPARTMENT OF COMPUTER SCIENCE

BY
AHSAN PERVAIZ

CHICAGO, ILLINOIS

JUNE 2023

Forthcoming

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Forthcoming

ABSTRACT

Modern computer systems are becoming increasingly complex. Apart from functional correctness, they are expected to provide strict guarantees on their quality-of-service, expressed as goals over important quantifiable metrics such as latency, in the face of unpredictable changes in operating conditions and workloads during the execution of the system. Furthermore, these systems expose a plethora of tunable parameters that, combined with the unpredictable external conditions, impacts the quality-of-service of the system. It is a well established fact that no single configuration of the tunable parameters is optimal for all workloads and operating conditions, hence, systems are required to dynamically *adapt* their tunable parameters to cope with dynamic changes in workloads and operating conditions.

The systems community views adaptation as a crucial capability for systems to deliver reliable quantitative behavior in the presence of dynamic external factors. However, developing modules for robust adaptation, or *adaptation modules*, is difficult and requires specialized knowledge of machine learning and/or control theory. This increases the burden on developers who are expected to be experts in the aforementioned fields along with their specific system domain. To alleviate this burden, prior work suggests packaging adaptation modules as a library or in the runtime of a language. And during execution, systems can simply instantiate these modules using the goal that needs to be met and the parameters that can be adapted to meet those goals. Once instantiated, the frameworks then continually monitor the relevant behavior of the system and adapt the configurations on behalf of the system to ensure that the system continues to meet its goal.

However, A major limitation of prior frameworks is that they are implemented for a specific, narrow set of goals and knobs. Hence, they cannot be used for complex adaptive systems that must meet different goals using different sets of knobs for different deployments, or different execution stages of one deployment. For such scenarios developers are expected to embed different frameworks in their systems to support different goals. This, in turn, increases the size of the system code base and makes development and deployment more difficult.

In our research we explore the benefits of providing a single generalized adaptation framework that is agnostic of knobs and goals. We show the deployment and runtime benefits of using such a framework in a number of real-world systems including a networked video analytics pipeline. Our research shows that it is not only possible to implement a generalized adaptation framework but that using such a framework is more favorable from a development, deployment and performance perspective.

Another problem in the same domain is that of colocation. To maximize resource utilization and efficiency, system administrators, such as cloud providers, colocate multiple systems on the same hardware. However, when multiple adaptive systems are colocated they negatively interfere with each other leading to misbehavior which results in significant degradation of quality-of-service.

Prior works have recognized the problem of negative interference and have suggested many approaches to mitigate it. Such approaches require all colocatable systems to be enumerated beforehand. However, this restricts the system administrators' ability to colocate different systems. Furthermore, they impose severe restrictions on the systems that use them. For example, they impose restrictions on the mehanisms that the systems can use for adaptation. Similarly, they require information of internal details of the colocatable systems to be shared amongst each other. Often times prior work also requires all colocated systems to delegate their adaptation to a monolithic adaptation module. Such restrictions render approaches suggested by prior work suboptimal for use in the real-world where proprietary information can seldom be shared and development can seldom be coordinated between stakeholders.

In the latter part of this body of work we explore a general framework that can be used by all systems individually to allow harmonious execution of colocated adaptive systems without explicit coordination or information sharing. We show that such a system is beneficial because it provides a degree of freedom to stakeholders to develop their systems independently without worrying about the other systems that they may be colocated with their systems.

Hence, the contributions of this body of work are frameworks for: (1) adding complex, generalized adaptation in computing systems and (2) ensuring their successful and harmonious execution when they need to be colocated with each other.

CHAPTER 1

INTRODUCTION

1.1 Thesis Statement

This dissertation explores three problems that have become essential for modern computing systems: (1) supporting general-purpose adaptation, (2) supporting dynamic changes in all aspects of adaptation and (3) supporting harmonious colocation of multiple adaptive applications.

We address the first two problems together by proposing a first-of-its-kind framework for general-purpose and dynamic adaptation that allows systems to interact with all aspects of adaptation dynamically as first-class programming objects. We address the last problem by proposing an easy-to-use framework that can be incorporated by colocated adaptive system individually to ensure harmonious execution without explicit coordination or information sharing between systems. With the presented frameworks we hope to ease the development, deployment and colocation of general-purpose adaptive systems that can meet complex and changing adaptation requirements throughout their lifetimes.

1.2 Challenges

Adaptation has become a first-order concern for modern computing systems. From embedded devices to servers, all need to meet strict quality-of-service requirements in the face of changing workloads and operating conditions. While prior works recognize the need for adaptation and present several approaches for adding adaptation to computer systems, their suggested approaches have severe limitations that constrain the applications' ability to express complex adaptation requirements. These limitations also make it difficult for system administrators like cloud providers to colocate multiple adaptive systems on the same hardware to maximize resource utilization. At a high-level, the challenges that have been left unaddressed by prior work can be broken down into two smaller components.

1.2.1 Supporting Generalized Adaptation

Adaptive computing systems automatically tune their internal *knobs*, or configurable components, to meet quality-of-service (QoS) *goals*—specified in terms of constraints and objectives on *metrics* such as latency, energy, or accuracy—despite unpredictable external changes in workload and operating conditions. Constructing an *adaptation logic* that ensures goals are met is difficult, so prior work has proposed several *adaptation frameworks*, which package such logic up in a language [110, 154, 103] or library runtime [43, 29, 183, 50] making it easy to add adaptation to existing applications. Unfortunately, a major limitation of prior adaptation frameworks is that their internal adaptation logic is implemented for a specific, narrow set of goals and knobs. This narrow focus on specific goals and knobs makes it difficult to develop systems that use a different set of knobs and metrics for different deployments. Achieving this capability would require reimplementing adaptation using a different framework for different deployments. As the reader can imagine, this is an untenable approach to development of complex systems because maintaining several different versions of the same system is very difficult.

Furthermore, due to their narrow focus, their implementation makes it inherently difficult to interact with different aspects of adaptation during execution. This behavior is important for systems that need to dynamically modify the goal or knobs used for adaptation dynamically. Just as there is no single configuration that is optimal for all operating conditions, there is no single goal or set of knobs that is best for adaptation in all situations during the lifetime of a system. Hence, prior works impede the development of complex systems that need to modify different aspects of adaptation dynamically.

The limitations of prior works are discussed in detail in Section 3.1. These limitations result in the core challenge that using approaches suggested prior work *computing systems are unable to fulfil complex, general-purpose adaptation requirements that are expected to change during the lifetime of the system.*

1.2.2 Supporting Generalized Multi-Agent Adaptation

To maximize resource utilization, system administrators often colcoate multiple applications on the same underlying hardware. This is a common practice in the cloud where resources are notoriously underutilized [38, 186, 39]. However, when adaptive applications are colocated, their adaptation decisions change the operating environment of the colocated adaptive applications at a rate that exceeds the applications' ability to cope with these changes [99]. This leads to misbehavior which if often manifested as severe oscillations in important quality-of-service metrics, such as tail latency. Hence, being able to mitigate this misbehavior is crucial to enable successful colocation of adaptive applications to ensure high resource utilization in the cloud.

To this end, prior work has suggested three approaches to mitigate misbehavior caused by colocating adaptive applications. The first suggested approach is to coordinate adaptation decisions between application by passing signals between them [99, 100, 162]. The second approach works by assigning priority to applications and using heirarchical adaptation methods to meet goals of as many high priority applications as possible [78]. Finally, the third approach works by delegating adaptation for all applications to a monolithic adaptation module [181, 148].

A common feature of all prior approaches is that they require internal details of colocated applications to be shared with each other. However, such details are often proprietary and stakeholders are seldom willing to share them. Similarly, they impose strong restrictions on the mechanism of adaptation that can be used by the colocated applications. This severaly limits the cloud providers ability to colocate different applications. Furthermore, even minor changes to one of the applications could cause compatibility issues with other applications. This makes the development of applications to be hosted on cloud platforms extremely difficult.

The real-world implications of these limitations are discussed in detail in Section 4.1.3. But at a high-level these limitations mean that *cloud providers are practically not able to colocate adaptive latency-sensitive applications resulting in continued underutilization of cloud platforms.*

1.3 Contributions

This dissertation makes three concrete contributions:

- DDS, The **D**NN-**D**riven **S**treaming protocol: An iterative and adaptive protocol for machine learning based video-analytics applications. We present a concrete implementation of this protocol and evaluate the efficacy of its iterative design in conserving bandwidth while achieving high accuracy and the efficacy of its novel control system for dealing with variations in bandwidth availability.

- GOAL, The **G**oal-**O**riented **A**daptation **L**anguage: A novel, first-of-its-kind, adaptation framework that allows developers to add general-purpose adaptation to their applications and allows applications to dynamically modify all aspects of adaptation as first-class programming objects. We illustrate the usefulness and efficacy of GOAL by adding adaptation to several applications from prior works and comparing their performance with version using application-specific adaptation frameworks.

- WASL[1]: A novel framework that allows adaptive applications to dynamically alter their rate of adaptation to ensure that they meet their adaptation goals in the presence of colocated competing adaptive applications. We illustrate the efficacy of WASL by incorporating it in a number of benchmark applications and colocating them to simulate real-world scenarios and evaluating their ability to meet their respective adaptation goals.

The remainder of this section provides brief introductions to the aforementioned contributions.

1. WASL is a roman transliteration of an Urdu word which means a meeting of friends.

1.3.1 DDS

Internet video must balance between maximizing application-level quality and adapting to limited network resources. This perennial challenge has sparked decades of research and yielded various models of user-perceived quality of experience (QoE) and a plethora of QoE-optimizing streaming protocols. In the meantime, the proliferation of deep learning and video sensors has ushered in many distributed intelligent applications (e.g., urban traffic analytics and safety anomaly detection [28, 5, 21]). They also require streaming videos from cameras through bandwidth-constrained networks [23] to remote servers for deep neural nets (DNN)-based inference. We refer to it as *machine-centric video streaming*. Rather than maximizing human-perceived QoE, machine-centric video streaming maximizes for DNN *inference accuracy*. This has inspired recent efforts to compress or prune frames and pixels that may not affect the DNN output (e.g., [211, 57, 58, 56, 216, 122, 205, 74]).

A key design question in any video streaming system is *where to place the functionality of deciding which actions can optimize application quality?* Surprisingly, despite a wide variety of designs, all video streaming systems (both machine-centric and user-centric) take an essentially *source-driven* approach—it is the content source that decides how the video is best decoded/streamed. In traditional Internet videos (e.g., YouTube, Netflix), the video server determines the best encoding for a given bandwidth constraint, without explicit feedback from the viewer. Similarly, current machine-centric video streaming relies largely on the camera (source) to determine which frames and pixels to stream.

While the source-driven approach serves user-viewed videos well, we argue that it is necessarily suboptimal for analytics-oriented applications. The source-driven approach hinges on two premises: (1) the application-level quality can be estimated by the video source, and (2) it is hard to measure user experience directly in real time. While the assumptions largely hold in Internet video streaming, they need to be revisited in machine-centric video streaming.

First, it is inherently difficult for the source (camera) to estimate the inference accuracy of the server-side DNN by itself. Inference accuracy depends heavily on the compute-intensive feature extractors (tens of NN layers) in the server-side DNN. The disparity between most cameras and GPU servers in their compute capability means that any camera-side heuristics are unlikely to match the complexity of the server-side DNNs, and thus, the performance of the source-driven protocols is inherently limited. For instance, some works use inter-frame pixel changes [56] or cheap object detectors [216] to identify and send only the frames/regions that contain new objects, but they may consume more bandwidth than necessary (e.g., background changes causing pixel-level differences) and/or cause more false negatives (e.g., small objects could be missed).

Second, while incorporating real-time feedback from human users may be hard in traditional video streaming, DNN models can provide *rich* and *instantaneous* feedback! Running an object-detection DNN on an image returns not only detected bounding boxes, but also additional feedback for free, like the confidence score of these detections, intermediate features etc. Moreover, such feedback can be extracted on-demand by probing the DNN with extra images. Unfortunately, such abundant feedback information has not yet been systematically exploited by prior work.

In this paper, we explore an alternative *DNN-driven* approach to machine-centric video streaming, in which video compression and streaming are driven by the server-side DNN and how it reacts to real-time video content. Architecturally, DNN-driven video streaming follows an *iterative* workflow. For each video segment, the camera first sends it in low quality to the server for DNN inference; the server runs the DNN and derives some *feedback* about the most relevant regions to the DNN inference and sends this feedback to the camera; and the camera then uses the feedback to re-encode the relevant regions in higher resolution and sends them to the server for more accurate inference. (The workflow can have multiple iterations though this paper only considers two iterations). Essentially, by deriving feedback directly from the server-side DNN, it will send high-resolution content only in the minimal set of relevant regions necessary for high inference accuracy. Moreover, unlike prior work that requires camera-side vision processing or hardware support (e.g., [216, 56, 122]), we only need standard video codec on the camera side.

The challenge to realize this promise, however, is *how to derive useful feedback from running DNN on a low-quality video stream?* We present *DDS* (*D*NN-*D*riven *S*treaming), a concrete design of a DNN-driven protocol which utilizes the *region proposals* from DNN output on the low-quality video and sparingly uses high-quality, high-bandwidth encoding for the relatively small number of regions of interest. The insight is that the low-quality video may not suffice to get sufficient DNN inference accuracy, but it can produce surprisingly accurate region proposals. Region proposals are robust to low-quality video because they represent a binary classification task (i.e.,*whether* contains an object or not), which is easier than multi-class object detection (i.e.,*what* objects are in which region). Most modern computer-vision pipelines have a region proposal step to identify areas of interest in an image (e.g., where the queried object/identity may appear), followed by inference modules to compute the boxes or masks for the queried objects. Even when region proposal is not explicitly used, we show that it can be extracted from DNN's per-pixel/region confidence scores commonly found in modern computer vision models. Thus, leveraging region proposals does not incur additional burden on the system development.

There is parallel between DDS and the emerging recent trend in deep learning community of using *attention* mechanisms in DNN architectures (e.g., [199, 106]). Attention mechanisms learn to attend to the important pixels that will likely improve the DNN accuracy, thus sharing the same high-level insight with DDS that not every pixel affects the DNN output equally. These works are complementary to DDS – they improve DNN accuracy with additional DNN layers to focus *computation* on the important regions, while DDS proposes a suite of techniques to increase *bandwidth* efficiency (by sending only a few regions in high quality) for the same DNN accuracy.

We evaluate DDS and a range of recent proposals, including reusing existing video streaming ([211, 205]) and camera-side filtering heuristics ([56, 216]) on three vision tasks. Across 49 videos, we find DDS achieves same or higher accuracy while cutting bandwidth consumption by upto 59% or uses the same bandwidth consumption while increasing accuracy by 3-9% (Figure 1.1 shows an example.)

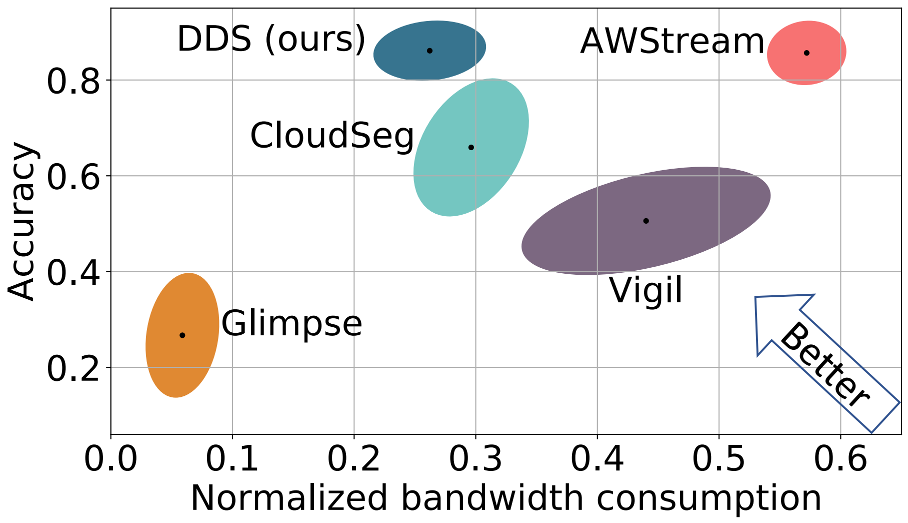


Figure 1.1: Performance of DDS and the baselines on a video dataset with object detection.

1.3.2 GOAL

Many software systems face the critical challenge of meeting quality-of-service *goals*, expressed as constraints and objectives on *metrics*; e.g., request latency, energy consumption, and result accuracy. As a further complication, these goals must be met despite unpredictable—yet inevitable—runtime variations in workload and operating environment. To provide predictable behavior in unpredictable deployments, it is crucial to build computer systems that *adapt* by adjusting their configurable components, or *knobs*, as they execute [133, 126].

Implementing adaptive systems requires an *adaptation logic* (AdaptLog) that can efficiently—at runtime—convert observed metrics into knob settings that meet the goals [208, 144, 185, 189, 92]. However, implementing a reliable and robust AdaptLog is difficult. For example, when the hardware for the Samsung Galaxy S9 was upgraded for the S9+, the achieved performance and energy efficiency was worse despite the better hardware [83]. The problem was tracked down to misconfigurations in the AdaptLog of the HMP scheduler [82]. In short, a heuristic-based AdaptLog that was appropriate for one hardware architecture, was inefficient on a closely related, but different architecture.

To ease development of adaptive computing systems, researchers have proposed several *adaptation frameworks* in the form of libraries or language runtimes [43, 173, 219, 123, 129, 49, 75, 112, 110, 87, 63, 62]. Using the framework's interface, developers provide an *adaptation specification* (AdaptSpec): a declaration of the system's goals and the knobs that can be configured to achieve it. The framework's internal AdaptLog then tunes the knobs in response to any runtime changes.

Although helpful, all existing frameworks focus on a narrow, predefined set of possible AdaptSpecs (i.e., one or two metrics and a small collection of knobs). For example, Eon [183] only supports accuracy and energy metrics using alternative method implementations as knobs. Similarly, PowerDial [103] only supports accuracy and throughput tradeoffs using application-level parameters as knobs.

This lack of generality arises because prior adaptation frameworks develop their AdaptLog using specialized *models* that relate specific metrics to specific knobs. Whether the AdaptLog is based on machine learning, control theory, or heuristics, the model is essential to predict how metrics will change with changes in knobs, which then guides the AdaptLog to set the knobs to ensure the goals are met. However, because the model relates specific knobs to specific metrics, the relevant knobs and metrics need to be enumerated before the model is constructed and that model must be reconstructed for use with a different knobs and/or metrics. This reliance on a narrowly defined model makes it difficult to implement a *general* adaptive system that can deploy with different goals in different environments.

The use of fixed models also prevents a system from *dynamically* changing AdaptSpecs during execution. We define the runtime alteration of an AdaptSpec as *meta-adaptation*. For many applications it is not enough to just adjust knob configurations; meta-adaptation is necessary as the goals themselves must be changed in response to external conditions [90, 165]. For example, consider a CCTV camera installed with a backup battery [177]. It must always meet a target frame rate to prevent data loss, but its other goals vary depending on power source. On line power the system must maximize quality, however, during a power outage, it must minimize energy to prolong battery life [3]. Running this system in either AdaptSpec for its entire execution is suboptimal, either wasting energy on battery or lowering quality on line power.

To support more general and dynamic adaptation, including meta-adaptation, this work presents GOAL: Goal-Oriented Adaptation Language. GOAL provides novel adaptation framework implemented in Swift [30]. Its key components are its (1) runtime AdaptLog and (2) its interface for writing AdaptSpecs.

Central to GOAL's design is its AdaptLog, which takes the form of a *virtualized, time-variant, adaptive* control system. Unlike prior approaches, GOAL's *virtual* control system is independent of any specific model relating metrics to knobs; instead, it is parameterized by a model which is passed in at runtime. Furthermore, GOAL's controller continually adjusts itself at runtime while also carefully exploiting structure of optimization problems so that it can control non-linear systems with a series of linear approximations.

GOAL's AdaptSpecs are written using a novel domain specific language (DSL), which is compiled just-in-time (JIT), separating the AdaptSpec declaration from system implementation. This separation allows different AdaptSpecs to be used with the same binary for deployments with different requirements or even for changing the requirements while the system is running. GOAL also provides a Library API so users can declare knobs and metrics and alter these values during execution. These features support complex adaptive behavior that would have been difficult and inefficient to implement with existing frameworks.

To demonstrate GOAL, we re-implement seven adaptive applications from the literature. Collectively, these case studies cover a wide range of metrics (throughput, latency, accuracy, power, cost, reliability and efficiency) and knobs (at both the application and system level, including two different hardware systems with distinct knobs). Our results show that GOAL's generalized approach meets goals just as well as prior work that synthesizes AdaptLogs specifically for each application's narrow goals and knobs [76]. To highlight GOAL's benefits, we then modify each application to perform meta-adaptation. We observe that due to GOAL's ability to support a wide range of AdaptSpecs and meta-adaptation, GOAL-based applications exhibit a 1.69× average improvement in corresponding metrics after meta-adaptation is performed, compared to prior approaches that cannot support meta-adaptation. Furthermore, we show that GOAL incurs negligible overhead and is robust to errors in profiling, changing workloads and operating conditions.

This work makes the following contributions:

- Motivates the benefits of support general purpose adaptation and meta-adaptation.
- Proposes a general adaptation logic and runtime that supports a wide range of knobs, metrics and goals.
- Proposes a programming framework and DSL for writing adaptation specifications.
- Implements GOAL and releases it as open source.[2]

1.3.3 WASL

In recent years cloud computing has become ubiquitous. And with the promise of resource efficiency and cost efficiency, the cloud has become the preferred platform for hosting latency-sensitive applications [37, 60, 9].

While ensuring functional correctness, modern latency-sensitive applications are also expected to fulfil quality-of-service (QoS) requirements defined as *goals* over important metrics, e.g. tail latency, in the face of changing request patterns and operating conditions [31, 61]. For example, a search-engine may be expected to maximize the accuracy of results while meeting a tail-latency constraint [159, 110]. Such applications also expose several configurable parameters, or *knobs*, that impact the applications' ability to their goals. Hence, such applications are often augmented with adaptation modules that dynamically tune their knobs to ensure that the ap-

2. GOAL source code available at: https://github.com/GOAL-Adaptation.

plication meets the QoS goal in the face of changing operating conditions and request patterns.

The cloud platform providers allocate resources to latency-sensitive applications carefully to balance the QoS of the applications with cloud efficiency. Since latency-sensitive applications experience diurnal usage patterns, statically assigning resources to latency-sensitive applications for peak usage can lead to severe underutilization of resources [38]. Hence, the cloud provider also use adaptation to dynamically adjusts the resources of applications to ensure that they meet their QoS requirements [128]. This allows the cloud provider to colocate applications with variable load on the same hardware to maximize utilization of hardware resources [62, 63]. However, in such situations colocated LS applications must compete for resources such as cores, cache and memory bandwidth. This resource contention causes *negative interference* between applications resulting in suboptimal performance. This leads to latency-sensitive applications missing tail-latency deadlines and, hence, violating their QoS requirements.

Prior work has proposed many different techniques to deal enable successful colocation of applications on the same hardware. A suggested approach is to simply not colocate latency-sensitive applications with other latency-sensitive applications [151, 64, 62, 63]. However, as one would expect this reduces the efficiency of the system. Furthermore, dynamically modifying resources for an LS application by checkpointing and migrating the colocated non-latency-sensitive applications requires time in the order of tens of seconds [40, 55, 141]. This is suboptimal for latency-sensitive workloads which experience changes in usage patterns over significantly shorter timescales [152, 114]. Another suggested approach is to not colocate applications that can interfere with each other [140, 196, 176, 63]. However, this approach severely limits the cloud-provider's ability to schedule different applications as this requires each new workload to be profiled with existing workloads to determine if there is a possibility of interference that will lead to misbehavior resulting in the applications being unable to meet their QoS goals. Finally, prior work also suggests methods to mitigate negative interference between applications. This is done by coordinating adaptation of all colocated applications and the underlying system by either delegating adaptation entirely to a monolithic adaptation module or by passing signals between different adaptation modules [162, 100, 99, 181]. Such approaches, while promising in terms of mitigating negative interference, are sub-optimal for use in the real-world. This is because they require colocated applications to share internal details with each other and the cloud provider. However, different stake-holders often use proprietary technologies in their applications and are seldom willing to share information about the internals of their applications. Furthermore, even if the information were to be shared, such techniques make strong assumptions about the techniques being used for adaptation and require wholesale changes to the colocated adaptation modules to enable co-ordination. This would also make the development of cloud applications significantly difficult because minor changes could require co-ordination between multiple independent stakeholders. This is also clearly impractical from the cloud provider's perspective because this would require the cloud provider to enumerate all combinations of colocatable applications.

Readers can easily imagine that in order for a proposed solution to be considered practical and usable in the real world, it is essential that it fulfills some important requirements. First, it must not require any kind of a model that could be invalidated if changes are made to the applications that are using it or its operating environment. Second, it should not be restricted to any particular method of adaptation. Third, it should impose a low (ideally negligible) overhead. Finally, it should integrate easily with the existing adaptation modules and not require extensive modifications. Unfortunately, to the best of our knowledge, no existing solution fulfils these crucial requirements.

In this project we propose WASL, a novel framework that, while meeting all of the aforementioned requirments, mitigates negative interference between applications to ensure that they are able to meet their respective QoS goals. Unlike prior approaches, WASL does not require any information sharing between colocated modules and does not make any assumptions about the techniques being used to perform adaptation. WASL is based on the fundamental understanding that *all* adaptation modules operate with the key assumption that changes in their operating environment are rare and last long enough for the adaptation module to react to those changes [96]. However, this assumption is broken when colocated adaptation modules are competing for resources. At a high-level, WASL reinstates this assumption for colocated modules dynamically to prevent misbehavior by modifying rate at which colocated adaptation modules modify each other's operating environment. To do so, WASL uses the difference between the expected behavior and the measured behavior of each application, which is already collected by all adaptation modules as feedback [96], to alter the rate of adaptation, a ubiquitous feature of adaptive systems, of applications individually. WASL is designed to be used by each adaptation module independently without needing to worry about the other modules with which it may be colocated.

We implement WASL as a framework that can be easily integrated with any existing adaptation modules. WASL is intended to be used by each adaptation module independently. Adaptation modules use WASL by calling a single function and provide it two floating-point arguments, i.e. the expected and measured behavior of the application, and returns a single floating-point value that represents the ratio by which the rate of adaptation of the module needs to be modified. Using WASL requires only minimal changes to the adaptation module as the information required for the arguments is either already available or can be easily calculate using the data that is already available to the adaptation modules.

To evaluate WASL we first add adaptation to applications from tailbench [125] using adaptation modules suggested by prior work. We colocate these applications along with a system level adaptation module that modifies system resources for each application dynamically. We show how in such a scenario colocated modules interfere with each other which results in misbehavior and loss of QoS. We then augment these adaptation modules with WASL and show how it is able to help eliminate misbehavior from these modules allowing them to return to the desired behavior. We show that using WASL, colocated control and learning based adaptive applications achieve upto 84% lower tail latency than naive adaptive execution without requiring any kind of information sharing or coordination. We show that WASL achieves this adding only a trivial performance and memory overhead to the adaptation modules.

This work makes the following contributions:

- Motivates the benefits of supporting harmonious execution of colocated adaptation modules without explicit coordination or adaptation delegation.
- Outlines key features that are required for a practical and feasible solution.
- Implements WASL that can be readily used by adaptation modules and releases it as open source[3].

1.4 Thesis Organization

This dissertation is organized to allow the reader to follow the progression of ideas between contributions. Chapter 2 provides details of the design, implementation and evaluation of DDS. Chapter 3 describes how during the implementation of DDS we realized major limitations of prior works in adaptation, presents details of adaptation approaches suggested by prior works and discusses their limitations and present GOAL a framework that overcomes said limitations. Appendix A provides details about the implementation and execution of the applications used for evaluating GOAL Finally, Chapter 4 describes the related problem to colocating adaptive applications to ensure high resource utilization, discusses the limitations of approaches suggested by prior works that make them impractical for use in the real-wolrd and presents a WASL, a novel framework that overcomes limitations of prior work to become a feasible solution for allowing colocation of adaptive applications

3. https://github.com/ahsanp/wasl.git

in real-world scenarios. Chapter 5 concludes the dissertation and provides a brief overview of the future work in this domain.

CHAPTER 2

DDS: DNN-DRIVEN VIDEO STREAMING FOR DEEP LEARNING INFERENCE

This chapter details the first contribution of this body of work: the *DNN-driven streaming* protocol. DNN-driven streaming is a generalized protocol that can be used for a variety of machine-learning centric analytics tasks.

DNN-driven streaming is a novel machine-learning centric streaming protocol that follows an *iterative* workflow. Using this iterative workflow each video segment is processed in two distinct phases. During the first phase, the server processes a low-quality input provided by the camera on the edge and produces *feedback* about regions that are most relevant to the analytics task and sends it to the camera. During the second phase, the camera uses the feedback to re-encode the relevant regions in a higher-quality and send them to the server for more accurate inference.

This project also presents DDS, a concrete implementation of the aforementioned protocol. DDS uses a novel control-based adaptation module that allows it to handle bandwidth variations significantly better than prior works that utilize hueristics to handle bandwidth variations. DDS can be used for a variety of vision-related analytics tasks. We evaluate DDS against a variety of prior works for video object detection, face detection and semantic segmentation. Our evaluation shows that DDS achieves same or higher accuracy while reducing bandwidth consumption by upto 59% or uses the same bandwidth consumption while increasing accuracy by upto 9%.

This chapter is organized as follows: Section 2.1 provides the motivation behind this work, Section 2.2 and Section 2.3 provides an overview and details of the proposed DNN-Driven Video Streaming protocol and the concrete implementation of the protocol, Section 2.5 provides an evaluation of DDS on three vision tasks. Finally, Section 2.6 provides an overview of the related work, Section 2.7 provides a brief discussion about the limitations of the protocol and Section 2.8 concludes our presentation of this part of this body of work.

Credit Assignment: This project was done in collaboration with Dr. Junchen Jiang's group and his students Kuntai Du and Xin Yuan. The presented discussion is taken directly from the publication resulting from the project [72].

For brevity and to maintain coherence between contributions of this project and the overall thesis we omit the discussion of vision tasks other than video object detection from this discussion of the project. The omitted discussion can be found in the publication of the project [72].

2.1 Motivation

We start with the background of video streaming for distributed video analytics, including its need, performance metrics and design space. We then use empirical measurements to elucidate the key limitations of prior solutions.

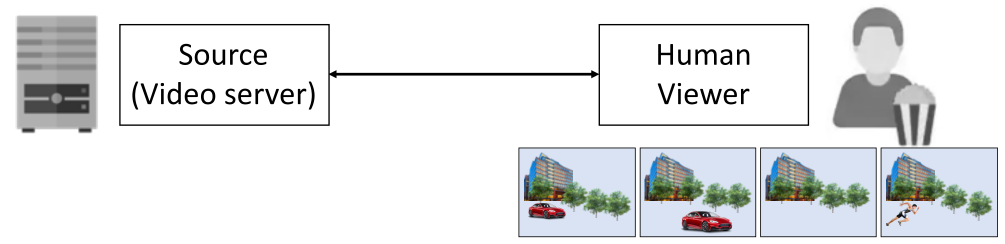
2.1.1 Video streaming for video analytics

Why video streaming? Two trends contribute to the wide spread of distributed video analytics and much recent effort (e.g., [211, 205]). On one hand, computer vision accuracy has been greatly improved by deep learning at the cost of increased compute demand. On the other hand, low prices of high-definition cameras have enabled cheaply scaling out the analytics of ever more cameras [20, 2, 6].

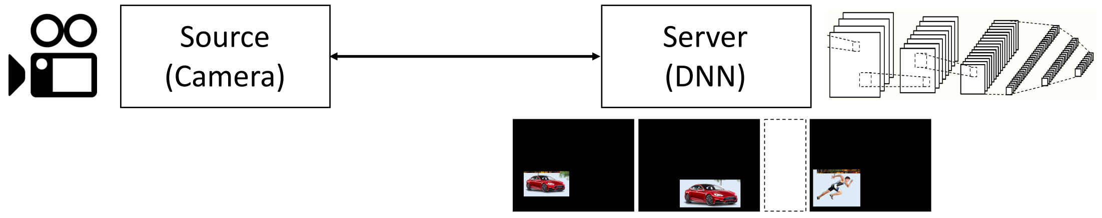
While one can always add accelerators [13, 1] to cameras,[1] an economical way of scaling out camera networks is to offload the compute-intensive inference (partially or completely) to remote GPU servers. For instance, buying 180 HD cameras and an NVIDIA Tesla T4 GPU (with a throughput of 5,700fps [14]) to process the feeds each at 30fps (total 5,400fps) costs $\$25\times180$(cameras)[17]+\$2000(GPU)[11]= \$6.5K; whereas buying 180 NVIDIA Jetson TX2 cameras (each runs ResNet50 barely at 30fps) costs about \$600[12]$\times180 =$ \$108K, 1-2 orders of magnitude more expensive. Therefore, camera network operators increasingly stream videos from network-connected cameras to servers for deep learning inference in traffic monitoring [28], surveillance in buildings, video analytics in retail stores [10], and inspection of warehouses or remote industrial sites with drone cameras [86].

Why saving bandwidth? We aim to reduce bandwidth usage of video analytics for three reasons. First, while many Internet videos are still destined to viewers, the video feeds processed by analytics engines have grown dramatically; by one analysis, the amount of data generated by new video surveillance cameras installed in 2015 is equal to all Netflix's current users steaming 1.2 hours of ultra-high HD content simultaneously [25]. Second, many cameras are deployed in the wild with only cellular network coverage [23], so setting up network connections and sending videos

1. Of course, some video feeds cannot be sent out due to privacy regulation and have to be processed locally, and they are beyond the scope of this work.



(a) Video streaming for human viewers



(b) Video streaming for computer-vision analytics

Figure 2.1: Contrasting video streaming for human viewers and machine-centric video streaming leads to unique bandwidth-saving opportunities.

upstream to remote servers can be expensive (more so than streaming videos from CDN servers to wifi-connected PCs). Finally, while purchasing GPU servers is not cheap, it is a one-time cost that is amortized over time (so cheaper in the long run); while the communication cost is not amortized thereby incurring substantial operational costs for many large-scale camera networks.

Performance metrics An ideal video streaming protocol for video analytics should balance among three metrics: accuracy, bandwidth usage, and freshness. Most real-world scenarios either detect/segment objects of interest, recognize faces or classify if the images contain a person or an object-of-interest. Here, we define them in multi-class object detection and semantics segmentation.

- *Accuracy:* Accuracy is measured by the standard metrics: *F1 score* for object detection (the harmonic mean of precision and recall for the detected objects' location and class labels) and IoU for semantic segmentation. We use the outputs of running the server-side DNN over the video in its highest quality as the reference and treat them as the ground truth when calculating accuracy. This is consistent with other work (e.g., [211, 117, 212]).

- *Bandwidth usage:* We measure the bandwidth usage by size of the sent video divided by its duration.

- *Response delay (freshness):* Finally, we define freshness by the average processing delay per object, i.e., the expected time between when an object first appears in the video feed and when its region is reported and correctly classi-

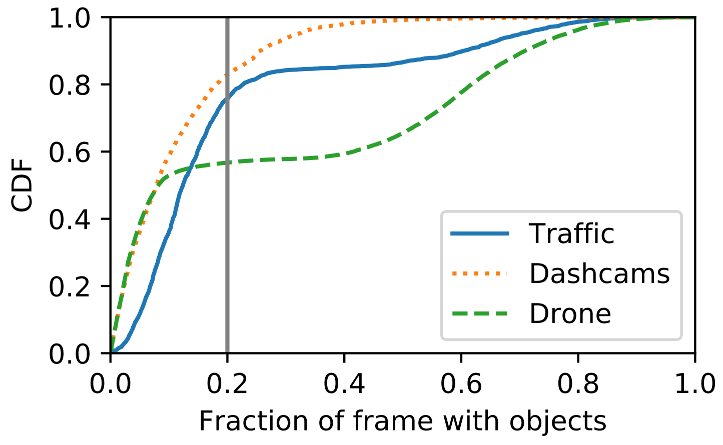


Figure 2.2: Bandwidth-saving opportunities: Over 50-80% of frames, the objects (cars or pedestrians) only account for less than 20% of the frame area, so most pixels do not contribute to the accuracy of video analytics.

fied, which includes both the time to transmit to the server and run inference. We report the average delay.

2.1.2 Potential room for improvement

Traditional video streaming maximizes human quality of experience (QoE)—a high video resolution and smooth playback (minimum stalls, frame drops or quality switches) [118, 70, 131]. For machine-centric video streaming, however, it is crucial that server-received video has sufficient pixels in regions that heavily affect the DNN's ability to identify/classify objects; however, the received video does not have to be high definition or smooth at all.

This contrast has profound implications that machine-centric streaming could achieve high "quality" (i.e., accuracy) with much less bandwidth. Each frame can

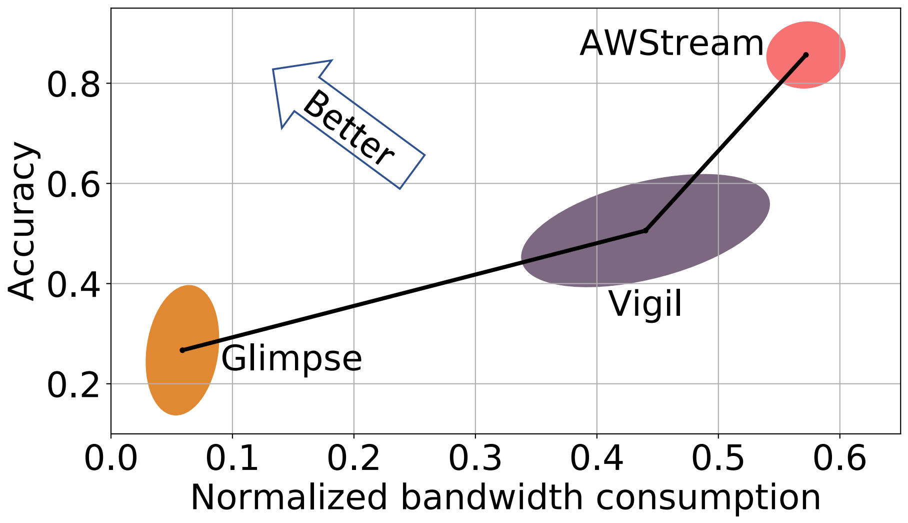


Figure 2.3: Current solutions exhibit a rigid bandwidth-accuracy tradeoff: any gain in accuracy comes at a cost in bandwidth.

be spatially encoded with *non-uniform quality levels*. In object detection, for instance, one may give low quality to or even blackout the areas other than the objects of interest (Figure 2.1(b))[2]. While rarely used in traditional video streaming, this scheme could significantly reduce bandwidth consumption and response delay, especially because objects of interest usually only occupy a fraction of the video size. Figure 2.2 shows that across three different scenarios (the datasets will be described in Section 2.5.1), in 50-80% of frames, the objects of interest (cars or pedestrians) only occupy less than 20% of the spatial area. We also observe similar uneven distributions of important pixels in face recognition and semantic segmentation. The question then is how to fully explore the potential room for improvement?

2. This may look like region-of-interest (ROI) encoding [153], but even ROI encoding does not completely remove the background either, and the ROIs are defined with respect to human perception.

Limitations of existing solutions Existing solutions for video streaming are essentially *source-driven*—the decisions of how the video should be compressed or pruned are made at the source with no real-time feedback from the server-side DNN that analyzes the video. However, they show unfavorable trade-off between bandwidth and accuracy (illustrated in Figure 2.3): any gain of accuracy comes at the cost of considerably more bandwidth usage. The fundamental problem is that any heuristic that fits camera's limited compute capacity is unlikely to precisely imitate a much more complex DNN, with no real-time feedback from it.

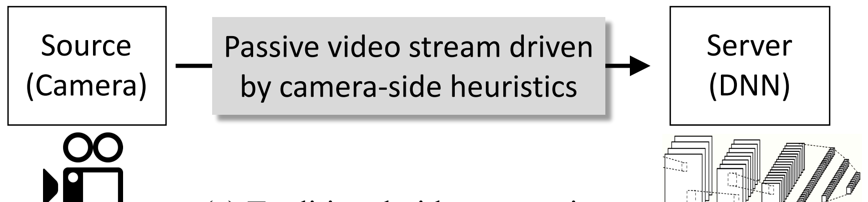
This problem manifests itself differently in two types of source-driven solutions. The first type is *uniform-quality streaming* that modifies the existing video streaming protocols and adapts the quality level to maximize inference accuracy under a bandwidth constraint. AWStream [211] uses DASH/H.264 and periodically re-profiles the inference accuracy under each quality level. CloudSeg [205] sends a video at low quality but upsizes the video by super resolution by the server. They have two limitations: First, they do not leverage the uneven distribution of important pixels; instead, the videos are encoded by traditional codecs with a spatially uniform quality. Second, while AWStream does get feedback from the server DNN, it is not based on real-time video content, so it cannot suggest actions such as increasing quality on a specific region in the current frame.

The second type is *camera-side heuristics* that identifies important pixels/regions/frames that contain information needed by the server-side analytics engine (e.g., queried objects) by running various local heuristics (e.g., checking significant inter-frame difference [56], a cheap vision model [57, 58, 216, 122]), or some DNN layers [74, 190]. These solutions essentially trade camera-side compute power to run inference for better accuracy/bandwidth trade-off [74]. For instance, Glimpse [56] and NoScope [122] rely on the inter-frame differences to signal which frames contain objects, but any false positives (e.g., pixel changes on the background) will cost unnecessary bandwidth usage and any false negatives will preclude the server from detecting important information.

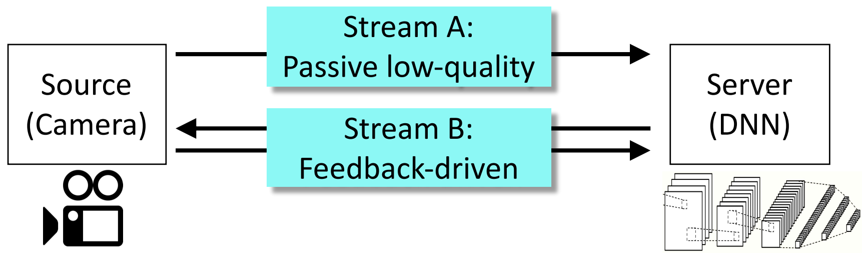
2.2 DNN-Driven Video Streaming

We explore an alternative approach, called *DNN-driven streaming* (DDS). In DDS, the compression/streaming adaptation is driven by feedback generated by the server-side DNN, rather than low complexity local heuristics, in order to capture what the analytics engine needs from the real-time video content. Figure 2.4 contrasts the workflow of DDS with the traditional source-driven approach. The key distinction is that source-driven streaming is "single-shot" (i.e., camera using simple heuristics to determine how the video should be streamed out), but DDS is *iterative* and logically contains two streams:

- **Stream A (passive, low quality):** The camera continuously sends the video in low quality to the server.
- **Stream B (feedback-driven):** The server DNN frequently (e.g., every handful of frames) generates and sends back feedback based on its output on the real-time low-quality video. Upon receiving the feedback from the server, the camera then re-encodes the recent history video accordingly and sends it to the



(a) Traditional video streaming



(b) Real-time DNN-driven streaming

Figure 2.4: Contrasting the new real-time DNN-driven streaming (iterative) with traditional video streaming in video analytics.

server for a second-round inference on these "zoomed-in" images.

An important design question then is *what should be the feedback?* An ideal feedback should meet two criteria. First, it needs to entail sufficient information to inform desirable follow-up actions in the second iteration. Second, it should generalize to a wide range of DNN-based vision applications.

The DDS feedback encodes *region proposals*—which regions/frames should be sent in the second iteration, if any, and at what quality level. We argue that region proposals meet the two principles of an ideal feedback. On one hand, if the proposed regions completely cover where all objects of interest appear, the server-side DNN should very likely detect all objects.[3] On the other hand, region proposals are

3. It may still miss some objects due to the known sensitivity of DNNs to pixel level alterna-

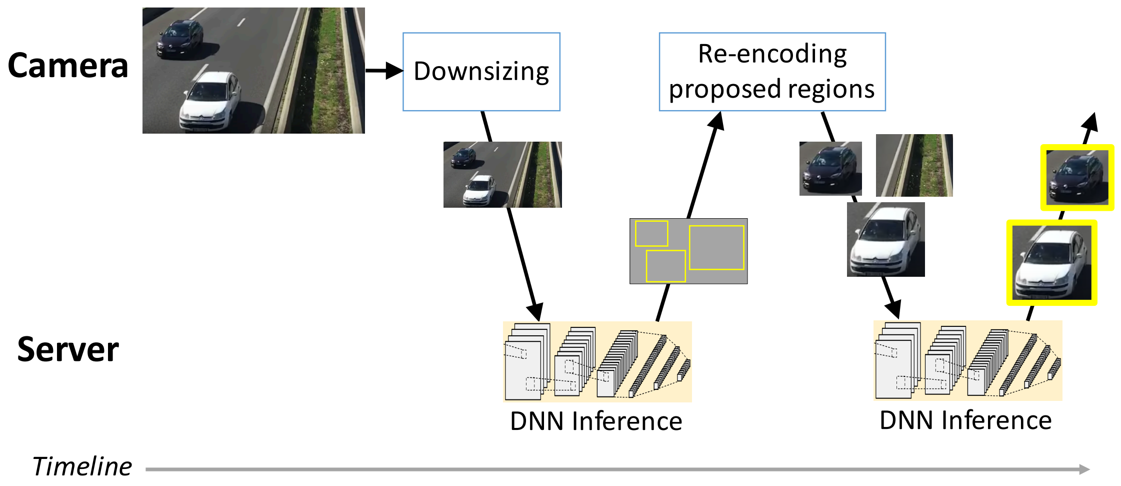


Figure 2.5: Illustration of the DDS workflow on a single frame. The two rounds of server-side inference use the same DNN.

(explicitly or implicitly) used in many modern object detection and face recognition pipelines [108]. See Section2.4.1 for details.

Figure 2.5 shows a simplified example of DDS in action. On the low quality frame, the server DNN proposes three regions and sends them back as feedback. The camera then re-encodes these regions in higher quality, and sends them to the server, which then runs DNN again and returns more accurate result.[4] DDS can adaptively choose the second-iteration quality level to cope with bandwidth variance.

We should stress that DDS makes minimum assumption about the codec and the server-side DNN. Thus, it can work with off-the-shelf video codecs (e.g., MPEG/H.264) and DNN models (e.g., FasterRCNN-ResNet101). DDS does not require any camera-

tions [88], but such behavior is generally negligible.

4. In theory, the iterative process can have more than two iterations, though the response delay will grow with more iterations and we found the performance benefit diminishes after two iterations.

side computing except basic video encoding.

Why DDS might work The rationale is that region proposal is essentially binary classification (i.e.,*whether* some object is a region), which is easier than object detection or face recognition (i.e.,*what* object/identify is in a region). By accurately identifying a few regions of interest based on just the low-quality video, DDS only needs to send selective regions in high quality.

At first glance, the idea of server-driven region proposal seems similar to Vigil [216], which also identifies and sends only regions likely with objects to the server, but as we will show in Section2.5.2, even if Vigil uses a model (MobileNet-SSD) that runs only 3× faster than the server-side DNN [18], it still misses about 40% more objects of interest than DDS and sends over 50% more data. The reasons are two-fold. First, Vigil's local model uses inherently simpler feature extractor, and thus tend to miss some (especially small) objects even on high quality images. Second, the local model does not change with the server-side vision task, so it may propose more regions than necessary (if the server only looks for a specific type of objects). Finally, unlike DDS, Vigil and Glimpse require extra computing power on the camera side.

2.3 DDS protocol design

2.3.1 Formalizing DDS control logic

We start with a general framework of DDS's control logic. A video stream is chopped to *segments*[5] (each contains a set of n consecutive frames), and DDS operates on a segment-by-segment basis. In each frame, one can "spatially sample" one or more regions (a region is a bounding box) and together they form a *region set*. We use S to denote a segment, R a region set, and $V_{S,R,Q}$ the video generated from a region set R of segment S encoded in quality level Q, which specifies the resolution and qp (quantization parameters). Two special types of regions are particularly useful. First, a region covering the full frame can form a region set, labeled R_{full}, so $V_{S,R_{\text{full}},Q}$ denotes the entire segment encoded in quality level Q (i.e., without any spatial sampling). Second, a frame can have an empty region set which effectively means it is skipped.

DDS's iterative protocol (Section2.2) can now be formalized as follows. For each segment S, the camera first encodes it at low quality level Q_{low} and sends the resulting video $V_{S,R_{\text{full}},Q_{\text{low}}}$ to the server (Stream A). Then, the server-side DNN runs inference on the video and returns the *high-confidence inference output* D_{dnn} (the part of DNN output that already has high confidence with low-quality video), as well as the *proposed region set* R_{dnn} (regions deemed as likely to contain inference). The high-confidence inference output and the set of proposed regions are then combined to generate the feedback region set, denoted by R_{feedback}. The feedback region set

5. The concept of segments is equivalent to video chunks used in the DASH protocols [7] and AWStream [211].

will then be sent back as feedback to the camera, which then encodes these regions at higher quality level Q_{high} and sends the resulting video $V_{S,R_{\text{feedback}},Q_{\text{high}}}$ to the server for a second iteration inference (Stream B).

2.3.2 Extracting feedback from DNN

We first introduce how the proposed region set R_{dnn} and high-confidence inference output D_{dnn} are extracted from DNN output for various vision tasks (e.g., object detection, face recognition, and semantic segmentation)[6]. It is important that their definitions are agnostic to the exact vision tasks and the DNNs. This means the generation of feedback regions is independent to the particular vision task and DNN.

High-confidence inference output D_{dnn}, in object detection (face recognition or semantic segmentation), is a list of DNN-returned elements (labeled bounding boxes in object detection, faces in face recognition, or pixels in semantic segmentation), each associated with its coordinates and confidence score. We use the elements that have confidence score over a threshold, which suggests they are ready to be reported.

Proposed region set R_{dnn} includes the regions (we use rectangle-shape boxes for encoding efficiency) that have high *objectness* scores—likelihood to be part of object of *any* class of interest. The objectness scores of proposed regions can be

6. Face recognition and semantic segmentation are discussed in the protocol design to illustrate the generality of the protocol. But evaluation of vision tasks other than object detection is omitted in this document. Please refer to the DDS paper for more details on evalution of other tasks.

obtained in two ways. First, they can be directly obtained from the RPN results, if the inference pipeline is based on RPN (e.g., FastRCNN, FasterRCNN, MTCNN, or MaskRCNN [94]) which explicitly proposes regions and evaluates their objectness scores. Second, they can be obtained from the DNN's inference output. For instance, Yolo [167] assigns each potential region a list of per-class confidence scores, and the scores of various classes can be summed up to get the objectness score per region[7]. Another example is that pixel-level inference output can be similarly used to derive objectness scores of regions. In semantic segmentation output, each pixel will be assigned with a per-class confidence score. For each region (DDS uses 64x64 blocks), we can calculate its objectness score by summing up the confidence scores across all pixels and all classes.

Selecting feedback regions As described in Algorithm 1, DDS picks from R_{dnn} the feedback regions that meet two criteria: (a) each feedback region (in terms of the fraction of the whole frame) is less than a threshold γ (by default, 4%[8]); (b) each feedback region does not overlap significantly with any high-confident inference output D_{dnn},[9]; and (c) the objectness score of a region is over a threshold θ; The rationale behind them is as follow. (a) filters out the regions that the DNN is already

7. We acknowledge that the sum of confidence scores across all classes does not directly translate to the likelihood of a region being in any of the classes. That said, we have empirically found it a good indicator of objectness.

8. Although 4% of the whole frame size looks low, it actual is quite high (roughly the size of a truck in a distance of 70ft from the camera).

9. In object detection, a proposed region does not overlap with a detected region if their IoU (intersection-over-union) is below a predetermined threshold (0.3, by default). In semantic segmentation, a proposed region (64x64 block) overlaps with the high-confidence pixels if by excluding these high-confidence pixels, the block's objectness score is below a threshold 0.5.

Listing 1 DDS logic to select feedback regions.

Input: The DNN-generated region proposal R_{dnn} and detection results D_{dnn} on the low-quality video.

Output: Feedback region set R_{feedback} (which the camera then sends again in higher quality level Q_{high})

```
Function create_feedback_per_frame($R_{dnn}, D_{dnn}$):
    /* Initialize feedback region set */
    $R_{\text{feedback}} \leftarrow \emptyset$
    for region $r \in R_{\text{dnn}}$ do
        $keep \leftarrow True$
        /* Skip large regions */
        if $r$.area $> \gamma$ then
            $keep \leftarrow False$
        /* Skip if it overlaps with high-confidence inference result */
        for $b \in D_{\text{dnn}}$ do
            if $r$ and $b$ overlap then
                $keep \leftarrow False$; break
        /* Skip low objectness sore regions */
        if $r$.objectness $< \theta$ then
            $keep \leftarrow False$
        if $keep$ then
            $R_{\text{feedback}} \leftarrow R_{\text{feedback}} \cup \{r\}$
    return $R_{\text{feedback}}$
```

confident based on the low-quality video. (b) filters out large proposed regions, because if it does contain an object, it should have been detected by the DNN with high confidence already. Finally, (c) uses a threshold of objectness score (by default 0.5) as a control knob to adjust the bandwidth demand of the second round (Stream B).

Figure 2.6 shows an example of Algorithm 1 in action (on a single frame). The region proposal R_{dnn} and inference result D_{dnn} are produced by running FasterRCNN-

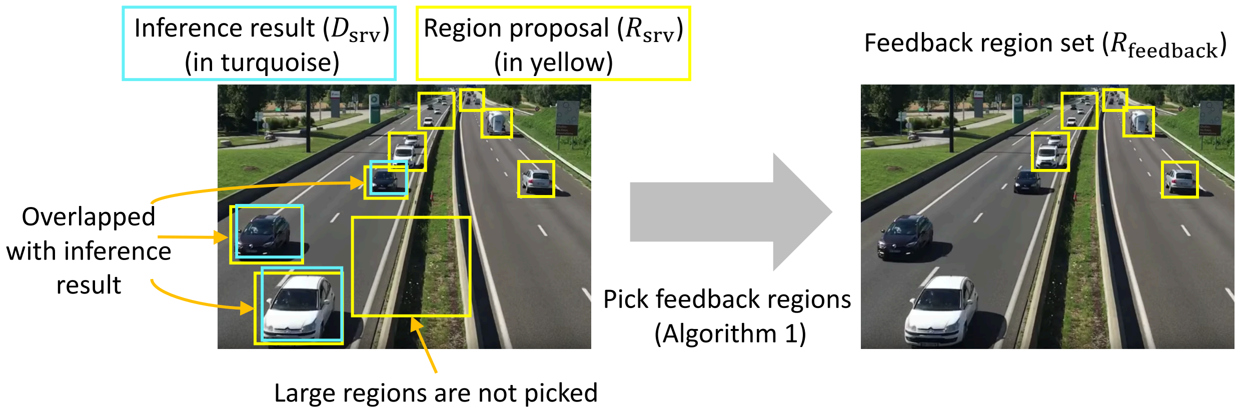

Inference result (D_{srv}) (in turquoise)
Region proposal (R_{srv}) (in yellow)
Feedback region set (R_{feedback})
Overlapped with inference result
Pick feedback regions (Algorithm 1)
Large regions are not picked

Figure 2.6: Illustrative example of DDS's logic to choose feedback region set (Algorithm 1) from region proposal and detected result returned by the server-side DNN.

ResNet101, a typical object detection model, on one frame from one of the traffic video; we get R_{dnn} the FasterRCNN output (after the built-in non-maximum suppression removes duplicated bounding boxes). The example shows that, among the nine DNN-proposed regions, one is too large and four overlap substantially with detected bounding boxes, so the feedback region set contains four regions.

2.3.3 Handling bandwidth variation

Like other video streaming protocols, DDS must adapt its bandwidth usage to handle bandwidth fluctuation. There are three control knobs: the low quality level and the high quality level, and the region proposal objectness threshold θ. We empirically find that these knobs affect the bandwidth/accuracy tradeoff in a similar way (i.e., on the same Pareto boundary; Section2.5.4), so DDS only tunes low quality level and high quality level, while fixing the objectness threshold at 0.5.

To tune the low and high quality regions we implement a feedback control system, illustrated in Figure 2.7. We base this controller on prior work that proposed a virtual, adaptive control system that can be customized for specific deployments [154, 35]. Instantiating the controller requires specifying three things: a bandwidth constraint to be met, feedback for monitoring whether or not the bandwidth constraint is met, and the tunable parameters that affect the bandwidth. For DDS, the bandwidth constraint is the estimated bandwidth for the next segment (labelled (1) in the figure), the feedback is the total bandwidth (for both low and high quality) from the last segment (2), and the tunable parameters are the resolution and quantization parameters of both the low and high quality (3). The controller continually estimates the *base* behavior—in this case, the last segment's bandwidth if the default parameter settings had been used. The controller then takes this base behavior as well as the difference between the desired bandwidth for the next segment and the achieved bandwidth for the previous segment and computes a scaling factor for the base bandwidth. This scaling factor is passed to an optimizer which finds the low and high quality settings that deliver the scaled bandwidth while maximizing F1 score.

Our dynamic adaptation has several useful formal properties based on its use of feedback control. First, the content estimator handles non-linearities in the relationship between the parameters and bandwidth. Intuitively, we can think of the parameter-bandwidth relationship as a curve and the content estimator as drawing a tangent to that curve. The controller then uses that tangent as a linear approximation to the true behavior [77]. The achieved bandwidth, then, will converge to

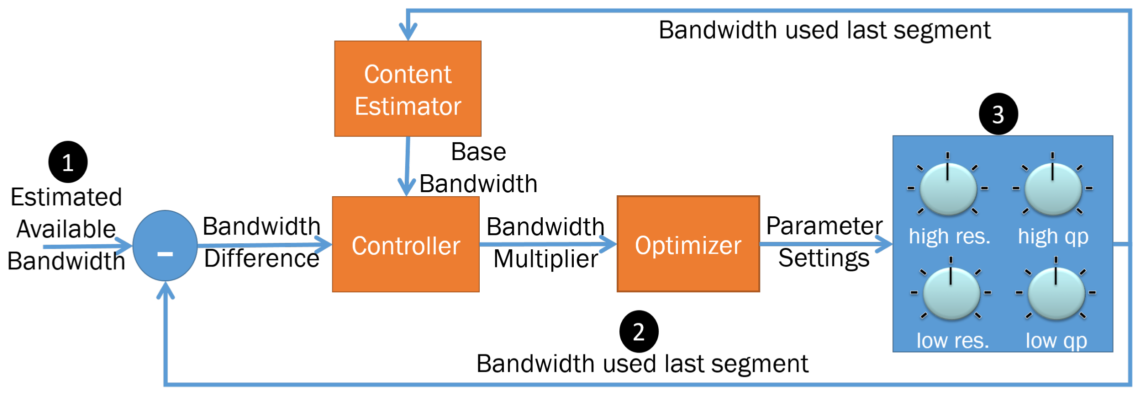


Figure 2.7: DDS's adaptive feedback control system dynamically tunes the low and high quality configurations based on the difference between the estimated available bandwidth for the next segment and that used for the previous segment.

the desired bandwidth in time proportional to the logarithm of the error in this estimation [154]. Second, the optimizer finds the highest quality given the bandwidth specified by the controller. This optimality is achieved by scheduling configurations over multiple segments. As the system has a small, constant number of constraints, an optimal solution can be found in constant time [127].

2.4 Implementation

We implement DDS with about 2K lines of code, mostly in Python and the code is available and will be updated in [8].

2.4.1 DDS Interface

DDS sits between the low-level functions (video codec and DNN inference) and the high-level applications (e.g., object query or face recognition). It provides "south-bound" APIs and "north-bound" APIs, both making minimum assumptions about the exact implementation of the low-level and high-level functions.

The South-bound APIs interact with the video codec and DNN. Our implementation only uses the APIs already exposed by the x264 MPEG video, such as `x264_encoder_encode` [24]. From DNN, DDS implements two functions: (1) region proposals (R_{dnn}), each with a specified location; and (2) detection results (D_{dnn}) including the detected objects/identities each with a specified location and a detection confidence score.

The North-bound APIs implement the same analyst-facing (north-bound) APIs as the DNNs (DDS can simply forward any function call to DNNs), so the high-level applications (e.g., [142, 132]) do not need to change and DDS can be deployed transparently to analysts. The only assumption is that, because DDS runs the DNN twice on the same video segment, the two DNN detection results must be merged into a single detection result; we did this in a similar way as how DNNs internally to merge redundant results (e.g., [193]).

2.4.2 Performance optimization

Saving bandwidth by leveraging codec A naive implementation of Stream B would encode each region as separate images where the requested regions have high quality and the rest of the frames have low quality. But the total size of these images would be much greater than the original video segment in even the higher quality level without cropping the regions! The reason is that video codecs (e.g., H.264/H.265), after decades of optimization, are very effective in exploiting spatial redundancy and temporal redundancy between video frames to reduce the encoded video size. So instead, we leverage this encoding effectiveness. In each frame, we set the pixels outside of the requested regions in the high quality image to black, i.e., RGB=(0,0,0), and encode these images into a video file (mp4).

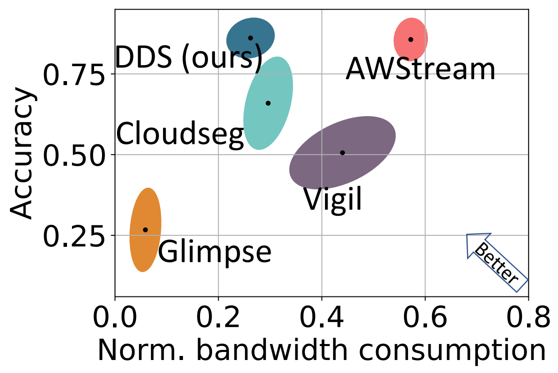
Reducing delay via early reporting The cost that DDS pays to get better performance is the worst-case response delay; the result of Stream B will wait for two "round trips" before it can be returned to the analysts. To address this problem, we leverage the observation that a substantial fraction of the DNN output from the low-quality video (Stream A) already has high confidence and thus can be returned without waiting for Stream B. While this optimization does not change the bandwidth consumption or worst-case response delay, it substantially reduces the delay of many inference results. In object detection, we empirically found that over 90% of all final detected objects could have been detected in Stream A. These objects can be returned much faster than any prior approach, because Stream A uses a quality level much lower than what other work (e.g., [211, 58, 57]) would need to achieve the same accuracy.

Leveraging camera-side heuristics DDS can also leverage camera-side computing power. The key advantage of doing so is not only incorporating more computing power, but to leverage that the cameras have access to the raw (full resolution) video. Here, we use tracking as a camera-side heuristics that can be incorporated in DDS. Tracking is usually less compute-intensive than tasks such as object detection, so some prior work has proposed having the camera track objects (e.g., using [97]) detected by the server-side logic (e.g., [56]). DDS can benefit from a similar idea by requesting fewer frames in the feedback regions and performing camera-side tracking between the requested frames. For instance, given a 30-frame segment, the server will decide regions in each of the 15 frames, but instead of requesting all of them from the camera, it only requests the regions in the first frame and sends back the final results of the first frame to the camera which then tracks them using local tracking logic.

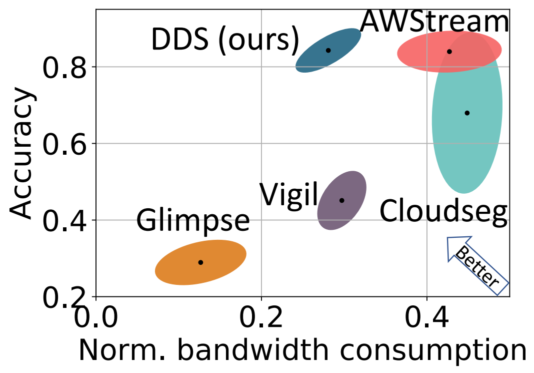
2.5 Evaluation

The key takeaways of our evaluation are:

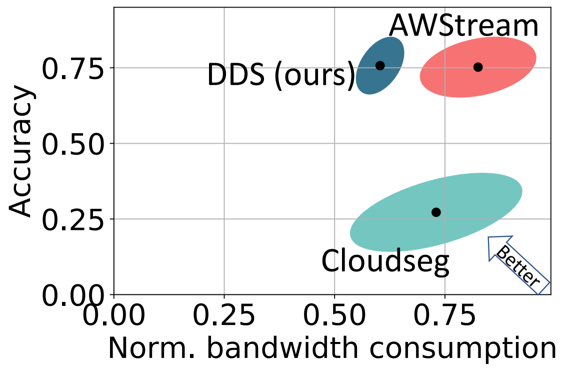
- On three vision tasks, DDS achieves same or higher accuracy than baselines while using 18-58% less bandwidth (Figure 2.8) and 25-65% lower response delay (Figure 2.9).
- DDS sees even more improvements on certain video genres where objects are small (Figure 2.10) and on applications where the specific target objects appear rarely (Figure 2.11).
- DDS's gains remain substantial under various bandwidth budgets (Figure 2.12) and bandwidth fluctuation (Figure 2.13).
- DDS poses limited additional compute overhead on both camera and server (Figure 2.15).



(a) Object detection (Traffic)



(b) Object detection (Dashcam)



(c) Object detection (Drone)

Figure 2.8: The normalized bandwidth consumption v.s. inference accuracy of DDS and several baselines on various video genres shows that DDS achieves high accuracy with 55% bandwidth savings. Ellipses show 1-σ range of results.

Name	*Vision tasks*	*Total length*	*# videos*	*# objs/IDs*
Traffic	obj detect	2331s	7	24789
Drone	obj detect	163s	13	41678
Dashcam	obj detect	5361s	9	24691

Table 2.1: Summary of our datasets.

2.5.1 Methodology

Experiment setup We build an emulator of video streaming that can measure the exact analytics accuracy and bandwidth demand of DDS and our baselines. It consists of a client (camera) that encodes/decodes locally stored videos and a fully functional server that runs any given DNN and a separate video encoder/decoder. Unless stated otherwise, we use FasterRCNN-ResNet101 [169] as the server-side DNN for object detection. DDS extracts region proposals of the object detection pipeline using FasterRCNN[169], those of the face recognition pipeline using MTCNN[214] using the method described in Section 2.3.2. As we will see in Section 2.5.3, different choices of DNNs will not qualitatively change the takeaways. When needed, we vary video quality along QP parameters (from {26,28,30,36,38,40}) and resolution (from scale factors of {0.8,0.7,0.5}), and DDS uses 36 (qp) as low quality and 26 (qp) as high quality, both with resolution scale of 0.8. In most graphs, we assume a stable network connection, but in Section 2.5.4, we will vary the available bandwidth as well as increase bandwdith variance.

Datasets To evaluate DDS over various video genres, we compile five video datasets each representing a real-world scenario (summarized in Table 2.1 and their links can be found in [4]). These videos are obtained from two public sources. First, we get videos from *aiskyeye* [22], a computer-vision benchmark designed to test DNN accuracies on drone videos. Nonetheless, the key difference between DDS and baselines not the DNN but the client-server pipelines which can be affected by other factors such as fraction of frames with objects of interest or size of the regions with objects. To cover a range of dynamism along these features, we also get videos from YouTube as follows. We search keywords (e.g., "highway traffic video HD") in private browsing mode (to avoid biases from personalization); among the top results, we remove the videos that are irrelevant (e.g., news report that mentions traffic), and we download the remaining videos in their entirety or the first 10-minutes (if they exceed 10 minutes). The vision task is to detect vehicles in traffic and dashcam videos, to detect humans in drone videos. Because we use public sources to cover more real-world videos, many videos do not have human-annotated ground truth. So for fairness, in all videos in our dataset, we use the DNN output on the full-size original video as the reference result to calculate accuracy. For instance, in object detection, the accuracy is defined by the F1 score with respect to the server DNN output in highest resolution (original) with over 30% confidence score.

Baselines We use four baselines to represent two state-of-the-art techniques (see Section 2.1.2): camera-side heuristics (Glimpse [56], Vigil [216]) and adaptive streaming (AWStream [211], CloudSeg [205]). Following minor modifications are made to ensure the comparison is fair. First, all baselines and DDS use the same server-side DNN. Second, although DDS needs no more camera-side compute power than encoding, camera-side heuristics baselines are given enough compute resource to run more advanced tracking [97] and object detection algorithm [174] than what Glimpse and Vigil originally used, so these baselines' performance is strictly better than their original designs. Third, all DNNs used in baselines and DDS are pre-trained (i.e., not transfer-learned with samples from the test dataset like in NoScope [122]); so our version of CloudSeg uses the pre-trained super-resolution model [26] from the website [15]. This ensures the gains are due to the streaming algorithm, not due to customization of DNN, and also helps reproducibility. Finally, although DDS could lower frame rate, but to ensure accuracies of all baselines are calculated on the same set of images, we only tune resolution and QP in DDS, which are also the most effective knobs [211].

Performance metrics We use the definition of accuracy and response delay from Section 2.1.1. To avoid impact of video content on bandwidth consumption, we report bandwidth demands of DDS and baselines after normalizing them against the bandwidth consumption of the original videos.

2.5.2 End-to-end improvements

We start with DDS's overall performance gains over the baselines along bandwidth savings, accuracy, and response delay.

Bandwidth savings Figure 2.8 compares the bandwidth-accuracy tradeoffs of DDS with those of the baselines, when the total available bandwidth is set to match the size of the original video (we will vary the available bandwidth in Section 2.5.4). Across three vision tasks, DDS achieves higher or comparable accuracy than AW-

Stream and CloudSeg but uses 55% less bandwidth in object detection. Glimpse and Vigil do sometime use less bandwidth but they have much lower accuracy. Overall, even if DDS is less accurate or uses more bandwidth, it always has an overwhelming gain on the other metric.

Response delay Figure 2.9 shows the response delay of DDS and AWStream (the baseline that is the closest to DDS performance-wise) with the same length of a segment (number of consecutive frames encoded as a segment before sent to the server). The segment length limits the lower bound of the delay, and same segment length, we see that despite the need for two iterations, DDS reduces the response delay by 25-65% compared to AWStream. Moreover, the gap widens as the segment length increases. To put it into perspective, popular video sites use 4-second to 8-second segments [217] (i.e., over 120 frames per segment), a range in which DDS's gains are considerable.

Intuition DDS achieves significant bandwidth savings compared to AWStream because DDS is driven by DNN-generated feedback on the real-time video content while AWStream is content agnostic. Thus, DDS aggressively compresses video by cropping out only regions of interest in the second iteration. The bandwidth savings compared to Glimpse or Vigil result from the fact that camera-side heuristics lack crucial server-side information required to select the important frames. For fairness, we do not customize the super-resolution model to the specific video content; as a result, the super-resoluted frames actually leads to lower accuracy from the server DNN. DDS's response delay is much lower than AWStream because it detects most

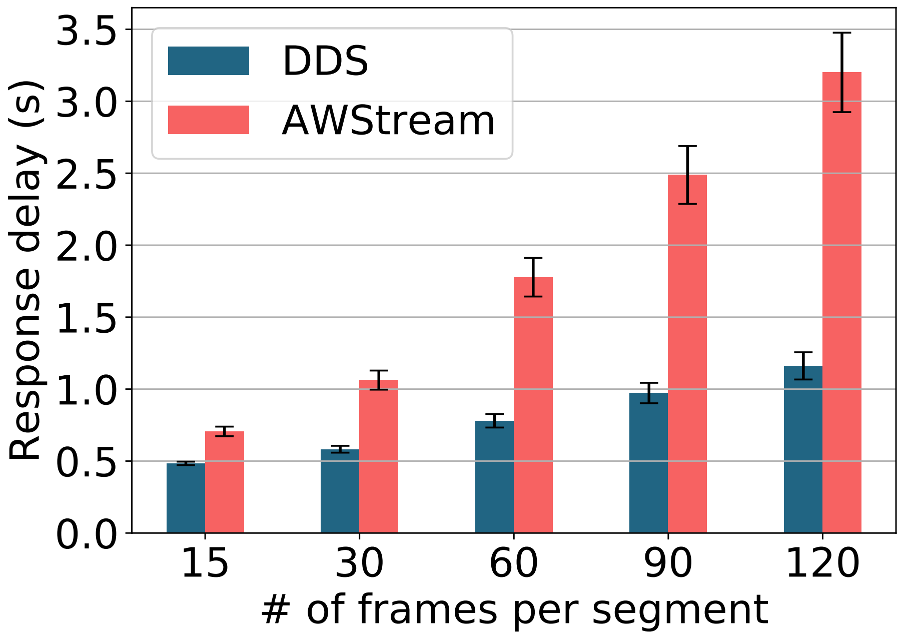


Figure 2.9: Response delay of DDS is consistently lower than AWStream under various lengths of video segment.

of the objects in the first, low quality iteration. AWStream sends a single video stream and spends more time to encode that stream with quality.

2.5.3 Sensitivity to application settings

Impact of video genres Next, Figure 2.10 shows the distribution of per-video bandwidth savings w.r.t AWStream (dividing AWStream's bandwidth usage by DDS's when DDS's accuracy is at least as high as AWStream) in three datasets. As expected, there is substantial variability of performance among videos of the same type. This is because DDS's gains depend on the fraction of pixels occupied by objects of interest, which can varies with videos.

That said, the impact of content on performance gains can also be task-dependent. In fact, it highlights the different nature between the two vision tasks. In object detection, in presence of a large object that cannot be confidently classified, DDS will need to send a large region in its entirety to the server. Dashcam videos have more large objects than drone/traffic camera videos, so contrast will appear only in dashcam videos.

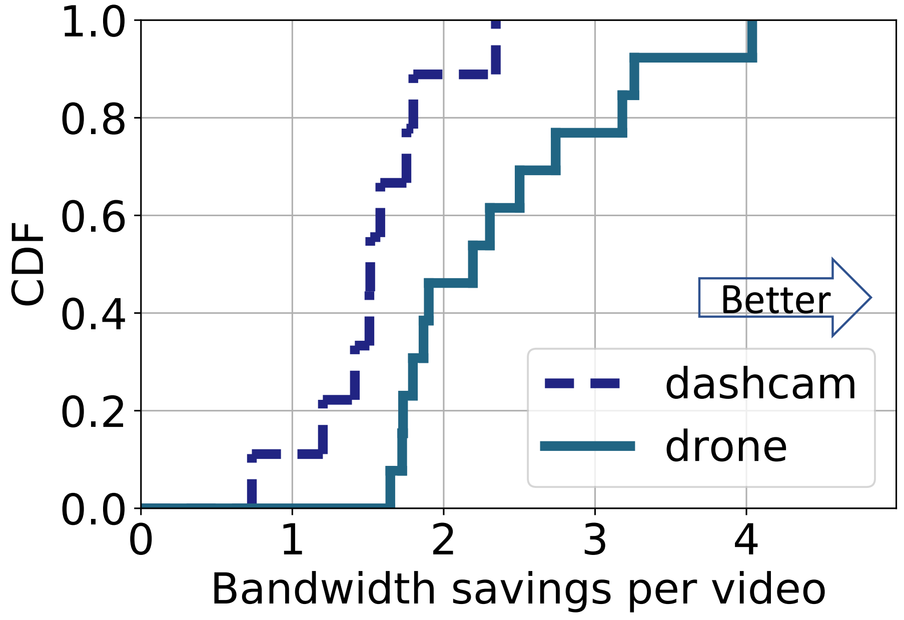


Figure 2.10: Distributions of per-video bandwidth savings in two datasets. The gains of DDS are video dependent.

Impact of inference tasks So far we have evaluated DDS when the vision tasks deal with the common object classes (or all classes) in videos. An additional advantage of DDS is it can save more bandwidth by taking advantage of the server-side application when needs only a few specific object classes. To demonstrate this, we change the segmentation task from detecting all objects to detecting only motorcyles. Figure 2.11 shows the DDS's bandwidth savings (when achieving same or higher accuracy than AWStream) on three traffic videos in which motorcyles appear for a small non-zero fraction of frames, and compare the savings with those when the task is over all classes. We see that DDS's gains are significantly higher when only motorcyles are the segmentation target, and the additional gains are higher when the motorcyclists take a smaller fraction of pixels and frames (e.g., Video 2).

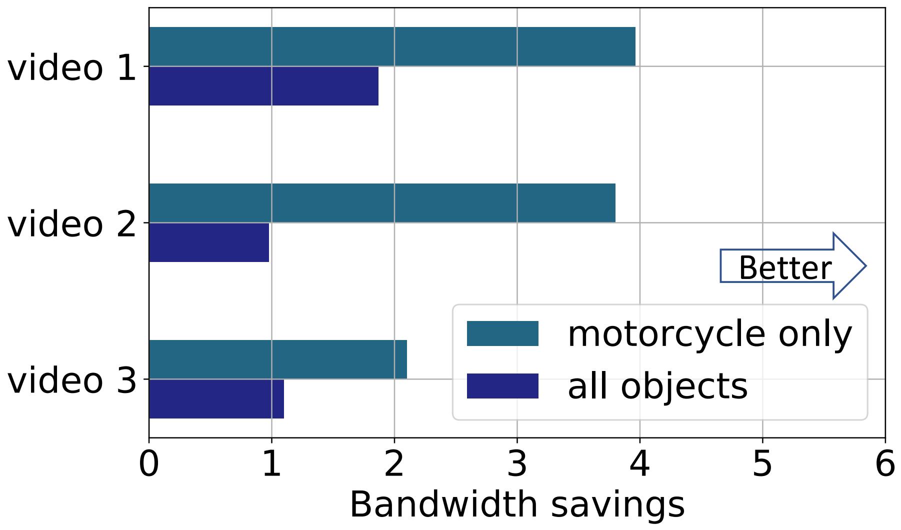


Figure 2.11: Segmentation on only motorcycles achieves 2-4× more bandwidth savings than segmentation on all classes.

Impact of DNN architecture Last but not least, we test different DNN architectures on a randomly selected traffic video (5-minute long), and find that DDS achieves substantial bandwidth savings under different server-side DNN models: FasterRCNN-ResNet101 (44%) and FasterRCNN-ResNet50 [18] (54%) has the same architecture but different feature extractors, while Yolo [167] (51%) uses a different architecture and feature extractor. This implies that we can design the video streaming protocol agnostic to the model architecture of the server-side DNN. We leave a full examination of different DNN architecture (e.g., MaskRCNN [94]) to future work.

2.5.4 Sensitivity to network settings

Accuracy vs. bandwidth budget We then vary the available bandwidth and compare DDS with AWStream, which is performance-wise the closest baseline. Figure 2.8 shows that as available bandwidth varies, DDS always uses 35-43% bandwidth in object detection while achieving higher accuracy.

Impact of bandwidth variance Figure 2.13 compares DDS with AWStream under increasing bandwidth variance. We use synthetic network bandwidth traces where available bandwidth is drawn from a normal distribution of $900 \cdot N(1, \sigma^2)$kbps while we increase σ from 5% to 100%. We observe that DDS maintains an accuracy advantage over AWStream. Although DDS and AWStream use the same bandwidth estimator (average of last two segments), DDS uses the available bandwidth more efficiently because DDS's feedback control system continually adapts the model relating configuration parameters to bandwidth. Thus, DDS adaptively selects the best possible configuration parameters at each time instant. Further, the available configuration parameters for low and high quality video enable a wider range of possible adaptations and thus DDS tunes its behavior at a finer granularity compared

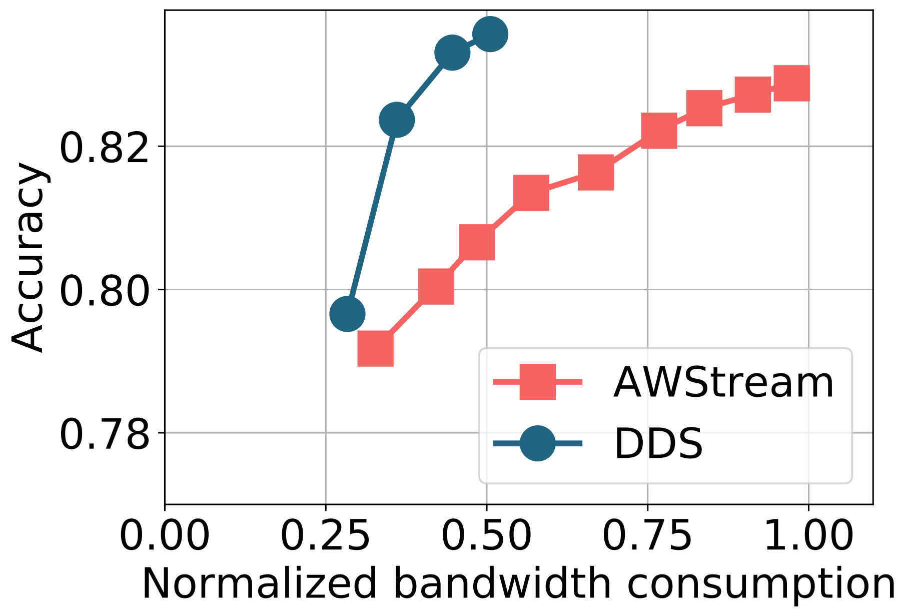
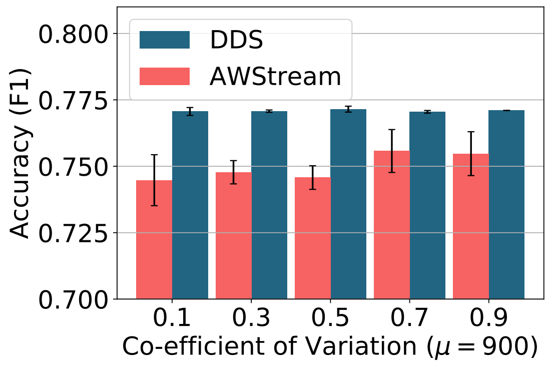


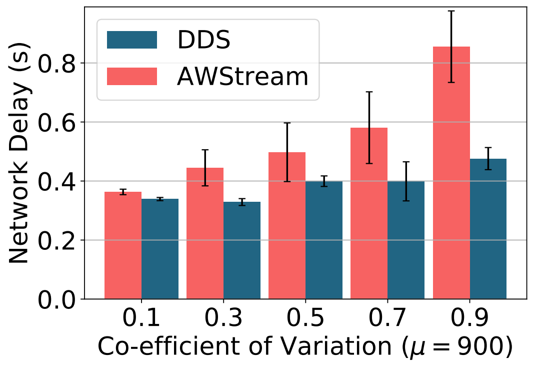
Figure 2.12: DDS outperforms (the closest baseline) in accuracy under various bandwidth consumption budgets.

to AWStream. Even when the variance in available bandwidth is high ($\sigma > 70\%$ of the mean), DDS maintains a relatively low response delay while AWStream's delay increases.

Impact of parameter settings Figure 2.14 shows the impact of key parameters of DDS on its accuracy/bandwidth tradeoffs: the QP of the first iteration in Stream A ("low qp"), the QP of the second iteration in Stream B ("high qp"), and the objectness threshold. We vary one parameter at a time and test them on the same set of traffic videos. The figures show that by varying these parameters, we can flexibly trade accuracy for bandwidth savings. Overall, we observe their effect roughly falls on the same Pareto boundary, so there may not be significant difference between the choices



(a) Accuracy



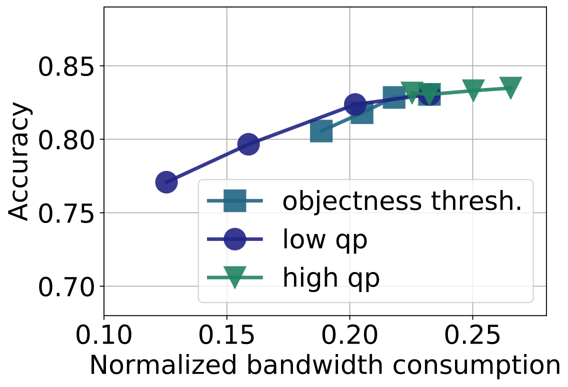
(b) Network delay

Figure 2.13: DDS can handle bandwidth variance and maintain a sizeable gain over the baseline of AWStream even under substantial bandwidth fluctuation.

of parameters to vary when coping with bandwidth fluctuation.

2.5.5 System microbenchmarks

Camera- & server-side overheads Figure 2.15 puts the systems overhead of DDS into the perspective of the baselines. We follow the assumptions in Section 2.5.2 that the camera has 4 CPUs and the server has 4 CPUs and 8 GPUs. On the camera (client) side, all methods use relatively low overhead, except Vigil and Glimpse which run local heuristics of tracking and object detection logic. On the server side, Vigil and Glimpse use much less GPU and CPU cycles than DDS and AWStream, because they run inference only on a small fraction of frames (which in part leads to lower accuracy). The reason DDS has lower GPU usage than AWStream, despite running inference twice per frame, is that AWStream profiles a list of configurations (e.g.,

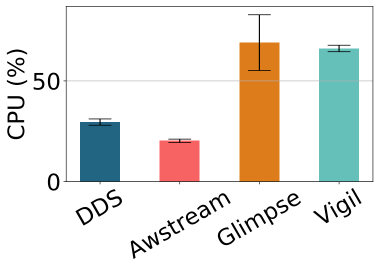


(a) Object detection

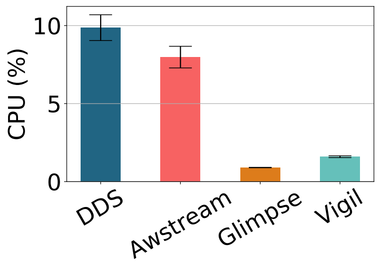
Figure 2.14: Sensitivity analysis of DDS's parameters. By varying these parameters, DDS can flexibly adapt itself to reach desirable accuracy for a given bandwidth constraint.

pairs of QP and resolution) (total 216 configuration in [211]) and that cost is non-trivial. In our implementation, we evaluate only 24 configurations on 15 frames in each 300-frames video, and the average GPU usage of AWStream is already higher than DDS. That said, we acknowledge that if AWStream updates the profile less frequently (e.g., every few minutes), its GPU usage could be lower than DDS, but that might cause its profile to be out of date.

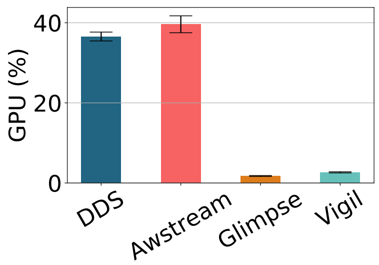
Fault tolerance We stress test DDS under the condition that the server-side DNN is unreachable. By default, the camera runs DDS protocol, and it also has a local tracking algorithm as a backup. Figure 2.16 shows the time-series of response delay and accuracy. First, DDS maintains desirable accuracy, and at $t = 5$ second, the server is disconnected. We see that DDS waits for a short time (until server times out at $t = 5.5$) and fall back to tracking the last detection results from the server DNN.



(a) Client CPU



(b) Server CPU



(c) Server GPU

Figure 2.15: Compared to prior solutions, DDS has low additional systems overhead on both client and server.

This allows for a graceful degradation in accuracy, rather than crashing or delaying the inference indefinitely. Between the server disconnection and the timeout, the segments will be placed in a queue temporarily, and when the local inference begins, the queue will be gradually cleaned up. When the server is back online (at $t = 13$), the camera will be notified with at most a segment-worth of delay (0.5 second), and begin to use the regular DDS protocol. Meanwhile the camera will continue to send a probe per segment.

Performance optimization Figure 2.17 examines two performance refinements. First, figure 2.17(a) shows that (1) putting the proposed regions on a black back-

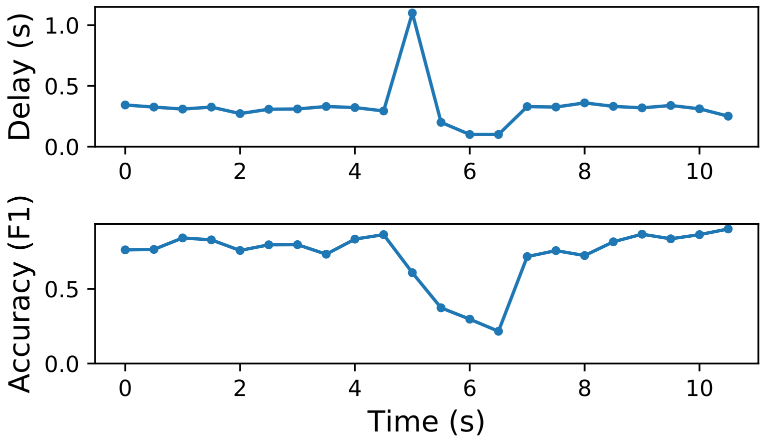


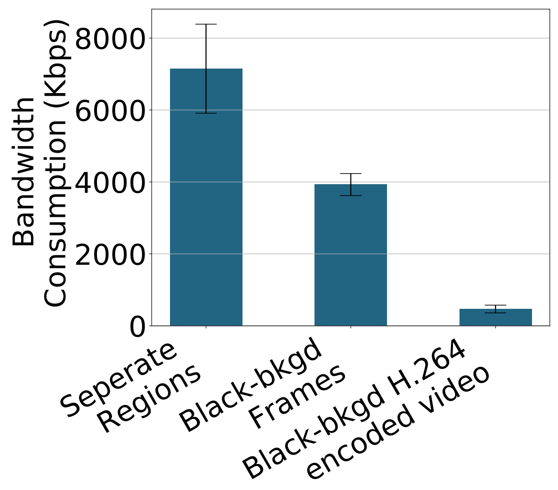
Figure 2.16: DDS can handle server disconnection (or server failure) gracefully by falling back to client-side logic

ground frame yields about 2× bandwidth savings over encoding each region separately and (2) compressing these frames in an mp4-format leads to another 10× bandwidth savings. Second, figure 2.17(b) shows that returning the first-iteration output, i.e. the high-confidence results in Stream A, before second iteration starts, we reduce the average response delay by about $\sim 40\%$.

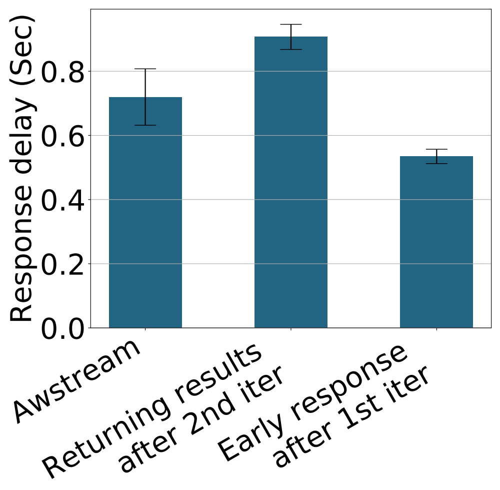
2.6 Related work

We discuss the most closely related work in three categories.

Video analytics systems The need to scale video analytics has sparked much systems research: DNN sharing (e.g., [116, 109]), resource allocation (e.g., [212, 135]), vision model cascades (e.g., [122, 180]), efficient execution frameworks (e.g., [142, 132, 160]), as well as camera/edge/cloud collaboration (e.g., [56, 216, 58, 211, 190, 205],



(a) Reducing bandwidth usage by smarter encoding



(b) Reducing response delay by early response

Figure 2.17: System refinements (introduced in Section 2.4.2) to (a) reduce Stream B bandwidth and (b) reduce response delay.

see Section2.1.2 for a detailed discussion) or multi-camera collaboration (e.g., [115, 117]). The most related work to DDS is AWStream [211] which shares with DDS the ethos of using a server DNN-generated profile. The key distinction is that such feedback is not real-time video content, so it cannot zoom in on specific regions on the current frames. Vigil [216] on the other hand does send cropped regions, but it is bottlenecked by the camera computing power (see Section2.1.2).

Vision applications Computer vision and deep learning research has a substantial body of research (e.g., [168, 169, 166, 209, 119, 139, 66]). Recent works on video object detection show it is inefficient to apply object detection DNN frame by frame; instead it should be augmented by tracking [98] (similar to Section2.4.2) or by a temporal model such as LSTM [138, 136]. This work complements DDS by designing new, server-side deep learning models. DDS's distinctive advantage is that it explicitly optimizes the bandwidth/accuracy tradeoffs in a way that is largely agnostic to the server-side DNN.

Internet video encoding/streaming Recent innovations in video encoding have provided better compression gains [59]. The closest efforts to DDS are scalable video coding [156] and region-of-interest encoding [153]. However, these approaches optimize human quality of experience (QoE) and are largely agnostic to what is in the video. Region-of-interest encoding requires the viewer to specify the region of interest. Scalable video coding can utilize the bandwidth more efficiently than traditional encoding methods, but it still compresses video uniformly in its entirety. Similarly, much work has been done in adaptive bitrate streaming (e.g., [178, 118, 131, 70]), which focuses on adapting bitrate of pre-coded video chunks to bandwidth fluctuation, rather than dynamic content as in DDS.

2.7 Limitations and discussion

Strict server-side resource budget In some sense, DDS reduces bandwidth usage at the expense of relying on server-side DNN to run inference more than once per frame. Similar tradeoffs can be found in AWStream, which triggers costly reprofiling periodically, and CloudSeg, which enforces an upfront super-resolution model customization process. All these techniques may not be directly applicable where server resources have strict budgets or GPU cost is proportional to its usage (e.g., cloud instances).

Implication to privacy: Privacy is an emerging concern in video analytics [200]. While DDS does not explicitly preserve privacy, it is amenable to privacy-preserving techniques. Since DDS does not send out full resolution image, it could be repurposed to denature videos before sending only a part of the video to the server for analytics.

Edge AI accelerators: Though DDS makes little assumption on camera's local computation capacity, it can naturally benefit from the trend of more accelerators being added on edge devices, by using camera-side heuristics as described in Section 2.4.2. We also envision DDS running along side the camera local analytics to share the workload to provide higher inference accuracy with minimal bandwidth overheads.

2.8 Conclusion

Video streaming has been a driving application of networking research that has inspired innovative design paradigms. This work argues that the emerging AI applications inspire a paradigm shift away from the basic source-driven approach to video streaming. We advocate for a DNN-driven design that exploits opportunities unique to deep learning applications: (1) unlike user QoE, video inference accuracy depends less on pixels than on what is in the video, and (2) deep learning inference (receiver) offers extra information that can be leveraged to decide how video should be encoded/streamed. We believe development of such video streaming protocols will significantly impact not only video analytics, but also the future analytics stack of many distributed AI applications.

CHAPTER 3

GOAL: GOAL-ORIENTED PROGRAMMING LANGUAGE

Chapter 2 provided details of a novel iterative algorithm for machine-learning centric streaming. The presented concrete implementation, DDS, was augmented with a control-theoretic adaptation module that allowed it to cope with changes in video content and changing bandwidth availability by modifying the parameters for its high and low quality iterations. The adaptation module allowed DDS to handle variations in bandwidth and video content significantly better than heuristics-based prior works. However, while implementing the adaptation module for DDS we noticed a significant issue in the approach towards adaptation in computer systems.

Along with functional correctness, modern computer systems need to provide certain quality-of-service guarantees [91, 134, 107]. However, it is a well established fact that no single parameter configuration is optimal for the entire lifetime of the system as the optimal configurations depend on factors outside of the control of the application [202, 134, 197]. Hence, prior work has suggested using principled adaptation to allow computer systems to dynamically modify their configurations to ensure that they meet their quality-of-service goals in the face of changing operating conditions and workloads.

However, a major issue with prior work is that it requires the details of the computer system that is to be adaptation, e.g. its parameters, important metrics and quality-of-service goals, to be enumerated before the adaptation module can be constructed. As such, a major limitation of prior work becomes that they are implemented for a specific, narrow set of goals and parameters. For example, the adaptation module in DDS was built specifically for DDS and supports only the goal of maximizing accuracy while meeting a bandwidth constraint using only the high and low quality knobs. This impededs the development of complex adaptive systems that must meet different goals using different sets of knobs for different deployments, or even change goals dynamically during one deployment. Another disadvantage of such an approach to adaptation is that the purpose-built adaptation module has to either be completely scrapped or needs to be heavily modified when adaptation needs to be added to another application. This is a significant deviation from good practices for engineering computer systems which hold that components should be designed to be easily reusable between different systems.

To overcome this limitation, we suggest a fundamental change in the way adaptation is handled in computer systems. All aspects of adaptation should be treated as first-class programming objects that the computer system can interact with dynamically through convenient interfaces. Hence, in this chapter we present GOAL, an adaptation framework distinguished by its virtualized adaptation logic implemented independently of any specific goals or knobs. GOAL supports this logic with a programming interface allowing users to define and manipulate a wide range of goals and knobs within a running program. We demonstrate GOAL's benefits by using it re-implement seven different adaptive systems from the literature, each of which has a different set of goals and knobs. We show GOAL's general approach meets goals as well as prior approaches designed for specific goals and knobs. In dynamic scenarios where the goals and knobs are modified at runtime, GOAL achieves 93.7% of optimal (oracle) performance while providing a 1.69× performance advantage over existing frameworks that cannot perform such dynamic modification.

This chapter is organized as follows: Section 3.1 motivates the need for adaptation, discusses approaches suggested by prior work and their shortcomings, Section 3.2 provides a tour of the GOAL framework by implementing adaptation in an example application and discusses how GOAL overcomes the disadvantages of prior work. Section 3.3 provides details of the underlying GOAL runtime, Section 3.4 provides details of the GOAL API and its novel Adaptation Specification Language. In Section 3.5 and Section 3.6 we provide details of evaluation methodology and present our evaluation results that illustrate the generality and robustness of GOAL. In Section 3.7 we present a brief discussion of the cost semantics and static analysis of GOAL-based systems. Finally, in Section 3.8 we conclude our discussion of GOAL. Additionally, we provide implementation and execution details of individual applications used for evaluating GOAL in Appendix A.

3.1 Related Work and Motivation

Adaptation is a key mechanism for building robust software systems that operate effectively despite unpredictable dynamic changes to inputs or operating environment [133, 126]. This section discusses prior approaches to building adaptive software systems, and discusses how limitations of prior works motivate the need for adaptation as a first class programming construct.

3.1.1 Complexity Requires Adaptation

Computer systems have numerous configuration parameters and settings that impact their ability to meet their quantifiable behavioral goals [126]. Improper configuration is a notorious source of performance issues and bugs [107, 163, 203]. Selecting a good configuration is difficult because optimal configurations depend upon dynamically varying external factors such as workload and operating conditions [91, 134, 197]. Adaptive systems address this problem by automatically and dynamically adjusting configuration parameters to ensure goals are met. Thus, there is a need for principled approaches to building adaptive computing systems recognized by both industry [126, 133, 44, 157, 95, 85, 104] and academia [73, 105, 158, 80].

Two design patterns—Observe-Orient-Decide-Act (OODA) [171, 45, 46] and Monitor-Analyse-Plan-Execute (MAPE) [126]—have been proposed for creating adaptive software. Both establish a *control loop* as the basic structure for adaptation. During a loop iteration the software first *observes/monitors* its environment and its own quantifiable behavior. It then *orients/analyzes* itself with respect to these metrics to *decide/plan* what should be done next. The subsequent iteration then *acts/executes* these decisions by changing the values of configuration parameters. Because it results in more robust and flexible software, many approaches implement adaptive loops in the OODA/MAPE pattern.

3.1.2 Existing Support for Adaptation

As mentioned earlier, prior work models the OODA/MAPE design pattern as a control loop and several researchers have proposed that general scheme as a basis for system [101], software [36], and language [170] design. Many existing works suggest control theory [78, 210, 215, 79, 96, 187, 179, 34, 149, 175, 198], machine learning [41, 204, 113, 68, 93], and combinations of the two [194, 130, 100, 154, 102] as the basis for building principled *AdaptLogs* that perform the orientation/analysis and decision/planning phase of the OODA/MAPE loop to dynamically adjust configuration knobs in a running computer system. Such approaches provide formal, mathematical guarantees about the precise assumptions and operating conditions under which the goals will be met. However, a challenge is that specialized knowledge in control, learning, or both is required to successfully deploy such approaches.

To make principled adaptation easy to implement, recent work proposes *adaptation frameworks* that package a control- or learning-based AdaptLog into a programming language [29, 183, 50, 33, 173, 219, 43, 123, 49, 129] or a library [87, 215, 110, 203, 164, 63, 154, 192, 54, 80, 182] runtime. Developers do not need to possess specialized knowledge in learning or control to use these frameworks. Instead, they instantiate it with an initial AdaptSpec, after which the AdaptLog monitors the metrics and independently tunes the knobs to meet the goal.

While the OODA/MAPE design patterns themselves provide a general strategy, their implementations in existing approaches have significant limitations. These limitations arise because each framework is designed to support specific metrics and knobs and their AdaptLogs do not generalize to the metrics and knobs in other works. In other words, the OODA/MAPE concepts are generalizable, but specific instantiations of these concepts are problem-specific. Table 3.1 illustrates this idea, showing that while there is wide support for different goals and knobs across frameworks, the

Framework	**Supported Goals**		**Supported Knobs**	**Runtime Modifications**
	Constraint Metric	**Objective Function**		
Aeneas [29]	App. level constraint	Energy	App. parameters	N/A
SiblingRivalry [29]	Throughput	Power	Alternate func. implementations	Constraint target value
Green [33]	App. level constraint	Power or Accuracy	Code approximation	N/A
Eon [183]	Power	Accuracy	Alternate func. implementations	N/A
JouleGuard [100]	Energy	Accuracy	App. and Sys. parameters	N/A
Truffle [75]	App. Level constraint	Energy	Code approximation	N/A
Odyssey [80]	Accuracy	Energy	App. alternatives and Sys. parameters	N/A
GOAL (this paper)	**Any**	**Any**	**Any first-class object**	**Goals, Metrics, and Knobs**

Table 3.1: Comparison of Selected Adaptation Frameworks

support provided by any one is specific and thus limited. For example, Green uses a heuristic AdaptLog based on a specific model of how alternative function implementations affect an application's power and accuracy tradeoffs [33]. Similarly, Aeneas's reinforcement learning model uses a reward signal based on energy measurements, and introducing new metrics requires a new reward function and learning model. To the best of our knowledge, there is no single framework that generalizes across a wide range of goals and knobs. This limitation also means that existing frameworks cannot be used for meta-adaptation because they do not support any alternative AdaptSpecs.

Overcoming these limitations requires users to extend the framework's internal AdaptLog for additional AdaptSpecs. This entails reconstructing the model and reimplementing the AdaptLog and its interface to support other knobs, metrics, goals and meta-adaptation. However, doing so defeats the purpose of using an adaptation framework because it requires users to have specialized knowledge of the AdaptLog.

While prior work has also explored making adaptation components configurable [27], we believe that performing meta-adaptation using such works is difficult because such works encapsulate the application itself rather than making the adaptation framework a natural component of the application which would allow it to exert fine grained control on all aspects of adaptation.

Thus, we argue that the aforementioned limitations would be best addressed through a generalized adaptation framework that allows applications to interact with all aspects of adaptation dynamically. However, developing such a framework is not trivial because it requires a uniform interface and an AdaptLog whose underlying model allows the declaration and use of AdaptSpecs that work with any user-defined measures, knobs and goals. Furthermore, to enable meta-adaptation, the framework must coordinate the interface with the runtime to ensure that goals are met even when the AdaptSpec changes. Our contribution is such a framework, GOAL.

3.2 Implementing Adaptation with GOAL

We provide a high-level overview of GOAL, later sections detail its AdaptLog (Section 3.3) and interface (Section 3.4). We implement a meta-adaptive video encoder for a CCTV camera that meets a target frame rate, while maximizing quality on line power and minimizing energy on battery. Figure 3.1 shows this example: the original Swift code without adaptation (a), the GOAL version (b), and an example GOAL AdaptSpec (c).

3.2.1 Original (Non-adaptive) Code

As shown in Figure 3.1a, the developer initializes the encoder and defines parameters: `qp` and `me` (quantization parameter and motion estimation algorithm, respectively), which govern tradeoffs between the video quality and frame rate. Subsequently, the system enters a while loop where it calls the `encodeNextFrame` method with `qp` and `me` as input. The `record` function logs per-frame encoding quality.

The code in Figure 3.1a neither meets a target frame rate, nor reacts to changes in power source. Carefully selecting static values for `qp` and `me` could ensure the frame rate; however, without adapting to frame-to-frame differences, quality will be sacrificed (e.g., set the parameters for the worst case and live with lower quality for

```
let encoder = initEncoder()
var qp = 30
var me = 1

while(true)
{

  encoder.encodeNextFrame(qp,
    me)
  record(encoder.getQual())


}
```

(a) Original Code

```
import GOAL
let encoder = initEncoder()
var qp = Knob("qp", 30)
var me = Knob("me", 1)

optimize("encoder", [qp, me])
{

  encoder.encodeNextFrame(qp.get(),
    me.get())
  measure("quality",
            encoder.getQual())

}
```

(b) GOAL Code

```
goal encoder
  max(quality)
    such that throughput == 30.0
measures
  quality: Double
  energy: Double
  throughput: Double
knobs
  qp = [10, 30 reference]
  me = [1 reference, 5]
  coreFreq = [600, 1200 reference]
  numCores = [2, 4 reference]
such that
  numCores * coreFreq > 2400
```

(c) Adaptation Specifications

Figure 3.1: The encoder application.

other cases). In fact, video encoding is a challenge problem for adaptive computing because it is difficult to tune encoding parameters [145].

3.2.2 Adding Adaptation with GOAL

GOAL provides constructs that allow developers to implement all parts of the OODA/MAPE control loop with minor modifications to existing software. Concretely, the developer needs to: (1) identify and declare configurable components as `Knob`s, (2) specify the code segment in which to perform adaptation, and (3) report application-level metrics to the GOAL runtime using `measure`.

Figure 3.1b illustrates the CCTV system implemented with GOAL. `qp` and `me` are declared as `Knob`s, using a stringified name and a default value for initializing the knob. The relevant metrics to be monitored are identified using `measure`. For example the encoding `quality` is reported for the CCTV. The `while` loop is replaced with GOAL's `optimize` loop (Section 3.4.1) whose inputs are the list of `Knob`s to tune, and the loop body. This syntax tells the GOAL runtime to iteratively execute the loop body (as in the original program) while tuning the knobs after each iteration, according to the AdaptSpec. Hence, with such minor modifications we have converted the non-adaptive application to an adaptive application that implements a complete OODA/MAPE control loop.

3.2.3 Writing Adaptation Specifications

The next step is writing an AdaptSpec. Figure 3.1c shows an example, defining the `knob`s, allowed values for each, the `measure`s to monitor, and the *goal* that formally specifies the constraints and objective. This example is how the CCTV should behave on line power: maximize quality, with a throughput of 30 iterations per second (which translates to 30 frames/s). The application-level metrics (e.g., quality) and knobs (e.g. `qp` and `me`) in the AdaptSpec must match the string arguments in calls to `Knob` and `measure`. GOAL's runtime automatically declares metrics such as throughput, energy, etc. and hardware specific knobs such as core count (`numCores`) and DVFS frequency (`coreFreq`), allowing them to be used without explicit declaration.

This AdaptSpec tells the runtime to measure quality, energy, and throughput, while tuning `qp`, `me`, `numCores` and `coreFreq`. Additionally, the AdaptSpec states that the runtime must choose configurations that meet a further constraint on the knobs: the product of `numCores` and `coreFreq` should be greater than 2400. See Section 3.4.3 for a full description of the DSL's grammar and capabilities.

GOAL AdaptSpecs are compiled just-in-time (Section 3.3.2) and can be written in text files independent of the application code. This allows developer's to produce a single binary that can be deployed to meet different goals by writing different AdaptSpecs. Such flexibility distinguishes GOAL from existing adaptation frameworks.

Given the GOAL system and an AdaptSpec, the last step before deployment is to use GOAL's model builder (Section 3.3.3), which runs the application on test inputs to learn a function the knob configurations' impact on the specified metrics. Figure 3.2 presents a high-level overview of the OODA/MAPE loop implemented using GOAL. During execution, GOAL monitors the metrics identified by `measure`, analyzes and plans which values to use for each of the `Knob`s to meet the goal specified in AdaptSpec, then sets the `Knob`s to those values and executes the loop body.

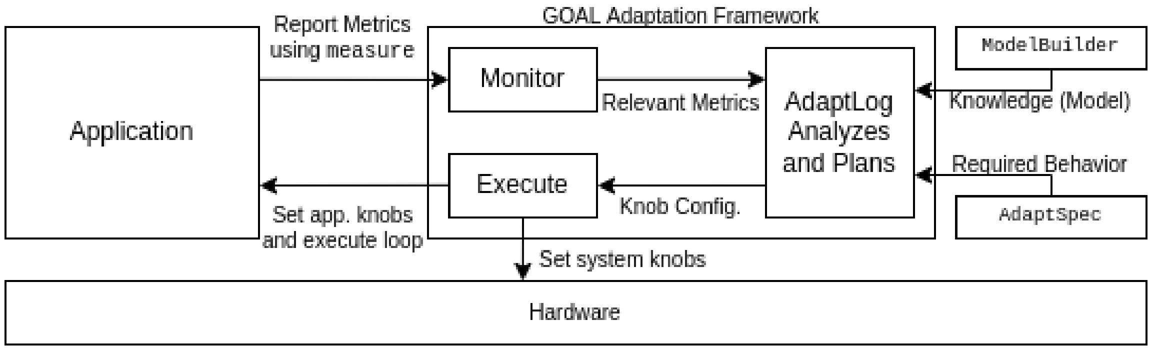


Figure 3.2: The GOAL Adaptation Framework.

Finally, the system can dynamically change any aspect of the AdaptSpec during execution, telling GOALs runtime to meet new goals, use new knobs, or both.

3.2.4 Adding Meta-Adaptation

The CCTV requires meta-adaptation to change goals based on power supply. GOAL's `intend` method uniquely supports this by dynamically changing the AdaptSpec in a running system. The `intend` function takes a string representation of the goal and replaces the active goal with this argument. To illustrate this, we augment the `optimize` loop in Figure 3.1b with the following code (not shown in the figure):

```
if getPowerSupply() == .DirectPower  {
  intend(to: .maximize, objective: "quality",
  suchThat: [(measure: "throughput",
    goal: 30.0)])
} else if getPowerSupply() == .Battery {
  intend(to: .minimize, objective: "energy",
  suchThat: [(measure: "throughput",
    goal: 30.0)])
}
```

The code in red specifies the parameters of `intend` and the code in blue represents the arguments specific to the requirements of the CCTV system.

With this handful of changes, the encoder will now meet the throughput constraint while optimizing either quality or energy based on the power source. This large increase in adaptive capability for small code changes highlight how GOAL's general adaptation framework supports complex adaptive behavior and seamless meta-adaptation.

3.2.5 Quantitative Benefits of Using GOAL

Figure 3.3 compares the execution of our GOAL CCTV with a version that uses prior work to synthesize a customized, Application-specific AdaptLog [76]. However, using prior work, the CCTV can only meet one goal; we begin with the goal corresponding to line power: maximize quality and meet a throughput constraint.

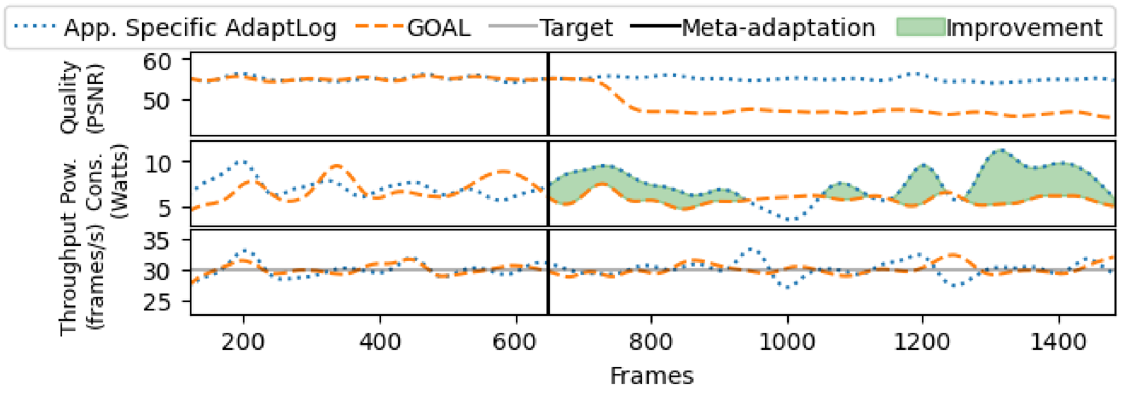


Figure 3.3: GOAL meets all CCTV requirements robustly.

Both versions execute identically until a power outage (at frame 650), where the GOAL version performs meta-adaptation to start reducing energy. In contrast, the Application specific version cannot change the goal, continues using high energy, and risks depleting the battery, rendering the camera inoperable. After frame 650, GOAL consumes 32% less energy than the Application-specific approach. An additional AdaptLog could have been synthesized to run the CCTV with minimal energy. However, such a version would needlessly sacrifice quality while operating on line power, and switching between independent AdaptLogs at runtime is quite costly—in both execution time Section 3.6.3 and engineering effort Section 3.6.4. This example shows the importance of providing support for general adaptation and meta-adaptation and also the ease with which this can be achieved using GOAL.

3.3 The GOAL Runtime

Figure 3.4 illustrates GOAL's runtime, which consists of three key pieces: (1) the virtualized AdaptLog, (2) the AdaptSpec Compiler, and (3) the Model Builder. The Glue code coordinates these three components with the rest of the software and hardware. Furthermore, some application and system level metrics such as throughput and power consumption are measured by the Glue code [111]. The AdaptLog (Section 3.3.1) is implemented independently of any specific AdaptSpec; it receives the goals, the model, and current metrics and produces a *schedule* of configurations. The compiler (Section 3.3.2) takes an AdaptSpec and produces four outputs: (1) the space of allowable knob configurations, (2) metrics to monitor, (3) any constraints on knobs and (4) the goal to meet. The goal is defined as a *constrained optimization problem* (COP) in GOAL's DSL (Section 3.4.3). Before deployment, the model

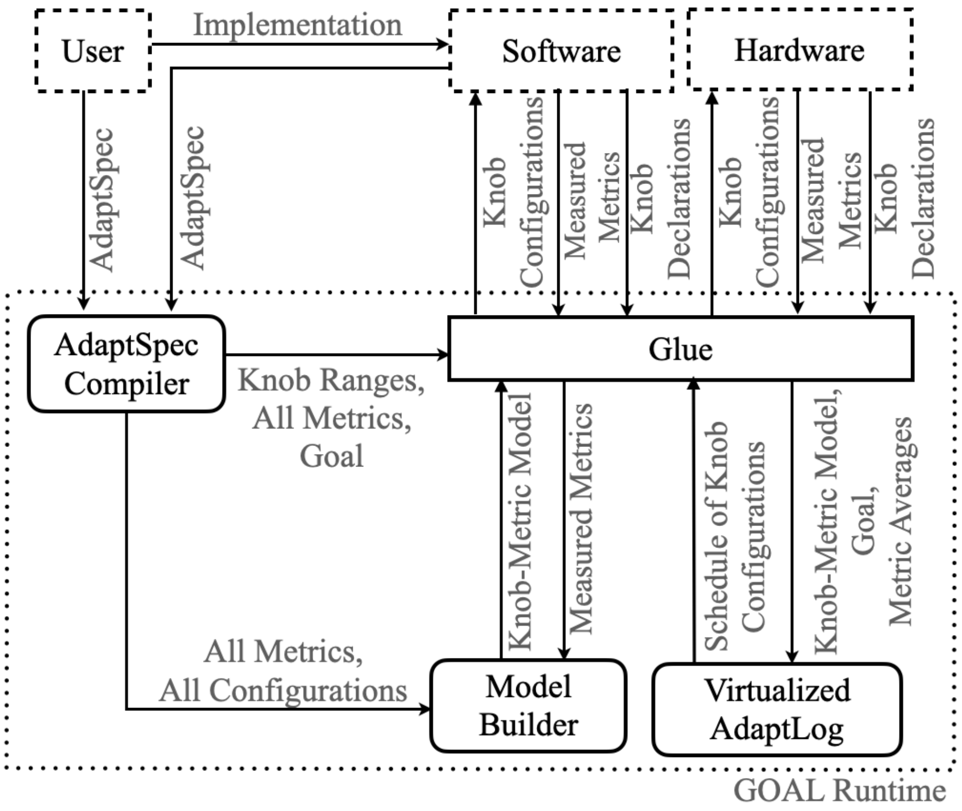


Figure 3.4: The GOAL Runtime.

builder (Section 3.3.3) samples configurations from the compiled AdaptSpec and executes the system in those configurations while observing the metrics listed in the AdaptSpec. From these observations the model builder learns a model that estimates changes in metrics as a function of knob configuration. During deployment, the AdaptLog schedules knob configurations to optimally meet the goal according to the model's estimates.

3.3.1 A Virtualized Adaptation Logic

GOAL virtualizes a control theoretic adaptation logic using two key principles: *relativity* and *translation*. While typical control systems find absolute values for the knobs they configure, GOAL controls virtual values that represent the necessary change in behavior—*relative* to the default knob configuration—that must be achieved to meet the goals. This virtual value by itself is not useful; it must be *translated* into a schedule of specific knob configurations. Critically, this translation can be done online once the metrics and knobs are known and the Model Builder (Section 3.3.3) has produced the function that estimates metrics given knob configurations.

A Traditional Control Example Filieri et al. propose a framework for automatically synthesizing controllers for software systems [76]. We use this approach to illustrate a typical way that AdaptLogs are created, specifically building a controller for the `qp` knob from the CCTV example (Section 3.2). We begin assuming the original goal: meeting a frame rate constraint with maximum quality. Following [76], we first learn a simple linear model, relating frame rate to `qp`:

$$\mathrm{FPS}(t) = 0.354 \cdot \mathrm{qp}(t-1) \tag{3.1}$$

The constant 0.354 is specific to *qp*. To meet a target performance, the controller monitors the current performance $\mathrm{FPS}(t)$ at time t and computes the error with the desired performance FPS_{goal}: $e(t) = \mathrm{FPS}_{goal} - \mathrm{FPS}(t)$. With this error and the model from Equation 3.1, we set `qp` at time t as:

$$\mathrm{qp}(t) = \mathrm{qp}(t-1) - \frac{e(t-1)}{0.354} \tag{3.2}$$

Equation 3.2 is a simple and efficient AdaptLog, suitable for `qp`. However, this AdaptLog is not general; it is entirely specific to (1) the constraint metric (frames per second in this example) and (2) the available knobs (`qp`).

While this example uses a control approach, similar problems occur with other techniques. For example, reinforcement learning (RL) is a popular basis for AdaptLogs [50, 195, 155, 150, 201]. However, RL requires a reward function and a set of possible actions; in existing frameworks, the reward is tied to specific metrics, while the actions are tied to sets of specific knobs (e.g., application alternatives for Aeneas [50]). We are not aware of a way to virtualize the actions in RL based AdaptLogs that is both useful for optimization and general with respect to user-defined knobs.

GOAL's Virtualized Control Logic We now show how subtle changes in the above formulation implement a virtual control signal. The key here is to model *relative* behavior rather than absolute metrics as in typical control approaches. We then *translate* that relative behavior into specific knob settings. This formulation provides a layer of indirection. The controller (which only understands relative values) can be implemented independently of any specific knobs or metrics. The additional logic for translating a relative value into specific knobs settings is parameterized by the learned model that enables the translation (Section 3.3.1).

We begin by noting that a simple equation relates the behavior in any metric m

at time t to a scalar multiple $xup(t-1)$ of some baseline behavior m_{base}:

$$m(t) = m_{base} \cdot xup(t-1) \tag{3.3}$$

Given this relationship, to control metric m, we first compute the error between the target behavior (m_{target}, the constraint from the AdaptSpec) and the measured behavior ($m(t)$):

$$e_m(t) = m_{target} - m(t) \tag{3.4}$$

We can then control the behavior by tuning the $xup(t)$:

$$xup(t) = xup(t-1) - \frac{e_m(t-1)}{m_{base}} \tag{3.5}$$

Equation 3.5 looks similar to Equation 3.2, but instead of a constant appropriate for one knob, Equation 3.5 is parameterized by the base behavior for this metric; i.e., the metric's expected value in the default knob configuration.

While independent of any specific knobs, this approach is clearly dependent on the base behavior m_{base} for the application in metric m; i.e., the expected metric value when the application's knobs are all set to their default values. This value will obviously vary from application to application and even as the application runs.

Adapting to Workloads To adapt the controller to the current behavior in metric m at runtime, GOAL's AdaptLog continually estimates m_{base}, using a Kalman Filter [206], an approach used in prior work [121, 120]. Thus, if the behavior varies during execution, this estimation compensates and ensures that the constrained metrics can still be controlled. This process is analogous to approximating a nonlinear function (in this case the application's behavior with respect to execution time) with a series of tangent lines, where m_{base} is the tangent's slope. More formally, GOAL's Kalman filter estimates the base behavior at time t as $m_{base}(t)$ and uses this estimate in place of m_{base} in Equation 3.5. GOAL uses a standard Kalman filter formulation:

$$\begin{aligned}
m_{base}^{-}(t) &= m_{base}(t-1) \\
e_b^{-}(t) &= e_b(t-1) + q_b(t) \\
k_b(t) &= \frac{e_b^{-}(t) \cdot xup(t)}{xup(t)^2 \cdot e_b^{-}(t)} \\
m_{base}(t) &= m_{base}^{-}(t) + k_b(t)(\frac{1}{m(t)} - s(t) \cdot m_{base}^{-}(t)) \\
e_b(t) &= [1 - k_b(t) \cdot xup(t-1)]e_b^{-}(t)
\end{aligned} \tag{3.6}$$

In this formulation, $k_b(t)$ is the Kalman gain for the constrained metric, m, being controlled. The $m(t)$ denotes measured behavior of the constrained metric during the last window. The $m_{base}^{-}(t)$ and $m_{base}(t)$ are the next to last and last estimates of m_{base}. Similarly, $e_b^{-}(t)$ and $e_b(t)$ are the next to last and last estimates of the error variance.

The Kalman Filter is useful because it provides optimal estimates of the application workload and is exponentially convergent [51]; i.e, the estimate will converge in a number of iterations proportional to the logarithm of its error. Furthermore, the user or the developer does not need to provide any additional data, these guarantees are provided using data that is available to GOAL during execution.

Scheduling Knob Configurations Even after accounting for workload changes, the virtual *xup* from Equation 3.5 is not useful by itself; it must be *translated* into actual knob configurations. A set of values for knobs can be represented as a vector k, and a knob configuration is an assignment of a value to each knob component in the vector. For example, our video encoder from Figure 3.1 has a knob vector with four components (one each for `qp`, `me`, `numCores` and `coreFreq`), and the default configuration is $\langle 30, 1, 1200, 4 \rangle$. Each knob configuration has an expected effect on each metric, as estimated by GOAL's model builder. In our example those metrics are throughput (frame rate), quality, and energy. In addition, the *xup* signals are continuous while the available knob settings are discrete.

We translate *xup* from Equation 3.5 to discrete knob configurations by *scheduling* over time; i.e., spending different amounts of time in knob configurations such that the average over the time period is the desired continuous value. This scheduling problem is formulated as a constrained optimization problem (COP, which is extracted from the AdaptSpec by GOAL's compiler) where the *xup* value is the constraint to be met and the decision variables are the time to spend in the knob configuration vectors:

$$\begin{aligned}
\text{optimize} \sum_{T} F(k) \cdot T_k & & (3.7)\\
\text{s.t.} \quad xup(t) &= \frac{1}{T} \cdot \sum_{k} x\hat{u}p_k \cdot T_k & (3.8)\\
T &= \sum_{k} T_k & (3.9)\\
T_k &\geq 0, \forall k & (3.10)
\end{aligned}$$

Here T is the time window (defaults to 40 but can be set using an environment variable) over which to schedule, T_k is the time to spend in the kth knob configuration, and $x\hat{u}p_k$ is the expected xup for k (from GOAL's model builder Section 3.3.3). $F(k)$ is an objective function over knobs defined in the AdaptSpec and extracted by the compiler. For example, in the CCTV application (Section 3.2), the objective on line power is to maximize quality. Equation 3.8 requires that the average of all configurations' predicted $x\hat{u}p_k$ values come out to the desired $xup(t)$ value, while Equation 3.9 requires that the sum of times spent in each configuration is equal to the total time over which we are scheduling. Equation 3.10 ensures that there are no negative time values. The controller sets the virtual $xup(t)$ to ensure the AdaptSpec's constraint is met, and this optimization problem ensures that virtual signal is translated into specific knob settings to deliver the desired xup.

Handling Non-Linear Behavior GOAL's AdaptLog uses a linear control model (Equation 3.5) and solves a linear optimization problem (Equations 3.7–3.10). Of course, most computer systems will exhibit non-linear behavior, especially when combining knobs. Thus, we discuss why this formulation suffices to handle non-linearities. Note the adaptive control (Section 3.3.1) approximates non-linear shifts in application behavior over time. We therefore focus on non-linear interactions in knob behavior. For example, the performance of a knob configuration is likely a non-linear function of each component knob in the configuration. The key intuition is that Equations 3.7–3.10 transform a non-linear optimization over the space of all knobs into a linear problem over an exponential search space. Fortunately, however, the problem structure makes it practical to solve.

GOAL's model builder estimates the $x\hat{u}p_k$ values for each knob configuration. Given that k here is a vector of knob configurations, the total size of the configurations space is the cross product of all allowable knob settings. So, while the optimization problem in the previous section is linear, there is a nonlinear number of decision variables (the times to spend in each configuration).

However, there are only two non-trivial constraints (Equations 3.8 and 3.9). By the duality of optimization problems there is an optimal solution where exactly two of the configurations are allocated non-zero time [47]. Furthermore, those two configurations correspond to two configurations on the convex hull of the tradeoff space represented by the values of F and xup. For example, if the goal is to meet a performance constraint (xup) and optimize accuracy (F), then the optimal solution will involve two configurations on the optimal frontier of performance and accuracy; specifically, the two configurations whose estimated $x\hat{u}p_k$ values are just below and just above the target $xup(t)$ [47]. Those two configurations can easily be found from a lookup table, so GOAL's Glue takes the model from the model builder and the AdaptSpec from the compiler and forms a lookup table that considers only the Pareto-optimal tradeoffs in the objective and constraint metrics. Sorting into the Pareto-optimal points can still be expensive (O($|k|\ \log(|k|)$), but is only done when a new AdaptSpec is made available. We evaluate the practical overhead in Section 3.6.3 and Section 3.6.8.

Control Theoretic Formal Properties The most important guarantee is that the system converges to the constraint. Unlike AdaptLogs based on learning methods such as RL or Bayesian Optimization which do not provide guarantees of convergence to constraint, GOAL's AdaptLog uses adaptive control. It thus inherits the formal control theoretic guarantees of a typical control system [96]. GOAL's AdaptLog will converge provided that the estimated base behavior $m_{base}(t)$ and the predicted speedups $x\hat{u}p_k$ are within a factor of 2 of their true values. This analysis is based on straightforward application of control theory [76]. Furthermore, even if the base behavior estimation is incorrect momentarily, the Kalman filter is exponentially convergent in error, meaning that estimate will be corrected in a logarithmic number of iterations. We evaluate GOAL's empirical error tolerance (Section 3.6.6) and find that GOAL's practical behavior matches the theoretical analysis: GOAL tolerates extremely large over-estimations of xup and tolerates under-estimations up to a factor of 0.55; e.g., predicting 16.5 frames per second when the true behavior is 30. This robustness is yet another reason we favor a control theoretic formulation for GOAL's AdaptLog and it is consistent with prior work that shows adaptive control techniques do a better job of meeting operating constraints than learning-based techniques [146].

Connecting with the Rest of GOAL GOAL's virtualized adaptation logic requires the following parameters to be supplied by the rest of the GOAL runtime:

- **Target behavior m_{target}:** Users specify this value as a constraint in their AdaptSpec and it is extracted by the JIT compiler (Section 3.3.2).
- **Objective function F:** This too is provided by the user in the AdaptSpec, extracted by the compiler.
- **Expected behavior for target metrics ($x\hat{u}p_k$):** These values are stored in lookup tables, dynamically constructed by the runtime using the model learned using the model builder (Section 3.3.3).
- **Schedule window (T):** This value is the `window` parameter to GOAL's `optimize` method. T is provided to the AdaptLog at initialization.
- **Measured Behavior ($m(t)$):** These measurements are provided by the runtime.

This parameterized AdaptLog is general and dynamic. It is general because it works with any knobs and measures that can be specified using GOAL. It is dynamic because the runtime can rapidly change the AdaptLog by simply passing in new parameters.

3.3.2 Adaptation Specification Compiler

The GOAL compiler extracts the following from the AdaptSpec: (1) a list of metrics, (2) a list of all knob configurations, (3) knob constraints and (4) the goal, which includes m, m_{target} and F (Section 3.3.1). All values except for F are used to initialize the GOAL runtime and interpret the model learned by the model builder.

However, F needs to be evaluated repeatedly when computing schedules as it is the objective function to optimize. F cannot be precomputed because it might be an arithmetic expression over several metrics. Therefore, we adopt a two-step evaluation approach.

In the first step, all aforementioned objects from the AdaptSpec's abstract syntax tree are translated to Swift objects using the technique proposed by Carrette et al [52]. All configurations from the extracted list are converted into *knob configurations objects*. A *knob configuration* is an object with an `apply` method, that sets the application and system level knobs to their corresponding values. These objects correspond to the k from Equations 3.7–3.10. The m and knob constraints are used to make a lookup table containing the $x\hat{u}p_k$ and a configuration object for configurations, k, that meet all knob constraints. F is translated into a Swift closure so that it can be executed by the AdaptLog with low overhead while computing schedules. In the second step, the AdaptLog continually executes the aforementioned closure—once the observed values for each `measure` are available—to evaluate F as it produces the knob schedule. During execution, a GOAL application can modify the AdaptSpec which triggers recompilation (starting over from the first step).

3.3.3 Model Builder

GOAL's model builder operates in two modes. In the first—pre-deployment—it runs user-specified test workloads and measures how changes in knobs cause changes in metrics. Specifically, it executes a sampled subset of all knob configurations and estimates the metrics for all allowable knob configurations. The model builder then learns the model using this collected data by using linear regression to compute a piecewise linear model representing the expected behavior for a metric given a knob configuration. In the second mode—during deployment—the model builder interprets and uses the learned model to predict the impact of particular knob configurations on all metrics. Prior work could be used to replace our learner with more accurate or efficient learners [69, 63, 62]. However, our empirical evaluation shows that GOAL's control system is robust to substantial errors in profiling (Section 3.6.6), so we use piecewise linear models as a proof of concept and leave the investigation of more advanced learning methods to future work.

3.3.4 Limitations

While GOAL provides significant advantages over prior work, there are several aspects that deserve further discussion.

Foremost, GOAL can only manage goals that can be expressed in terms of quantifiable measures that can be used by the adaptation logic to make adaptation decisions.

GOAL's AdaptLog handles non-linear, non-convex optimization problems by exploiting problem structure and Pareto-optimality to schedule combinations of knob configurations (Section 3.3.1). This approach works because there are a small number of possible constraints. Essentially, we have transformed a problem that would be exponential in the space of possible knob configurations into a problem that is exponential in the number of constrained metrics. In practice, we believe this is a reasonable tradeoff because there are typically many knobs that affect one constraint and there is usually only a small number of metrics for which there is required behavior. Thus, the number of constraints will be much, much smaller than the number of possible knob configurations.

GOAL actuates hardware-specific knobs on the system's behalf. While it is easy to manipulate some hardware platform specific knobs (e.g., the core usage), others (e.g., DVFS frequency) require special permissions. Hence, GOAL systems may require elevated privileges to manipulate certain knobs. Section 3.6.5 shows that system and application knobs can be split into multiple modules, however.

As a proof of concept, GOAL's semantics currently only support discrete knobs. Interesting future work would be assessing the benefits of continuous knobs.

3.4 The GOAL Programming Interface

We implement GOAL as an extension to Apple's Swift [30]. Thus, GOAL is statically typed and memory-safe with predictable performance thanks to reference counting storage management and a strict evaluation strategy. GOAL consists of ∼9.5K lines of code written in a combination of Swift and C, not including third party libraries.

3.4.1 The GOAL *Library API*

The GOAL library API provides a type (`Knob`), and two core functions (`measure` and `optimize`). Figure 3.1b shows the adaptive version of the application that makes use of the GOAL programming interface

GOAL provides three additional functions: `restrict`, `control` and `intend`. These allow programmers to interact with the runtime from the core application. `restrict` limits the range of allowable knob values. `control` returns the knobs to their original range. `intend` updates the entire adaptation specification. We discuss the benefits and uses of `control/restrict` functions in Sections A and 3.6.

The Knob Type From the user's perspective, the type `Knob<T>` is a type-safe immutable cell that replaces a variable in the base program, and enables the runtime to adapt the cell's value. In our example, `qp`, `me`, `rf` become knobs and their original values initialize the `Knob` instances. The *reference values* represent the knob's default values and make it possible to compile and execute a GOAL application as a normal Swift program, with the original semantics. A `Knob`'s is accessible with a `get()` method.

While the GOAL runtime will generally be responsible for setting the knobs, the `Knob` type can be manipulated directly from within the core application using: `restrict` and `control`. `restrict` explicitly defines a range of values for a particular `Knob`. The runtime uses this range to constrain the configuration space available for adaptation. Calling the method without any arguments fixes the `Knob` to the value it had at the time of the method call. The `control` method removes any limits from previous calls to `restrict`. Note that these functions are really semantic sugar for `intend` and are provided because many changes to the adaptation specifications simply change the knobs' allowable settings.

The Measure Function The `measure(name,value)` API reports a `value` of the measure `name`, providing a view of the application's current quantifiable behavior.

Some measures are explicitly reported through the `measure` API and others are implicitly recorded by the runtime. The implicit measures include performance, latency, and system-level measures such as total energy, energy per iteration, and power consumption. The runtime expects the measures declared in the adaptation specification to be reported during execution. If a measure is not reported the application terminates gracefully and provides an error message including the name of the missing measure. Any extra measures reported by the application that are not a part of the active AdaptSpec are tracked but not used for adaptation.

The developer may dynamically introduce new measures dynamically. However, the developer must also provide a corresponding model (Section 3.3.3). The runtime then starts tracking the new measure as it would any other.

Optimize Loop `optimize` takes as parameters: a list of knobs that the runtime should use for control, a routine (block of code) that should be monitored to provide feedback to the adaptation backend, and an optional *window size* over which measure statistics are computed. Conceptually the `optimize` function replaces the part of code that is repeatedly executed by the application and needs adaptation. In our example application the `optimize` function replaces the `while` loop over frames. Thus, `optimize` captures the adaptive loop that is common to any application that implements the OODA or MAPE patterns.

The window size segments the sequence of iterations into conceptual blocks. For a sufficiently small window size, the application behavior is assumed to be sufficiently uniform that average measure values of the previous window are representative of the iterations of the next window. On the other hand, the window size should be large enough that the windowed statistics filter out transients, caused by rare and unsystematic events. Thus, the window size is typically identified by the application developer, who will be a domain expert, familiar with the expected behavior of the application under different conditions.

GOAL is not tied to a single control structure. For brevity, we omit full of discussion an alternative design for `optimize` that can be used with recursive and asynchronous codes. In this version, `optimize` is a method invoked by the core application at the completion of a ‘step’ of computation upon which the runtime needs to modify the values of the `Knob`s to continue to meet the adaptation goal for the next ‘step,’ where a step is just the time between recursive calls or asynchronous events.

The Intend Method `intend` changes the adaptation specification at runtime. The method takes as arguments the type of optimization that needs to be performed, a string representation of the objective function, and a string representation of the constraint expression. While all GOAL applications start with an initial adaptation specification, `intend` overwrites any previous specification and the runtime immediately adjusts to meet the new AdaptSpec.

3.4.2 Supporting Multi-Module Adaptation

In large computing systems, multiple developers might want to independently develop adaptive modules and GOAL supports this by allowing multiple `optimize` calls. However, the sets of knobs used by different `optimize` calls have to be disjoint and the GOAL runtime will throw an error if this is not the case. After initialization, the runtime manages execution and actuates knobs of each `optimize` independently.

During execution, one module might modify a knob that affects a metric monitored by another module. For example, a module might meet a power goal by lowering clock frequency and thus reduce throughput in a different module. The base speed of the latter module will thus be underestimated. However, GOAL's adaptive control (Section 3.3.1) will observe and account for this change. In this example, the runtime's base speed estimation will update to account for the lowered frequency. GOAL will meet goals of all modules if they are noncompeting (i.e., GOAL cannot constrain system power in one module and minimize it in another). Supporting competing goals is an open research problem in adaptive computing. Having language support could make this problem easier and we leave that to future work.

3.4.3 Adaptation Specification Language

Figure 3.5 shows the adaptation specification language's grammar. $\{\cdot\}$ and $\langle\cdot\rangle$ denote a set and a sequence of elements, respectively. The arity of a function f is denoted by $|f|$. The set of names of knobs declared in an AdaptSpec $s \in \mathsf{Spec}$ is denoted by K_s. The set of names of `measure`s declared in s is denoted by M_s. Figure 3.1c is an example AdaptSpec for the encoder in Figure 3.1b. An AdaptSpec consists of a

Name	**Platform**	**# App. Knobs**	**# Sys. Knobs**	**Initial Objective**	**Initial Constraint**	**Change after Meta-adaptation**
CCTV Camera [100]	Embedded	3	2	max(qual)	tput == 30	min(pwr)
Video Object Detector [145]	Embedded	4	2	min(pwr)	tput == 20	Restrict QP
Service Oriented Architecture (SOA) [78]	Server	3	N/A	max(rel)	latency == 0.5	min(cost), rel == 0.6
Synthetic Aperture Radar (SAR) [188]	Embedded	4	2	max(qual)	tput == 80	max(tput), qual == 0.7
AES Encryption [89]	Embedded	1	2	max(tput)	pwr == 1.5	max(tput/pwr),bs == 256
Search Engine [33, 103]	Server	1	2	min(pwr)	tput == 18.0	Restrict searched doc
Optical Character Recognition (OCR) [19]	Server	1	2	min(pwr)	tput == 8.0	max(qual), tput == 8.0

Quality, throughput, pwr consumption, reliability, block strength
have been abbreviated in the table to qual, tput, pwr, rel and bs respectively.

Table 3.2: Properties of applications used to evaluate GOAL.

$$
\begin{array}{rclcl}
s & \in & \mathsf{Spec} & ::= & \texttt{goal}\ \ o(e)\ [\ \texttt{such that}\ m\ \texttt{==}\ c\ ] \\
 & & & & \texttt{measures}\ \{\ m_i : t_i\ \}^{1 \le i \le |M_s|} \\
 & & & & \texttt{knobs}\ \{k_i\ \texttt{=}\ e_i\ [\texttt{reference}\ c_i]\ \}^{1 \le i \le |K_s|} \\
 & & & & \quad [\ \texttt{such that}\ e\ ] \\
o & \in & \mathsf{Opt} & ::= & \texttt{min}|\texttt{max} \\
e & \in & \mathsf{Expr} & ::= & c|m|f\langle e_i \rangle^{1 \le i \le |f|} \\
c & \in & \mathsf{C} & ::= & \textit{constants} \\
t & \in & \mathsf{T} & ::= & \textit{type names} \\
m & \in & \mathsf{M} & ::= & \textit{metric names} \\
k & \in & \mathsf{K} & ::= & \textit{knob names} \\
f & \in & \mathsf{F} & ::= & \textit{function names}
\end{array}
$$

Figure 3.5: The GOAL Adaptation Specification Language

COP, the metrics to monitor and the knob along with their valid values that can be used to meet the COP.

Goal This section encodes a constrained optimization problem (COP), expressing correct application behavior in terms of its measures. The goal has five parts:

- The *optimization type* ($o \in \mathsf{Opt}$), either `min` or `max`.
- The *objective* ($e \in \mathsf{Expr}$), an expression on `measures`.
- The *constraint metric* ($m \in \mathsf{M}$), a metric (`latency` in our example) that is associated with constraints.
- The *constraint target* ($c \in \mathsf{C}$), the constraint value (`30.0` seconds per iteration in our example).

Measures This section declares quantifiable metrics like latency, bitrate, etc., that should be observed by the runtime. Currently, metrics may only have the `Double` type, but any totally ordered type that supports the operations used in the objective function could work.

Knobs This section defines the available *configuration space*. The `knob` definitions consist of a name, a *range* expression that evaluates to a list of constants, and a reference value. The name associates the knob definition with a `Knob` instance in the application. The list of `Knobs` in the AdaptSpec can be a subset of the `Knobs` from the code. The runtime will only use the `Knobs` defined in the AdaptSpec during execution. For all the `Knobs` not defined in the AdaptSpec, the runtime will only use the reference value in the code. Reference values in the AdaptSpec override those passed to the knob constructor. Optionally, developers can define a *knob constraint*: an arbitrary Boolean expression over the knobs. GOAL's runtime then filters out any configurations that violate these knob constraints before passing the model to the AdaptLog.

3.5 Evaluation Methodology

We evaluate the following:

- **Generality**: Can GOAL support a wide range of metrics and knobs while meeting goals as well as prior approaches designed for specific goals and knobs?
- **Dynamism**: When meta-adaptation is performed, does GOAL converge to the new AdaptSpec while providing near-optimal behavior for the objective function?
- **Robustness**: Does GOAL reliably meet goals even in the face of multi-module adaptation, errors in the learned model, and time varying workloads?

This section details the applications, platforms, points of comparison, and metrics used to evaluate these properties.

3.5.1 Applications and Platforms Evaluated

We implement 7 adaptive applications from prior work and then augment them to perform meta-adaptation. We start each with an AdaptSpec equivalent to one used in existing literature. At runtime, however, we change the goals or knobs. Table 3.2 shows the details, including the number of application and system knobs, the initial objective and constraint, and the required meta-adaptation. The applications cover a wide range of system and application knobs, metrics, and goals; and they exhibit nonlinear knob interactions and thus are difficult to model and control [218, 71, 67]. Consider the Synthetic Aperture Radar (SAR) application. Figure 3.6 shows SAR's throughput (represented by color) as a function of the application (y-axis) and system (x-axis) knobs. The figure shows the non-linear relationship with many local optima. All other example applications have similar knob interactions. Hence, optimizing such applications requires escaping local optima and accounting for non-linearities.

To further demonstrate generality, we use two system platforms with distinct knobs. Four applications target an embedded system: an ARMv7 based ODROID-XU3 (Exynos5422) with 2GB of RAM, running Ubuntu 16.04 (GNU/ Linux 3.10). Three target a server: an x86 (Intel i7-6700) with 8GB of RAM, running Ubuntu 18.04 (GNU/Linux 5.30). All use multi-threading equal to the core count.

3.5.2 Adaptation Approaches Compared

We compare GOAL to prior approaches including:

- **Linear Control**: This approach uses a single linear control model for all

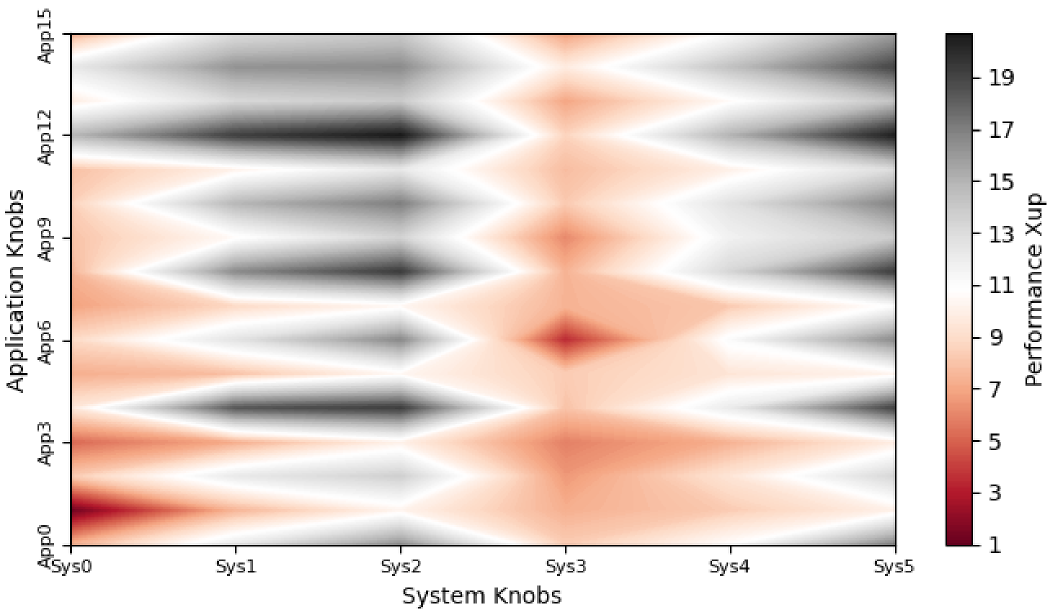


Figure 3.6: SAR's throughput is non-linear with local optima.

applications [96]. This comparison is illustrate GOAL's ability to cope with profiling errors and shifts in application workload.

- **Application-specific AdaptLogs**: Using prior work, we synthesize a specialized AdaptLog for each application's specific AdaptSpec [76]. This approach can meet each application's initial AdaptSpec, but not the new one created through meta-adaptation. This comparison is performed to prove that GOAL is able to meet goals as well as any application specific AdaptLog.

- **Multiple Application-specific AdaptLogs**: We synthesize a specialized AdaptLog for each application's AdaptSpecs both before and after meta-adaptation. At runtime, we perform meta-adaptation by shutting down the first AdaptLog and initializing the second. However, it should be noted that this approach is clearly impractical as users must know the new goals and knobs ahead of time

- **Oracle:** We exhaustively search the configuration space to find the best knob configuration for all AdaptSpecs.

3.5.3 Evaluation Metrics

We quantify the approaches using the following metrics:

- **Mean Absolute Percentage Error (MAPE)**: Each application's AdaptSpec has a constraint for a particular metric. To calculate MAPE, at each call to `optimize()` we measure the absolute error between the constraint and the achieved metric, then take the mean over all iterations. Lower is better.

- **Normalized Performance**: The AdaptSpecs also specify an objective, or a metric to be optimized. To measure how well GOAL optimizes the objectives we normalize to the Oracle's performance. Higher is better as the oracle will achieve an optimum performance of 1.

- **Iterations until Convergence**: We measure the iterations required to converge to the goal after meta-adaptation. This metric will be used to compare GOAL to the approach that pre-synthesizes multiple, application specific AdaptLogs. Lower is better as it represents faster convergence after meta-adaptation.

3.6 Evaluation And Observations

This section evaluates key aspects of GOAL, specifically:

1. Does GOAL meet user-defined constraints across a range of knobs and metrics?
2. Does GOAL achieve near-optimal objectives when performing meta-adaptation?
3. Does GOAL converge quickly after meta-adaptation?
4. How much user effort does GOAL require?
5. Does GOAL support multi-module adaptation?
6. Is GOAL robust to errors in modeling?
7. Is GOAL robust to large changes in workload?
8. How much overhead does GOAL incur?

These aspects justify GOAL's generality, dynamism and robustness. In Section A, we discuss each case study in detail.

3.6.1 Does GOAL *meet user constraints?*

For each, we measure the MAPE for the application-specified constraints (as listed in Table 3.2). Figure 3.7 shows the results with MAPE on the vertical axis, applications on the horizontal axis, and a bar for each AdaptLog. Due to the complicated and varied application behavior, the Linear Controller fails to reliably meet the constraint exhibiting a mean MAPE of ∼29%. However, both GOAL and Application-specific

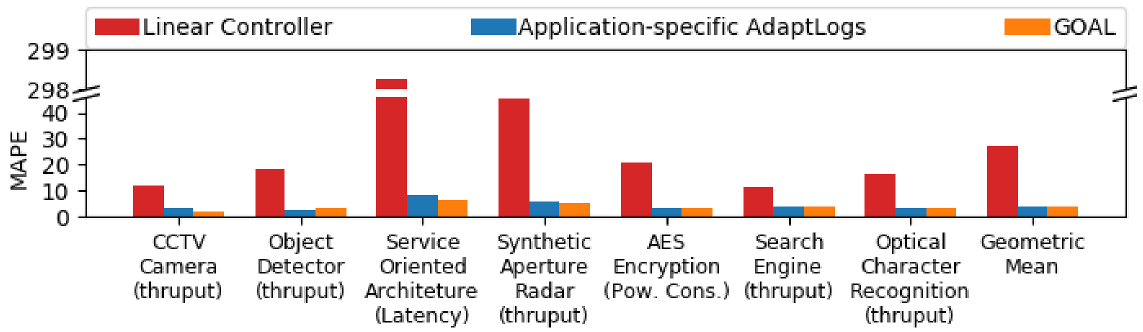


Figure 3.7: GOAL reliably meets the constraint of adaptation.

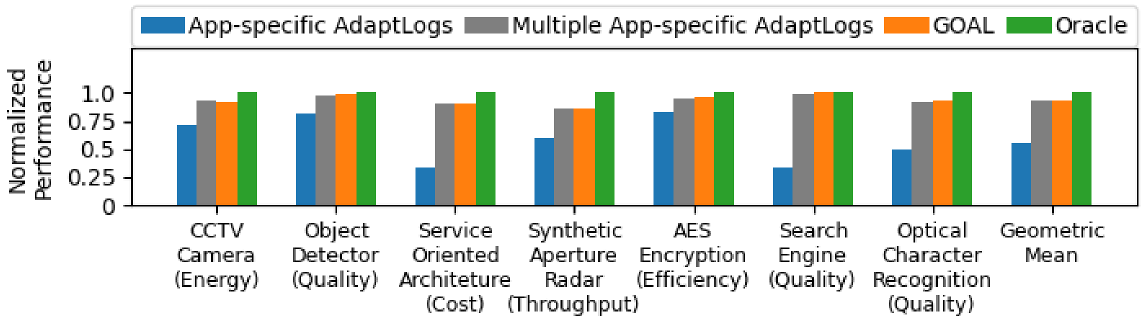


Figure 3.8: GOAL is optimal when meta-adaptation is needed.

AdaptLogs meet the goal reliably, exhibiting less that 4% MAPE. We note that the results for the Application-specific AdaptLogs and Multiple Application-specific AdaptLogs are the same for this experiment, so we omit the latter for clarity. *In summary,* GOAL *provides a single AdaptLog implementation that generalizes across a range of metrics and knobs while achieving errors comparable to approaches that do not generalize.*

3.6.2 Does GOAL *optimize objectives?*

We now consider the normalized performance when each application performs meta-adaptation, changing its AdaptSpecs as listed in Table 3.2. We report results for application-specific AdaptLogs, Multiple application-specific AdaptLogs, GOAL, and the Oracle. We omit the Linear Controller results because it fails to meet the constraints (as shown in the previous section), so its performance is not meaningful.

Figure 3.8 shows the results. The single Application-specific AdaptLog has poor performance because it must use the initial AdaptSpec throughout execution. However, the versions using Multiple application-specific AdaptLogs and GOAL improve on relevant metrics by 1.69× over application specific AdaptLogs, while achieving ∼93.7% of the Oracle's performance.

These results show that when an application's requirements change, the application performs suboptimally with prior work that does not support meta-adaptation. GOAL, however, performs as well as an approach that knows how the requirements will change ahead of time and synthesizes multiple application-specific AdaptLogs for both requirements. *This demonstrates* GOAL*'s support for dynamic changes in adaptation specifications, as it performs near optimally when applications trigger meta-adaptation.*

3.6.3 How quickly does GOAL *converge?*

The previous section shows that GOAL's average performance is equivalent to an approach that is custom built for a specific AdaptSpec. We now show that GOAL provides an additional benefit by measuring how quickly each approach (Multiple

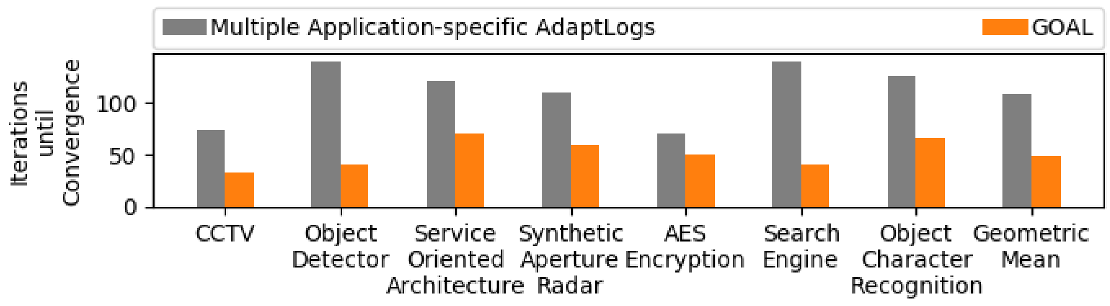


Figure 3.9: GOAL's runtime allows rapid convergence after meta-adaptation.

Application-specific AdaptLogs and GOAL) reconverges to the new constraint after meta-adaptation.

Figure 3.9 shows the results with iterations required for convergence on the vertical axis. GOAL converges ∼2.14× faster than using multiple application-specific AdaptLogs. The application-specific approach takes longer due to the expense of destroying the old AdaptLog and initializing the new one. This process incurs two sources of overhead: (1) the time required to perform these costly operations and (2) a loss of accumulated history of observed metric values, which must be collected by the new AdaptLog from scratch. Such history is crucial for any control or learning based AdaptLog and without it, convergence is delayed. This ability to retain history across multiple metrics and knobs is an advantage to a single, general adaptation framework like GOAL. GOAL converges quicker to the new goals because the runtime tracks all relevant metrics at all times, ensuring that no data is lost when performing meta-adaptation. *These results support the claim of dynamism by showing that* GOAL *converges much quicker than combining existing AdaptLogs without explicit support for meta-adaptation.*

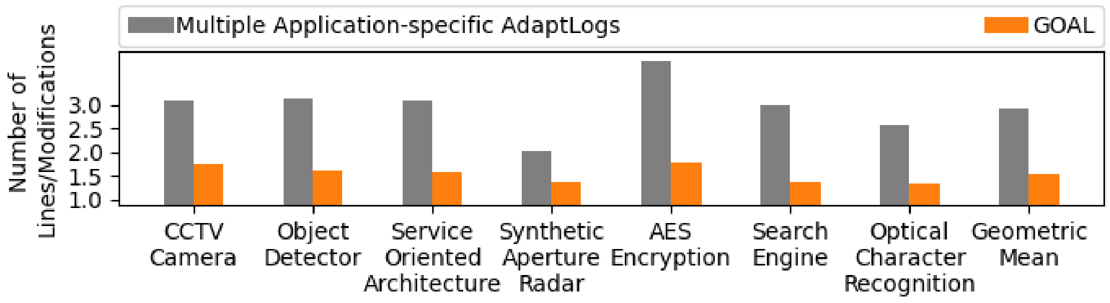


Figure 3.10: GOAL requires minimal development effort for adding adaptation and meta-adaptation.

3.6.4 How much effort does GOAL *require?*

We count the lines of code that need to be added/modified to the original non-adaptive versions of the applications. We do not count lines of code to implement the AdaptLog. Here we only compare GOAL and Multiple Application-specific AdaptLogs because these are the only two approaches which can implement meta-adaptation.

Figure 3.10 shows the number of lines added/modified on the vertical axis. Multiple-application specific AdaptLogs require around 1.92× more changes than GOAL. *These results show that* GOAL*'s interface facilitates adding adaptation in a wide range of applications without significant development effort, and much less than trying to dynamically switch between prior approaches.* Finally, note that GOAL can support any type of meta-adaptation, while prior work takes more effort and yet only supports the AdaptSpecs specific to each application.

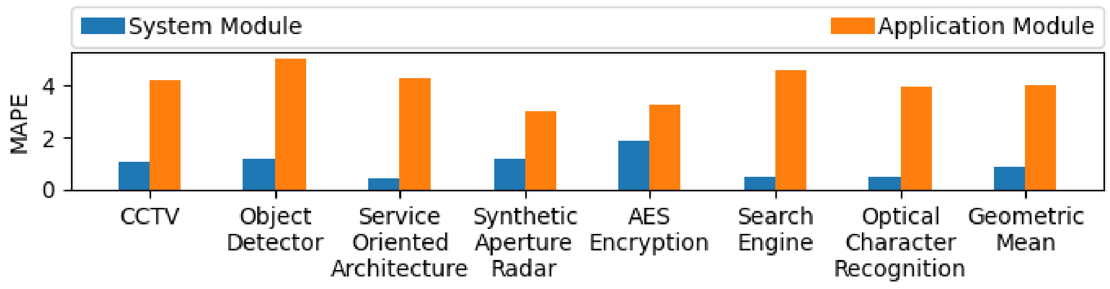


Figure 3.11: GOAL reliably meets goals of multiple modules.

3.6.5 Does GOAL *support multiple modules?*

Results so far have used a single `optimize()` call per application. However, in many scenarios, multiple developers may each want to contribute a module with its own `optimize()` method (as described in Section 3.4.2). Here, we evaluate how well GOAL performs in such multi-module adaptation scenarios.

Specifically, for each application, we implement two modules: one for the application knobs and one for system knobs. The application modules meet the constraints from Table 3.2. The system modules meet a power constraint while delivering maximum performance. The challenge here is that both modules affect performance, meaning that even though the knobs are independent they still affect each other.

We then measure the MAPE for both the application and system modules. Figure 3.11 shows the results. Both the application and system modules meet their constraints reliably, exhibiting a low mean error of ∼4%. *These results show that* GOAL *effectively deals with complex, multi-module adaptation, a capability that other approaches do not support.*

3.6.6 Is GOAL *robust to errors in modeling?*

As mentioned in Section 3.3.3, GOAL development includes modeling the application. We now evaluate how sensitive GOAL is to possible errors in the model learning step. For each application, we introduce errors by scaling all the model's estimates (i.e., the $x\hat{u}p_k$ values used in Equations 3.8–3.10) by an error factor and perform the same experiments as above.

Figure 3.12 shows two charts with MAPE (vertical axis) for each application given a model that is scaled by the value on the horizontal axis; the left chart shows under-estimates, the right chart shows over-estimates. For underestimated models, GOAL performs reliably in the presence of large errors, producing a MAPE that stays relatively low at a value that is roughly equivalent to results from above (Figure 3.7) until scaled down to 0.55. Thus, the model has to be significantly underestimated before GOAL fails to meet its constraints. In contrast, MAPE is not impacted by overestimated models. This suggests that GOAL can support a wide range of learners for model building, but they should be biased slightly towards overestimates. *Overall, these results illustrate that* GOAL *is robust to substantial errors in modeling.*

3.6.7 Is GOAL *robust to changes in workload?*

Adaptive systems, in general, should meet goals despite unpredictable, external disturbances. We now demonstrate that GOAL provides this capability by adapting to different scenes for a video encoder. The first is a high-motion soccer game and the second is a low-motion news cast. Without adaptation—i.e., with a constant knob

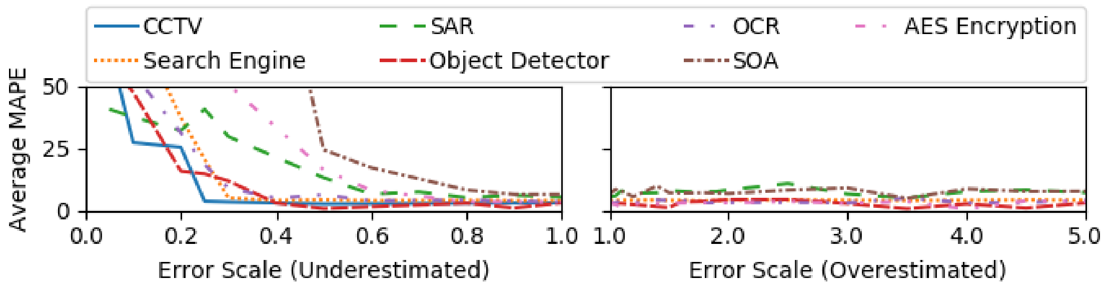


Figure 3.12: GOAL is robust to substantial error in profiling.

configuration—the scene change causes significant changes in throughput because different levels of motion require significantly different amounts of work to encode.

Figure 3.13 shows the power and throughput for three different video encoder implementations: (1) Non-Adaptive, using the fixed, default knob configuration; (2) Linear Controller, similar to the one above, but now we carefully calibrate the control to meet the goal for the initial scene, and (3) GOAL. The different colored regions show two different input scenes. The impact of scene changes can be seen in the execution of Non-adaptive whose throughput and power change from scene to scene. Linear Controller and GOAL both have a throughput constraint of 30.0 frames/s. The changing workload negatively impacts Linear Controller, which, is stable only for the first scene, but then starts to oscillate after the scene change and overall has a high MAPE of 22.5%. However, GOAL copes with these changes and meets the goal reliably during all scenes, exhibiting a much lower MAPE of 2.8%. The Linear Controller fails because the difference in the base throughput (m_{base} from Equation 3.5) is over a factor of 2× different from scene to scene. GOAL can handle this change, however, because it continually estimates the time-varying base behavior to

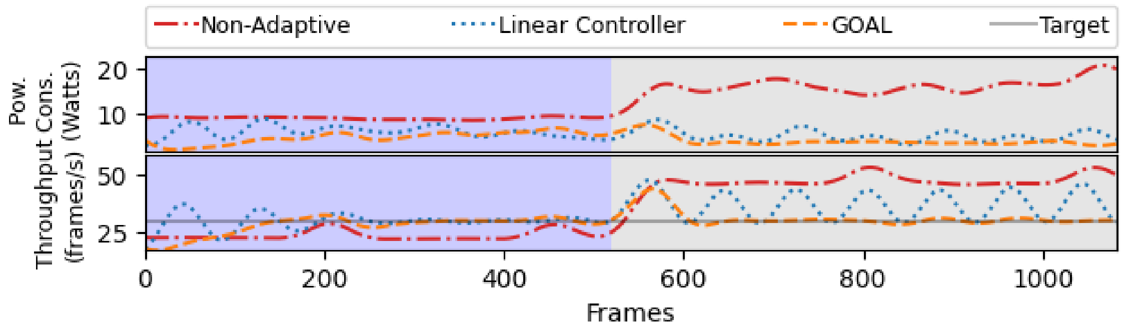


Figure 3.13: GOAL is robust to shifts in workload.

account for these types of changes. *These results further demonstrate the value of* GOAL*'s adaptive control approach that adjusts the control to handle non-linearities in application workload.(Section 3.3.1).*

3.6.8 How much overhead does GOAL *incur?*

We evaluate the overhead of performing adaptation and meta-adaptation in terms of the geometric mean of times required to perform essential operations using GOAL.

Adaptation only requires computing schedules using the AdaptLog (Equations 3.4–3.10). The time to compute a single schedule in the embedded applications is ∼0.68ms, while for server applications it is ∼0.045ms. Obviously, these numbers are highly influenced by the underlying hardware. Schedules are only computed once per window (Section 3.3.1). Hence, the total overhead per iteration becomes negligible.

Meta-adaptation requires calls to `intend`, `restrict` and `control`. The time required for `intend` in the embedded applications is ∼5.14ms, while in the server applications it is ∼1.07ms. The times for calls to `restrict` and `control` in the embedded applications is ∼11.79ms and ∼8.31ms, respectively. These times in the server applications are ∼0.68ms and ∼0.55ms, respectively. This overhead includes the time to recompile the AdaptSpec and compute the new lookup table (Section 3.3.1). GOAL adjusts the goal for the window immediately after meta-adaptation is performed. For all subsequent windows, the goal is set to the value in the AdaptSpecs. This means that, GOAL accounts for its own effect on metrics and ensures constraints are met despite GOAL's own overhead. It should be noted that applications are expected to perform meta-adaptation far less frequently than they compute schedules. *These results show that* GOAL *incurs low overhead for both adaptation and meta-adaptation.*

3.7 Analysis of GOAL-based Programs

Analyzing adaptive software is strictly harder than analyzing non-adaptive software. The difficulty arises due to both the number of dimensions along which the system can adapt and that adaptation happens over time. Thus, extending a programming language with adaptation specifications drastically alters its semantics, and also introduces several classes of bugs. Though a full treatment of verification for adaptive programs in GOAL is beyond the sope of this body of work, the following sections briefly introduce a notion of cost semantics for adaptation inclusive programming, define some simple static analyses that are useful in this context, and describe a testing approach for iterative adaptation inclusive programs.

3.7.1 Cost Semantics

While many different cost semantics have been proposed, they share the property that the cost of an expression is compositional [42, 184]. In other words, an expression's cost is solely determined by the cost of each of its sub-expressions, and the rule that binds them together. If this algebraic system captures the real cost of computation across different machines with respect to a set of parameters, then we obtain a useful abstraction of real computational cost. Most existing cost semantics only model a specific kind of cost; e.g. time or memory. To extend such a semantics to GOAL, we must further abstract away the the details of the computation's resource usage. We call this an *asymptotic cost semantics.* As long as the resource is: (a) not reusable, (b) measurable for every single execution, and (c) its cost is additive, then the cost of a sequence of executions of a single expression can be modeled by a sequence of *i.i.d.* random variables, conditional on the value of the expression's input. Further, the average of this sequence converges to some fixed value with an *i.i.d* sequence of inputs, by the law of large numbers. We call this value the *asymptotic cost* of an expression e with respect to the underlying machine M, the resource r and the distribution of the inputs d and denote it by $C[e; M, r, d]$.

This semantics tries to capture the average (and thus cumulative) cost instead of the cost of each call. Intuitively, this means that the impact of compiler optimizations and external disturbances may be smoothed out in the long run, and the asymptotic cost semantics offers a more robust and manageable way to model the behavior for further tuning, in contrast to call-wise cost semantics and even relational cost semantics [53].

A convenient property of an asymptotic cost semantics is that if we "unroll" the loop (i.e. this infinite sequence) n times (denoting $e; e; \ldots; e$ by e^n), then:

$$C[e^n; M, r, d^n] \leq nC[e; M, r, d],$$

where the inequality would become strict only after some sort of (program) optimization. From this, we can further define a lower bound for our asymptotic cost:

$$C_L[e; M, r, d] = \liminf_{n \to \infty} \frac{C[e^n; M, r, d^n]}{n} \leq C[e; M, r, d],$$

which characterizes (gives an upper bound for) the best cost that (program) optimization could achieve asymptotically.

This notion of asymptotic cost semantics provides a natural way to understand GOAL program behavior. Further, it provides a distinct advantage over other (non-asymptotic) cost semantics, since constrained optimal parameter tuning is easily achievable with it and almost always (that is, with probability one for all constrained non-degenerate cases which we will elaborate below) better than tuning each execution independently. Let us consider the knob control problem in GOAL. Suppose we have an expression e_k, which depends on a parameter k that corresponds to application knobs and a machine $M_{k'}$, which depends on a parameter k' that corresponds to platform configuration knobs both defined previously in Section 3.4.3. Our cost functions $C[e_k; M_{k'}, \cdot, d]$ represent metrics defined in Section 3.4.3. To minimize the asymptotic cost, we solve an infinite system corresponding to our AdaptSpec defined in Section 3.4.3:

$$\begin{aligned} \min_{\mathbf{k},\mathbf{k}'} \quad & \lim_{N\to\infty} \sum_{i=1}^{N} \frac{1}{N} C[e_{k_i}; M_{k'_i}, o, d] \\ \text{subject to} \quad & \lim_{N\to\infty} \sum_{i=1}^{N} \frac{1}{N} C[e_{k_i}; M_{k'_i}, r_l, d] \preceq \mathbf{R}_l, \forall l, \end{aligned}$$

where o is the resource corresponding to the objective function in the AdaptSpec, r_l is the resource we want to constrain, and $\mathbf{R}_l$ is the constraint value. Notice here that, although the cost is defined differently for every different AdaptSpec, the `restrict` API merely changes the domain of optimization and thus (1) the original asymptotic cost still provides an upper bound after restriction and (2) all the information contained in the original cost function can be reused.

The above is equivalent to the following system:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \sum_{k\in K} C[e_k; M_{k'}, o, d] w_{k,k'} \\ \text{subject to} \quad & A\mathbf{w} \preceq \mathbf{R}, \\ & \sum_{(k,k')\in K} w_{k,k'} = 1, \\ & w_{k,k'} \succeq 0. \end{aligned}$$

where $A_{l,k} = C[e_k; M_{k'}, r_l, d]$, $w_{k,k'}$ is the weight corresponding to each configuration (k, k'), $\mathbf{R}$ is the constraint value vector, and we can denote the corresponding optimal schedule $\{(k_i^*, k_i'^*)\}_{i=0}^{\infty}$. A uni-constraint version of this equivalence has been established and studied by Imes et. al [110].

It is easy to see that:

$$\lim_{N\to\infty} \sum_{i=1}^{N} \frac{1}{N} C[e_{k_i^*}; M_{k_i'^*}, o, d]$$

$$\leq \lim_{N\to\infty} \sum_{i=1}^{N} \frac{1}{N} C[e_{k^{**}}; M_{k'^{**}}, o, d]$$

$$= C[e_{k^{**}}; M, o, d],$$

where (k^{**}, k'^{**}) is a solution of the corresponding system without iterating:

$$\begin{array}{ll} \min\limits_{k} & C[e_k; M_{k'}, o, d] \\ \text{subject to} & C[e_k; M_{k'}, r_l, d] \preceq \mathbf{R}_l, \forall l. \end{array}$$

The equality happens either when the system is unconstrained, or the system is degenerate. This inequality shows what we stated earlier: in the long run, optimal tuning with our asymptotic cost model almost always outperforms tuning with a non-asymptotic cost semantics (i.e., tuning executions independently).

3.7.2 Static Analysis

The GOAL architecture separates the specification of programs from the specification of AdaptSpecs. The resulting flexibility comes at the cost of possible program errors, when the definitions in the two parts of a GOAL program are inconsistent. For example, for a program to be correct, uses of the `Knob` type must correspond to entries in the `knobs` section of an associated AdaptSpec. In addition, the `optimize` construct takes a list of knobs and passes them to the runtime. Using static analysis techniques, the system can provide meaningful feedback early during GOAL application development, and eliminate the possibility of certain types of runtime errors. This subsection presents three static analyses, illustrated in Figure 3.14, which are specific to the implicit programming model.

Finding Unused Knobs Knobs defined in the AdaptSpec but not declared as a Knob type are ignored by the runtime. On the other hand, declared knobs without configurations in the AdaptSpec cannot be used in trade-offs while tuning the system. This can be detected statically by collecting the list of knobs defined in the AdaptSpec, and those declared in the GOAL application code, respectively. The analysis reports the difference, if any, between the two lists of knobs, and marks them as unused to aid in further analysis.

Define K_I to be the set of knobs defined in the AdaptSpec, K_D to be the set of knobs declared in the GOAL application code, K_O to be the set of knobs passed to the optimize construct, and K_A to be the set of knobs affecting the body of optimize construct. A knob $k \in K_A$ if there is a branch of execution which depends on the value of k. Then the unused knobs K_{UU} are defined as $(K_D \setminus K_I) \cup (K_I \setminus K_D)$. The uncaptured knobs K_{UC} are defined as $K_D - K_O$. The unaffected knobs K_{UA} are defined as $K_O - K_A$.

Finding Uncaptured Knobs In the case where a knob is declared, but is not passed to the optimize construct, it is not controlled by the system. We say that these knobs are *uncaptured*. An analysis finds such knobs and emits a warning.

Finding Unaffected Knobs Even when a knob is captured, the user may

```
import FAST

let uncaptured = Knob("uncaptured", 1)
let unaffected = Knob("unaffected", 1)
let affected = Knob("affected", 1)

optimize("app", [unaffected, affected]) {
    var x = read(...)
    if (x < affected.get())
    {  sleep(unaffected.get())  }
    else
    {  for i in 1..<10 { sleep(20) }  }
}
```

(a) Application Code

```
goal app
    min(energy)
    such that
    latency == 0.1
measures
    latency: Double energy: Double
knobs
    unused      = [1 reference,2,3,4]
    uncaptured  = [1,2 reference,3,4]
    unaffected  = [1,2 reference,3,4]
    affected    = [1,2,3,4 reference]
```

(b) Adaptation Specifications

Figure 3.14: An Example Showing Three Kinds of Problematic Knobs

forget to use it, or inadvertently misuse it in the program. Any tuning done by GOAL of such a knob will have no effect on the system, meaning that the runtime will be unable to exploit any trade-offs exposed by this knob. Profiling can expose the presence of such knobs, which will correspond to near-identical entries in the measure table (Section 3.3.3). However, such dynamic analysis can be prohibitively expensive when the configuration space is large. A more efficient alternative is to use static analysis to identify such situations. The analysis begins by building a data flow graph starting from the `optimize` construct. For each node in the graph, it computes the knobs that affect it. Given the annotated graph, it is possible to compute the list of all effective knobs for the `optimize` construct, and issue a warning when this list is missing some declared knob.

This definition of an unaffected knob amounts to an all-or-nothing identification problem, and the optimization problem to be solved by the controller may be ill-defined in the presence of such knobs. Therefore, we did not include sensitivity in our static analyses. Surely, some form of sensitivity (how the initial condition will affect the solution) similar to the "condition number" of a linear system can be defined for our control problem, to detect whether the control system is functioning well. However, this would be computationally expensive, input-dependent and platform-dependent. On the other hand, finding a branch of code that possibly will not be controlled by some knob at runtime is a cheap solution that is both input-independent and platform-independent. To reiterate the main difference: the statistical approach is to detect whether the system is "ill-conditioned" while our current approach is to find whether the system is ill-defined.

3.8 Conclusion

This work motivates the benefits of supporting general purpose adaptation and meta-adaptation. We implement this idea as GOAL, a first-of-its-kind adaptation framework which, unlike prior work, is not restricted to a particular set of knobs and goals. Instead GOAL uses a virtualized AdaptLog that is parameterized by a model after the developers have declared application knobs and metrics. This allows GOAL applications to define the AdaptSpecs, using wide range of knobs and metrics, and seamlessly modify them during execution. The AdaptLog is, itself, adaptive and therefore robust to adapt to nonlinear behavior and changing workloads and operating conditions. We show that GOAL's general approach handles a diverse range of goals and knobs, while performing just as well as problem specific AdaptLogs. However, when the application requires meta-adaptation, GOAL performs significantly better than prior work. Furthermore, GOAL provides an easy-to-use interface that requires minimal effort to add adaptation and meta-adaptation to applications and even supports applications with multiple, independent adaptive modules. We believe that GOAL's generality, dynamism and robustness goes a long way to fulfilling the requirements of complex emerging systems.

CHAPTER 4
WASL: SUPPORTING HARMONIOUS EXECUTION OF COLOCATED ADAPTIVE COMPONENTS

Our work, GOAL, provided a natural way for developers to implement adaptation and meta-adaptation in their applications. However, during our evaluation of GOAL we encountered a critical missing piece for our vision of general-purpose adaptation.

We noticed that once developers have augmented their applications with adaptation they need to deploy their applications. These applications will meet their goals desirably if they execute individually. However, when multiple adaptive applications are colocated, they begin competing for resources and break key assumptions that are required for successful adaptation. This results in negative interference between adaptive applications resulting in an inability to meet their goals.

Latency-sensitive (LS) adaptive applications are often deployed on cloud platforms which are severely underutilized. Hence, to combat this underutilization, cloud providers often colocate multiple applications on the same hardware. Furthermore, to improve efficiency cloud providers may also run a system-level module that adapts knobs of the underlying hardware to minimize power consumption in order to minimize the cloud provider's operating cost. Hence, adaptive applications are expected to deal with negative interference from colocated modules in the real-world.

Prior works have suggested several approaches that allow adaptive applications to deal with negative interference due to colocated adaptive applications. However, the design of prior works have strict requirements on when they can be used. However, severve limitation of prior work such as their requirement of access to knowledge of internal details of the applications, the colocated applications and the underlying platform render them impractical for use in the real-world.

To overcome these limitations we introduce the last part of this body of work, WASL, a novel model-free framework that can be used by adaptive applications individually to allow them to meet their goals in the presence of competing colocated adaptation modules without explicit coordination. At a high-level, WASL works by reinstating assumptions that are broken due to the negative interference between colocated modules by modifying the rate of adaptation of adaptive applications individually so that they do not negatively interfere with each other. WASL is based on properties that are common to all principled adaptation mechanisms, hence, it is a truly general-purpose framework that can be used with any application regardless of the internal details of the application, its adaptation mechanism or the adaptation modules with which the application may be colocated. Our evaluation shows that using WASL applications are able to meet their adaptation goals in the presence of more than one colocated adaptation modules. Latency-sensitive applications achieve up to an 84% reduction in tail latency over naive uncoordinated execution. WASL allows applications achieve performance that is similar to prior work without requiring any explicit coordination making it a practical solution for use in the real-world.

This chapter is organized as follows: Section 4.1 motivates the need for WASL by describing the need for multi-tenancy in the cloud, discussing prior works and their limitations that make them impractical. Section 4.2 the requirements for a usable solution, describes how colocated applications negatively interfere with each other and presents the design and implementation of WASL. Section 4.3 and Section 4.4 provides details of the experimental setup used for evaluation and the evaluation itself respectively. Finally, Section 4.5 provides a brief dicussion of potential limitations of WASL and Section 4.6 provides a conclusion.

4.1 Background and Motivation

In this section we discussion prior work and its limitations that motivate this work. Sectionhow low utilization of resources in cloud datacenters leads to inefficiencies. Section 4.1.1 discusses inefficiencies in cloud datacenters' resource utilization and presents contributions from prior work that aim to increase the efficiency of datacenters via multi-tenancy. In Section 4.1.2 discusses adaptation in latency-sensitive applications, the challenges of colocating multiple adaptive applications and prior works for adapting colocated latency-sensitive applications. Finally, in Section 4.1.3 we discuss the limitations of prior works that make them impractical for use in the real-world.

4.1.1 Multi-Tenancy in the Cloud

Several studies have established that average resource utilization in datacenters is low [186, 39, 151]. This is because user facing, latency-sensitive applications experience significantly different levels of traffic at different times to the day while being scaled out to thousands of servers. This low utilization negatively impacts the operational cost of datacenters. Hence, to maximize server utilization, service providers have started colocating different workloads on the same physical resources [151, 81, 55] rather than powering new resources for each workload.

However, colocating workloads is not straightforward because interference caused by colocated workloads can lead to significant performance degradation. Prior work suggests tacking this problem in two ways. The first way is to infer and colocate workloads that are least likely to interfere with each other [151, 64, 62, 63]. The second approach is to minimize interference by using different software and hardware-level isolation techniques [65, 124, 140, 141].

4.1.2 Adaptation for Latency-Sensitive Applications

Latency-Sensitive (LS) applications operate with service level objective (SLOs) on tail latency [32, 61, 48]. Furthermore, LS application stakeholders want to reduce their own costs my minimizing resources required to meet their SLOs.

However, meeting these requirements is difficult because modern LS applications have several configuration parameters which impact their ability to meet their requirements [126]. Selecting configurations that meet the application's requirements is not easy because optimal knob configurations depend on dynamically varying external factors such as workload and operating conditions [91, 134, 197]. Furthermore, suboptimal configurations can lead to significant performance issues and bugs [84, 207, 107, 163, 203].

Prior work has suggested adaptation as a way of dynamically adjusting these parameters at both the application and the system layer to ensure that application requirements are met. Control theory [110, 103, 78], machine learning [41, 204, 93] and combination of both [100, 154, 102] have been suggested as basis for principled adaptation.

While adaptation allows applications to meet their goals reliably, their guarantees are violated in the presence of other colocated adaptation modules. This is because colocated adaptive modules compete for resources and destructively interfere with each other leading to SLO violations. Prior work mitigates this interference in several ways. First, prior work has suggested synthesizing a single adaptation module that sets parameters on behalf of all colocated adaptation modules [181, 213, 191, 148]. Another approach is to assign priorities colocated modules so that lower priority modules sacrifice their SLO satisfaction for higher priority modules [78]. Alternatively, prior works suggests modifying colocated adaptation modules to enable passing signals between them to perform global coordination [162, 100, 99].

4.1.3 Limitations of Prior Work

To the best of our knowledge, all prior work uses the internal details of each colocated module to delegate adaptation to a global module or perform global coordination using signaling. Table 4.1 shows the adaptation mechanism and the coordination required for using some of the prior works in this domain. Hence, to use prior work, all colocated workloads need to be redesigned and reimplemented along the lines suggested by prior work. This requires different stakeholders to share internal details and coordinate the development with other stakeholders. Additionally, minor changes to individual adaptive modules would require retesting and modification of the entire multimodule system.

Name	**Underlying Principle**	**Required Coordination**
CoAdapt [99]	Adaptive Control	Share details of control actions and tunables between controllers.
Yukta [162]	Robust Control	Share signals on measures that the controller cannot directly impact.
JouleGuard [100]	RL and Adaptive Control	Share decisions of the RL agent with the control-theoretic controller.
Maggio et al. [148]	Adaptive Control	Delegate all adaptation to a monolithic controller.
SimCA [181]	Simplex Control Adaptation	Delegate all adaptation to a monolithic controller.

Table 4.1: Underlying Principle of Adaptation and the required Coordination for Prior Work

The restrictions and requirements imposed by prior works make their use impractical because stakeholders are seldom willing to share proprietary information and even minor changes to individual modules would require reconciliation with all modules. Furthermore, even if the stakeholders are ready to share proprietary details and perform required changes, it would be impossible to enumerate the modules with which their application might be colocated dynamically. This would significantly restrict cloud providers' ability to perform dynamic colocation requiring them to colocate only compatible modules.

Furthermore, such requirements break the assumption of transparent and exclusive resource usage in the cloud as cloud users will have to modify their applications according to the colocated modules. This adds signficant burden on cloud providers as they will need to implement interfaces for discovering and coordinating with colocated modules.

4.2 WASL Design and Implementation

WASL is based on a model-free control algorithm that can be used by colocated adaptive systems independently to dynamically adjust the rate at which they perform adaptation to meet their goals. In this section we describe WASL in detail. We explain how WASL's design allows it to overcome the limitations of prior work, making it a practical framework for mitigating negative interference between colocated adaptive systems.

4.2.1 Key Requirements

To address all limitations of prior work any proposed solution must fulfill the following requirements:

- **No a priori requirements.** While the adaptive modules themselves may or may not require models to operate, the proposed solution itself should not rely on any model because such models are invalidated if the colocated modules change, hence, restricting the cloud provider's ability to colocate any desired modules.

- **Does not require internal details.** The solution should not require colocated adaptive modules to expose internal details such as the knobs being actuated and the method used for adaptation.

- **General-Purpose.** The solutions should be agnostic of the application that it is going to be used with, the principles used for adaptation and the modules with which it may be colocated.

- **Low Overhead.** The solution should have a trivial overhead compared to the adaptive module itself, both in terms of performance and memory.

- **Easily Integrated.** Using the solution should not require the module to make significant changes to its implementation.

In the rest of this section, we discuss how WASL fulfills the aforementioned requirements.

4.2.2 Key Insights

This section illustrates principles underlying WASL. We first show how co-located adaptive systems can destructively interfere with each other, show intuitively how this misbehavior can be avoided, and then demonstrate the key insight that allows us to apply the intuition in practice for a wide variety of adaptive systems.

Specifically, the example is an indexing search engine—Xapian [125]—running on a cloud server. Both have been modified using prior work to make them adapt [154]. We modify Xapian so that it adapts to its workload, meeting a QoS (latency) requirement while minimizing operating cost. Our version of Xapian adaptively changes its use of hyperthreading and core utilization (i.e., it matches the resources it requests from the cloud provider to the workload). Similarly, the cloud provider adapts to workload by changing both core and uncore frequency. The cloud provider's goal is to ensure that the application meets its QoS requirements while minimizing its own operating cost. Both application and system adaptation modules are based on control theoretic principles and thus both provably meet the QoS requirement—when run in isolation [154].

Understanding Adaptive Interference

Figure 4.1 shows Xapian's core usage (top), power (middle) and request latency (bottom) as a function of time while running in three different scenarios. In the first (shown with the blue line), only the system module is active (the application does not adapt). In the second (orange line), only the application module adapts (the system does not). In the third (green line), both modules adapt.

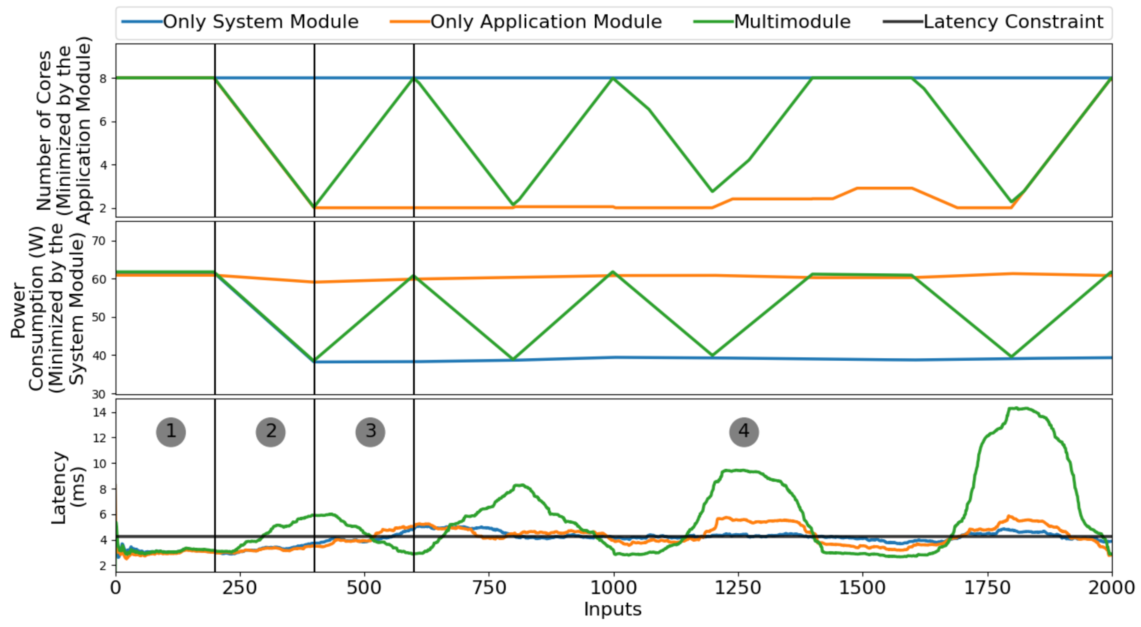


Figure 4.1: Execution Trace of Xapian

This figure illustrates the problem with co-locating multiple, independent adaptive modules. Xapian starts well below the required latency (region ① in Figure 4.1) and thus all adaptation approaches reduce resource usage to minimize costs. The application module reduces core usage (from 8 to 2) and the system module reduces core and uncore frequency (from 2.8Ghz to 2.1Ghz and from 2.8Ghz to 1.2Ghz respectively). Figure 4.1 shows that when only one module is allowed to adapt (i.e. without a co-located adaptation modules), the latency requirements are met. However, when both modules adapt simultaneously, the latency far exceeds the target and the QoS constraint is violated (region ② in the figure). Hence, in the next adaptation window, both modules independently increase resource usage, but the cumulative impact of these changes drives the latency back below the target (region

③). This oscillatory behavior repeats throughout Xapian's execution (region ④ of Figure 4.1 and causes its 95th percentile latency to increase by 35% compared to scenarios with only one adaptive module. This example leads to observation 1:

Observation 1: *Multiple adaptive modules interfere with each other because they assume they are the only actor in the system and thus do not account for the behavior of other adaptive modules.*

Mitigating misbehavior

Observation 1 tells us that because of the cumulative impact of changes made by all co-located modules, the application behavior becomes significantly different from the behavior expected by individual modules during each adaptation window. This mismatch in measured and expected application behavior leads to the aforementioned oscillatory behavior. However, the magnitude of mismatch could have been decreased if the modules had lowered the amount by which they attempt to modify application behavior in a single adaptation window by making less drastic changes to their respective knobs. For example, in the multimodule scenario, if the application module had reduced the number of cores to 4 and the system module had reduced the core and uncore frequencies to 2.4Ghz and 1.2Ghz respectively instead of the values mentioned in our discussion on understanding adaptive interference, the application would have achieved the target latency during the second adaptation window (region ② in the figure), thus avoiding oscillations. This reasoning leads us to observation 2:

Observation 2: *Interference between co-located adaptive modules can be mitigated if each module modifies the magnitude by which it tries to adapt the application be-*

havior.

Modifying the Rate of Adaptation of Modules

To use observation 2 modules need to dynamically modify the rate at which they attempt to change the behavior of the application. We found that, to the best of our knowledge, all prior principled adaptation modules contain quantities, often scalar variables that are a part of their formulations, that determine the weight of the most recent measurements towards adaptation decisions. Setting them to high values increases sensitivity to recent measurements leading to more drastic configuration changes in each adaptation window and vice versa. Hence, by changing these values dynamically we can directly modify the magnitude by which the adaptation module tries to adapt the application behavior. For example, in Odyssey and Lin et al. we can directly change variables denoting the weight assigned to the most recent measurements [80, 137]. In POET, CALOREE and Powerdial this behavior can be achieved by modifying the poles [110, 103, 154]. Similarly, we can modify the Kalman Gain, interaction rate and the workload in ALERT, Aeneas, Grape respectively [198, 50, 175]. This leads us to observation 3:

Observation 3: *To the best of our knowledge, all adaptation modules contain variables that can be dynamically modified to change the rate at which they respond to the measured behavior.*

Determining the Amount by which to Change the Rate of Adaptation

Using observation 2 in any meaningful way requires adaptation modules to determine the amount by which their rate of adaptation should be modified. We note that a vast majority of prior adaptation modules implement adaptation using feedback control loops [172, 147, 161]. Hence, modules need to measure application behavior for feedback. Similarly, modules base the choice of configurations for the following window on a computed expected behavior or reward, this is the behavior expected to be achieve after running the application with the chosen configurations for the next adaptation window. As explained in Section 4.2.2, a high difference between the expected and measured behavior is the root cause of application misbehavior. For our discussion we define *inaccuracy* as the difference between the expected and measured behavior of a module. We note that the higher the rate of change of inaccuracy, the higher the chances that changes made by the adaptation module will not achieve expected behavior and, hence, lead to misbehavior. This leads us to observation 4:

Observation 4: *The inaccuracy of an adaptation module can be computed easily using information that is already being used by the module and the rate of change of this inaccuracy can be used to determine the amount by which to modify the rate of adaptation of a module.*

4.2.3 WASL *Design*

In this section we describe how WASL implements the aforementioned insights while meeting the requirements stated in Section 4.2.1. To this end, we first describe how

WASL uses the rate of change of inaccuracy to update the rate of adaptation of adaptive modules dynamically. Then we describe how WASL addresses the challenge of estimating the rate of change of inaccuracy.

Algorithm

At the end of each adaptation window, WASL's algorithm estimates the rate of change of inaccuracy and computes the factor by which the adaptation module needs to modify its rate of adaptation.

Instead of estimating inaccuracy using raw latency, WASL calculates the inaccuracy at each step using the latency slowdown, this allows WASL to account for the magnitude of the inaccuracy as well. However, we note that not all adaptation modules explicitly use or calculate the slowdown during their operation. But since the measured and expected latency is already available, the slowdown can be calculated easily by dividing it by the estimate of the latency achieved using the most performant configuration which can be estimated in constant time using only the information that is already available in the adaptation module [110, 154, 206].

WASL's algorithm, shown in Algorithm 2, is a straightforward implementation of the aforementioned insights. The algorithm takes as input the expected latency from when the knobs were last modified and the measured latency. It computes a difference between the two and provides this as an input to the `computeRateOfChange` method which returns an estimate of the rate of change of inaccuracy. If this rate of change is greater than a predefined threshold, `inaccuracyThreshold`, then it computes a fraction by which the rate of adaptation should be modified and returns it to the adaptation module, otherwise, it returns the last calculated fraction.

Listing 2 WASL Algorithm

```
Initialize:
    $\text{inacc}_{-1} = 0$
    $\text{inacc}_{-2} = 0$
    $\text{fraction}_{-1} = 0$
    inaccuracyThreshold = T
Inputs:
    expected = Expected Latency Slowdown
    measured = Measured Latency Slowdown
$\text{inacc}_0$ = measured - expected
rateOfChange = rateOfChangeEstimator($\text{inacc}_0$, $\text{inacc}_{-1}$, $\text{inacc}_{-2}$)
if rateOfChange > inaccuracyThreshold then
    fraction = min(max(inaccuracyThreshold / rateOfChange, 0), 1)
else
    fraction = $\text{fraction}_{-1}$
$\text{inacc}_{-1}$ = $\text{inacc}_0$
$\text{inacc}_{-2}$ = $\text{inacc}_{-1}$
$\text{fraction}_{-1}$ = fraction
return fraction
```

The `inaccuracyThreshold` is used to filter out the noise both from process and measurement and action of colocated controllers. It is expected that all controllers are robust to such noise but not robust to the noise due to the actions of colocated controllers. The value of this threshold is independent of other controllers and depends on the environment in which the controller is deployed. WASL gives additional feature for the controller to be aware of the others. Each controller itself should be robust from the noise from the very beginning during the design phase.

This algorithm does not require any knowledge of the of the internal details of the adaptation module and since the algorithm is based on insights that are widely applicable to adaptive systems, the WASL can be used with adaptation modules regardless of the underlying principles used to make adaptation decisions and the knobs being actuated by the adaptation modules.

Next, we discuss four possible implementation for `rateOfChangeEstimator` which is used by WASL to estimate the rate of change of inaccuracy using the inaccuracy and present the most promising solution that was used for WASL.

Estimating Rate of Change of Inaccuracy

Estimating the rate of change of inaccuracy is an integral part of calculating the rate by which an adaptation module should change its rate of adaptation. Hence, we compare four different approaches for estimating the rate of change of the inaccuracy. All methods rely on the inaccuarcy,`inacc`.

The first estimator is the *Linear Instantaneous* that calculates a rate of change of inaccuracy as a numerical derivative [96]:

$$\texttt{rateOfChange} = \texttt{inacc}_0 - 2 \cdot \texttt{inacc}_{-1} + \texttt{inacc}_{-2} \tag{4.1}$$

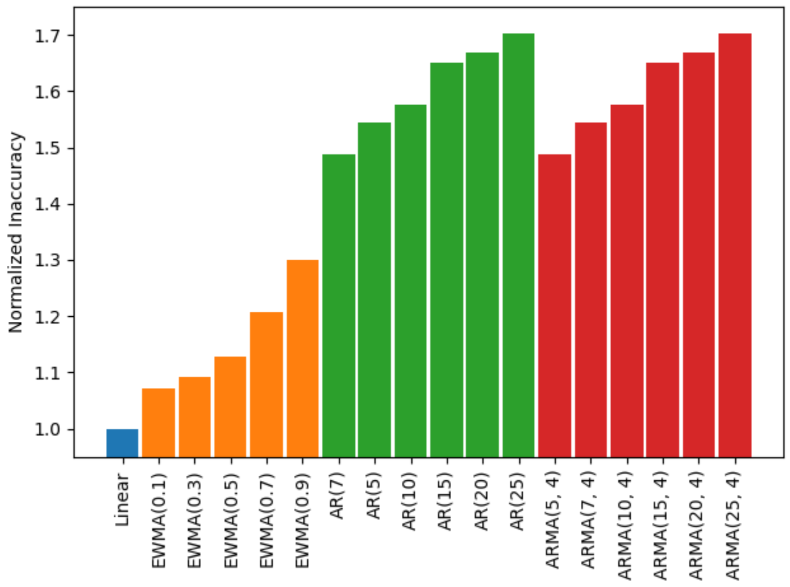
The second estimator is based uses *Exponential Weighted Moving Average*, EWMA(α), with a standard formulation in which α denotes the weight assigned to the historical average. The thrid estimator uses an *Autoregressive Model*, AR(p), in which p is the order of the model. Finally, the fourth estimator uses an *Autoregressive Moving Average Model*, ARMA(p, q), in which p is the order of the autoregressive part and q is the order of the moving average part of the model.

To evaluate the aforementioned estimators, we use the same application and system setup that was described in Section 4.2.2. We augment both the adaptation modules with versions of WASL that use the aforementioned estimators. Figure 4.2 shows the inaccuracy when the modules use WASL with different rate of change estimators normalized to inaccuracy resulting from using the linear estimator. Our experimental evaluation shows that EWMA, AR and ARMA exhibit higher inaccuracy compared to linear instantaneous estimator for both the application and the system module regardless of the parameters for each approach. In fact, the inaccuracy increases as we modify the parameters for EWMA, AR and ARMA to assign more weight to the corresponding historical terms for estimating the rate of change.

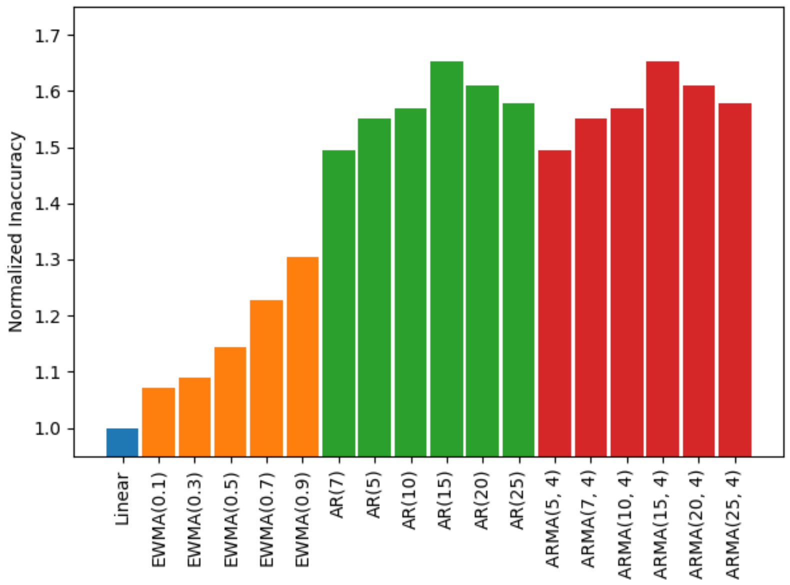
AR, ARMA and EWMA exhibit high inaccuracy because they retain some proportion of the rate of change from previous adaptation steps. However, this information is already accounted for by the adaptive module itself. Hence, these methods double count inaccuracy from the previous terms. Furthermore, decisions taken by the adaptation modules and the change in their operating environment can render this historical information useless for new estimations. This phenomenon can be most clearly observed with EWMA as the error increases with increase in α which is the weigh assigned to previous averages.

Linear instantaneous estimation is ideal for WASL because it does not rely on estimations from previous windows that are no longer relevant due to decisions taken by the adaptation modules. This understanding is validated by our empirical evaluation as shown in Figure 4.2.

Hence, in the final version of WASL we use the linear instantaneous estimator



(a) System Module Error



(b) Application Module Error

Figure 4.2: Inaccuracy in application and system modules when using different methods for estimating rate of change of inaccuracy.

as the `rateOfChangeEstimator`.

4.2.4 Implementation

is implemented as an easy to use framework with a single function which is designed to be used by adaptation modules independently. The function, `get_multiplier`, is a direct implementation of Algorithm 2 with a linear instantaneous rate of change estimator. `get_multiplier` takes 2 floating point variables as arguments:

1. The expected latency slowdown.

2. The measured latency slowdown.

As mentioned in Section 4.2.3, the latency slowdown can be calculated by dividing the raw latency with an estimate of the lowest achievable latency, i.e. the latency when using the most performant configuration [110, 154, 206]. `get_multiplier` returns a single scalar floating point number that indicates the ratio by which the adaptation module needs to modify its rate of adaptation. The returned value can be used directly by the calling adaptation module as described in Section 4.2.2.

4.3 Experimental Methodology

We evaluate WASL in several different scenarios to validate two main properties:

- **Generality**: Is WASL compatible with different types of adaptation modules?

- **Robustness**: Is WASL usable in complex real-world scenarios in which multiple adaptive applications potentially using different methods for adaptation may be colocated with a system-level adaptation module being run by the cloud provider?

To evaluate these properties, we augment benchmark applications with adaptation modules suggested by prior work and colocate them with a system level adaptation module that is also based on prior work.

This section provides details of the components required for adaptation. Section 4.3.1 provides details of the applications used for evaluations and Section 4.3.2 describes the adaptation modules that are used to add adaptation of the applications. Finally, Section 4.3.3 provides details of the hardware used for evaluation.

4.3.1 Benchmark Applications

We use five applications from the tailbench benchmark suite to represent the latency-sensitive applications that need to be hosted by the cloud provider [125]. These applications are:

1. **Xapian**: An open-source search engine.
2. **Moses**: A statistical machine translation system.
3. **Masstree**: A fast, scalable key-value store.
4. **Silo**: An in-memory transactional database.
5. **DNN**: An OpenCV based handwriting recognition application.

All applications execute with multiple threads that take up all of the cores that are allotted to them. During execution, based on their requirements, all applications can request the cloud provider to increase or decrease the number of cores allotted to them based. Similar to Section 4.2.2, for the purposes of this evaluation, the number of cores requested from the cloud provider is considered a proxy for the cost that the application owners have to pay to host their applications on the cloud platform. As mentioned earlier, the applications are augmented with the adaptation modules mentioned in Section 4.3.2.

4.3.2 Adaptation Modules

We make use of three different adaptation modules presented in prior work to add adaptation to the test applications.

- **Adaptive Control Module**: A time-varying adaptive controller module [154].
- **PI Module**: A discrete-time Proportional-Integral (PI) controller module [96].
- **RL Module**: A reinforcement-learning based adaptation module [198].

All applications use one of the aforementioned adaptation modules to meet a high-level goal of minimizing operating cost while meeting a tail latency constraint, as described in Section 4.2.2, by actuating hyperthreading and the number of cores requested from the cloud provider.

The system module, expected to be operated by the cloud provider, also uses one of the aforementioned adaptation modules to actuate the core and uncore frequnecy (as mentioned in Section 4.2.2) to meet the high-level goal of minimizing power consumption while ensuring that all hosted applications are able to meet their goals. Hence, the system module is expected to have some insight into the performance of the hosted applications [81, 55]. In our experiments, all applications hosted on the node periodically provide their tail-latency constraint and the measured tail-latency to the system module.

4.3.3 Hardware Platform

We conduct all of our experiments using hardware provided by the Chameleon configurable cloud computing platform [117]. We run all experiments on a single node running Ubuntu 18.04 (GNU/Linux 5.4) with an Intel Xeon 6126 Gold processor with hyperthreading and TurboBoost and 192 GB of RAM. The node has 12 physical threads and 24 hyperthreads. The system modifies the core and uncore frequency of the hardware to ensure that the hosted applications meet their goal.

4.4 Evaluation

In this section we present our evaluation of the key aspects of WASL, specifically:

1. How well does WASL perform with different adaptation modules?
2. How well does WASL perform with multiple application modules?
3. How does WASL impact the operating environment of colocated adaptation modules?
4. How much overhead does WASL incur?

These aspects justify the generality and robustness of WASL making it a feasible solution for the real-world. In all of the presented experiments the tail latency constraint are set to values close to the expected real-world requirements of such applications [125, 61].

4.4.1 How well does WASL *perform with different adaptation modules?*

To validate generality, we evaluate WASL's ability to allow applications that are using different adaptation modules to meet their latency constraints when they are colocated with a system-level module.

For comparison, we present results from three different executions. First, we present results from a version in which the application and the system module are executing naively (Uncoordinated Multimodule). Second, we present results from a version based on prior work in which all adaptation has been delegated to a single specialized adaptation module (Monolithic). Finally, we present results from a version in which both the application and the system module individually modify their rate of adaptation dynamically using WASL (WASL).

Adaptive Control Module In these experiments both the application and the system module use the Adaptive Control Module for adaptation. The monolithic module used for this set of experiments is also based on the Adaptive Control Module. Figure 4.3 shows the tail latencies from the three different versions of executions normalized with respect to the Monolithic version.

As we can see the uncoordinated multimodule execution leads to a ∼1.3x increase in the tail latencies of the applications based on the geometric mean. This results in a significant drop in applications' QoS. However, when using WASL the application

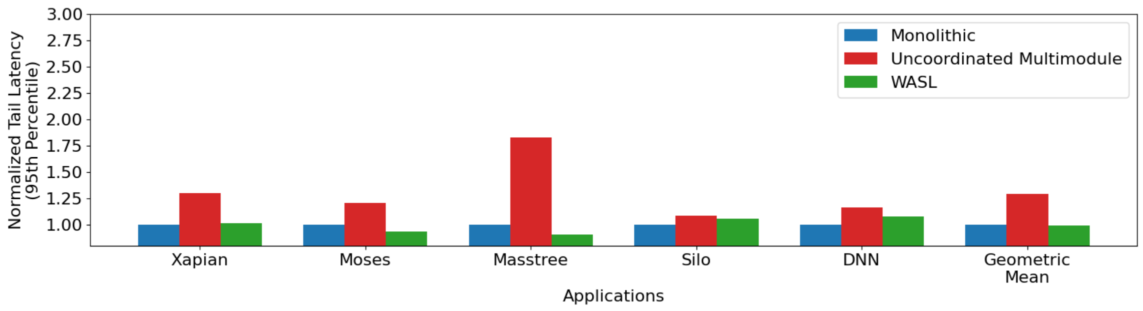


Figure 4.3: Tail latencies of applications colocated with a system module both using Adaptive Control Module for adaptation.

is able to successfully temper its rate of adaptation such that it successfully meets its tail latency constraint providing performance that is comparable to versions in which both the application and the system module delegate their adaptation to a monolithic module.

PI Module In these experiments all the applications, the system module and the monolithic adaptation module use the PI module for adaptation. The parameters for each of the aforementioned modules are fine-tuned such that when they are able to meet their goal when they are executing in isolation. Figure 4.4 shows the tail latencies of the different versions of the experiment normalized with respect to the monolithic version.

Similar to the results with the adaptive control module, based on the geometric mean, the uncoordinated multimodule execution leads to a ∼1.84x increase in the tail latencies of applications resulting in a significant drop in their QoS. On the other hand, WASL allows the adaptation modules to effectively modify their rate of adaptation ensuring that the tail latencies are roughly equivalent to the executions

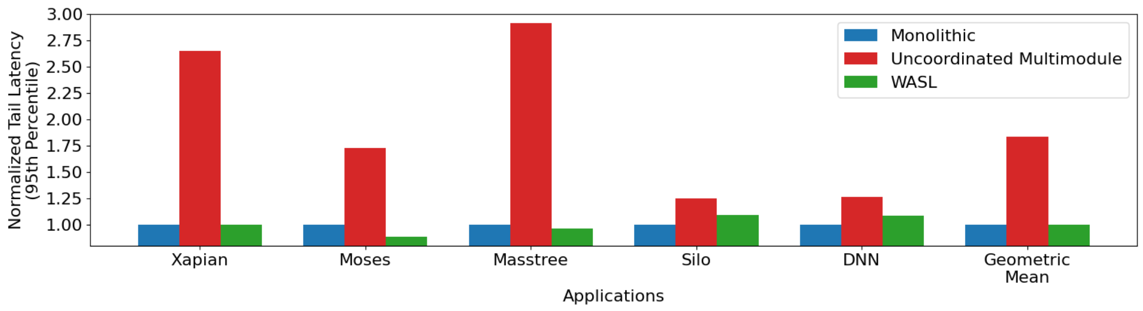


Figure 4.4: Tail latencies of applications colocated with a system module both using a PI Module for adaptation.

using a monolithic version.

RL Module We get a similar outcome when all components use the RL module for adaptation. Figure 4.5 shows the tail latencies normalized with respect to the tail latency of the monolithic version.

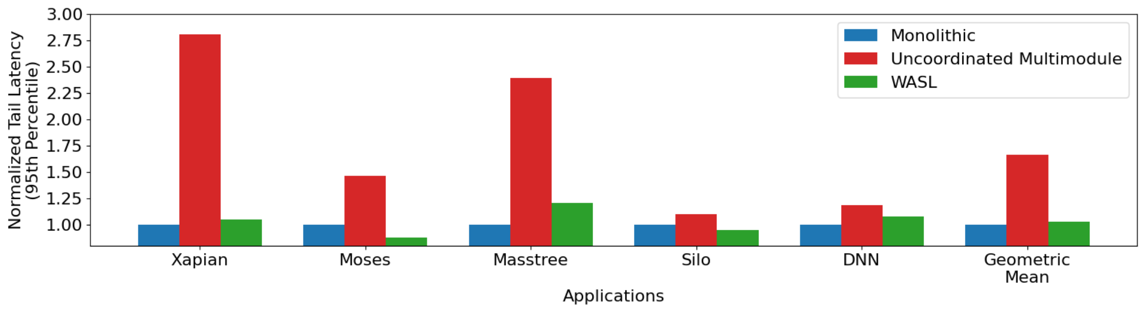


Figure 4.5: Tail latencies of applications colocated with a system module both using the RL Module for adaptation.

Based on the geometric mean, uncoordinated execution results in a ∼1.66x increase in tail latencies of applications. However, WASL is able to guide the rate of adaptation of the adaptation modules individually to ensure that the applications provide performace that is similar to the performance when using a monolithic adaptation module.

It should be noted that while the monolithic versions allow the applications to meet their QoS requirements, they are impractical for the real-world because of the reasons stated in Section 4.1.3. On the other hand, WASL is able to provide performance similar to the monolithic versions without requiring any adaptation delegation, information sharing between modules or significant changes in implementation. This is achieved by incorporating WASL in each adaptation module individually with only minimal changes. Furthermore, WASL did not require any kind of a model to allow the applications to recover their performance.

These experiments illustrate the generality of WASL. WASL allows applications to meet their QoS goals without possessing any knowledge of the application itself, the technique being used for adaptation or what external factors impact the operating conditions of the application.

4.4.2 How well does WASL *perform with multiple application modules?*

As mentioned in Section 4.1.1, cloud providers are expected schedule multiple applications together on the same node. Hence, to valide the robustness of WASL for use in real-world scenarios, we evaluate the tail latency when the cloud provider colocates two applications together with a system-level adaptation module (as described in Section 4.3.2) on the same node. In such scenarios, using prior work to share information or delegate adaptation will be impossible because it will not be possible to know which applications are colocated by the cloud provider. We breakdown this part of the evaluation into two distinct scenarios.

Symmetric Adaptation Scenario In this scenario both the applications and the system module use the same module for adaptation. We present tail latencies normalized with respect to the tail latency of the versions of applications that use WASL.

Figure 4.6 shows the tail latencies of the applications when all colocated adaptation modules use adaptive control for adaptation. As we can see, in this scenario uncoordinated multimodule execution hopelessly degrades the QoS service with latency increasing by upto ∼2.5x. However, WASL allows applications to recover their performance. Based on the geometric mean, applications that use WASL are able to provide ∼1.36x lower tail latency.

Similarly, Figure 4.7 shows tail latencies when all adaptation modules use the PI Module for adaptation. As mentioned in Section 4.4.1, the parameters of the PI module are set such that each module individually meets its QoS goal.

Similar to the Adaptive Control Module, the PI Module significantly degrades performance, increasing the tail latency by upto 30x in certain situations. However, WASL allows applications to recover this lost performance. Based on the geometric mean, WASL based applications provide a ∼3.6x reduction in tail latency.

Finally, Figure 4.8 shows results when all adaptation modules use the RL Module for adaptation. Using WASL applications are able to provide a tail latency that is ∼1.48x lower than naive uncoordinated adaptation.

Asymmetric Adaptation Scenario Now we present results from another scenario that is likely to occur in the real-world in which all colocated adaptation

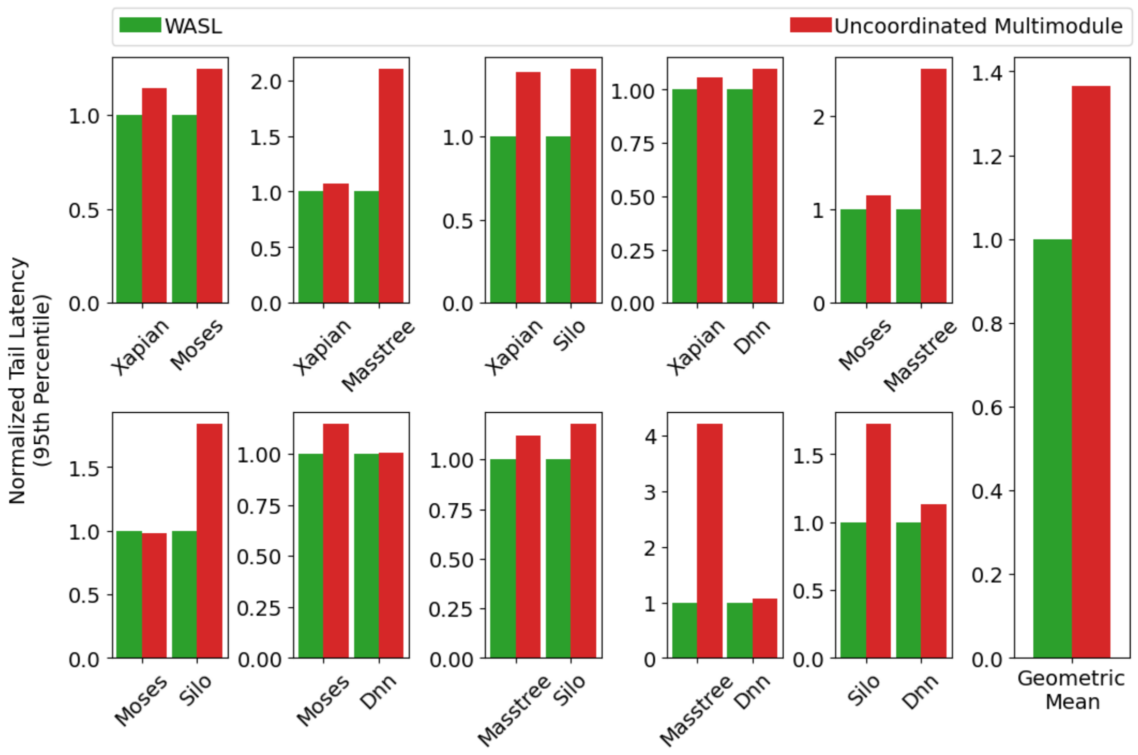


Figure 4.6: Tail latencies of two applications colocated with a system module all using the Adaptive Control Module for adaptation.

modules use a different underlying adaptation module for adaptation. Figure 4.9, shows results from our experiments. For these experiments, the first application always uses an adaptive control module, the second application always uses a PI module and the system module always uses a RL module for adaptation. As expected, uncoordinated execution degrades performance significantly. However, WASL allows applications to recover the lost performance even when colocated modules use different adaptation methodologies. As we can see, using WASL applications exhibit ∼1.35x lower tail latency than uncoordinated execution.

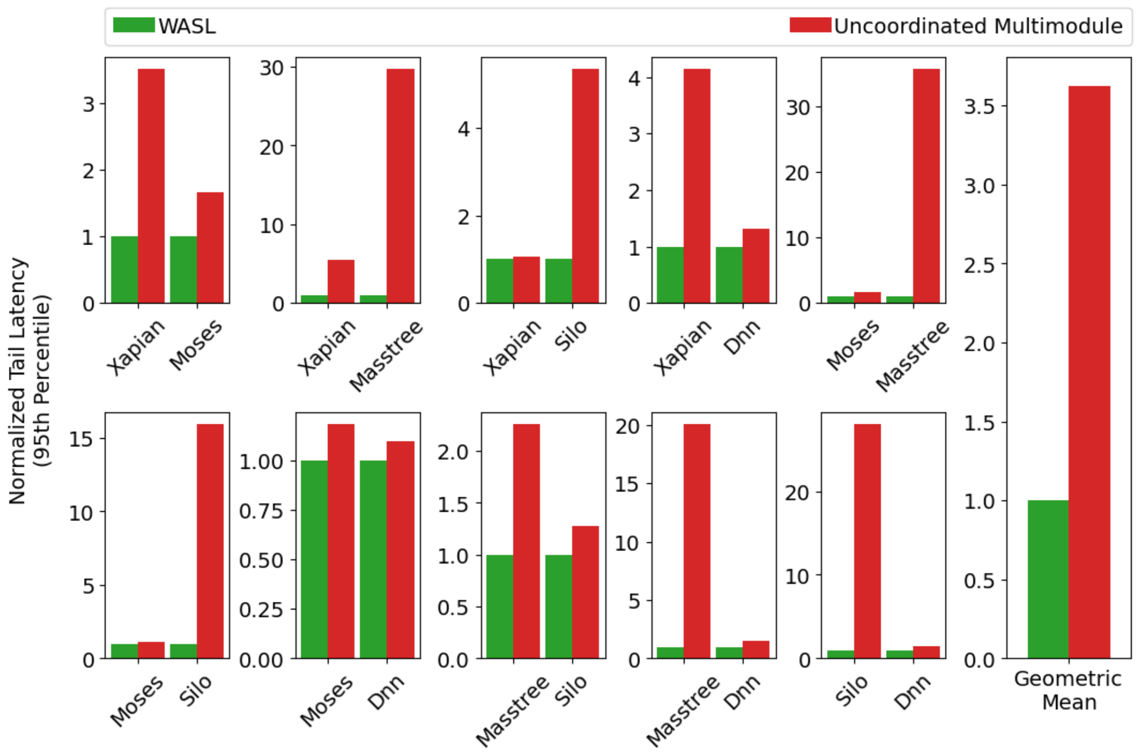


Figure 4.7: Tail latencies of two applications colocated with a system module all using the PI Module for adaptation.

It should be noted that along with ensuring a low tail latency, WASL also ensures that the applications meet their tail-latency constraint in absolute terms as well by providing a mean-absolute-percentage-error of less than 5%.

These experiments illustrate that WASL allows applications to meet their QoS goal without requiring any knowledge of the application itself, the mechanism used for adaptation or any kind of a model. Similarly, it does not need to know the modules that may be colocated with it or the mechanism used for adaptation by the colocated modules. Hence, WASL overcomes the limitations of prior work making

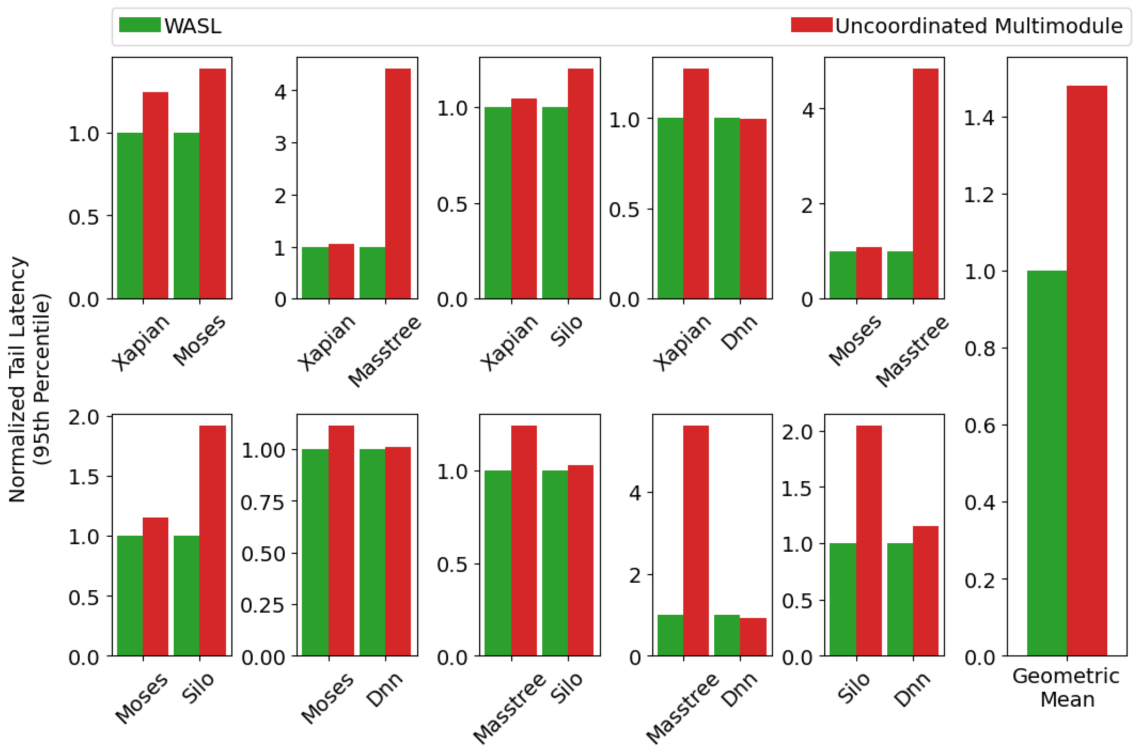


Figure 4.8: Tail latencies of two applications colocated with a system module all using the RL Module for adaptation.

it a practical solution for use in the real-world.

4.4.3 How does WASL *impact the operating environment of colocated adaptation modules?*

Now we examine how well WASL is able to manipulate the operating environment of colocated modules to allow them to adapt applications successfully. As mentioned in Section 4.2.2, adaptation modules operate with the assumption that their operating environment either does not change or when it does change the changes

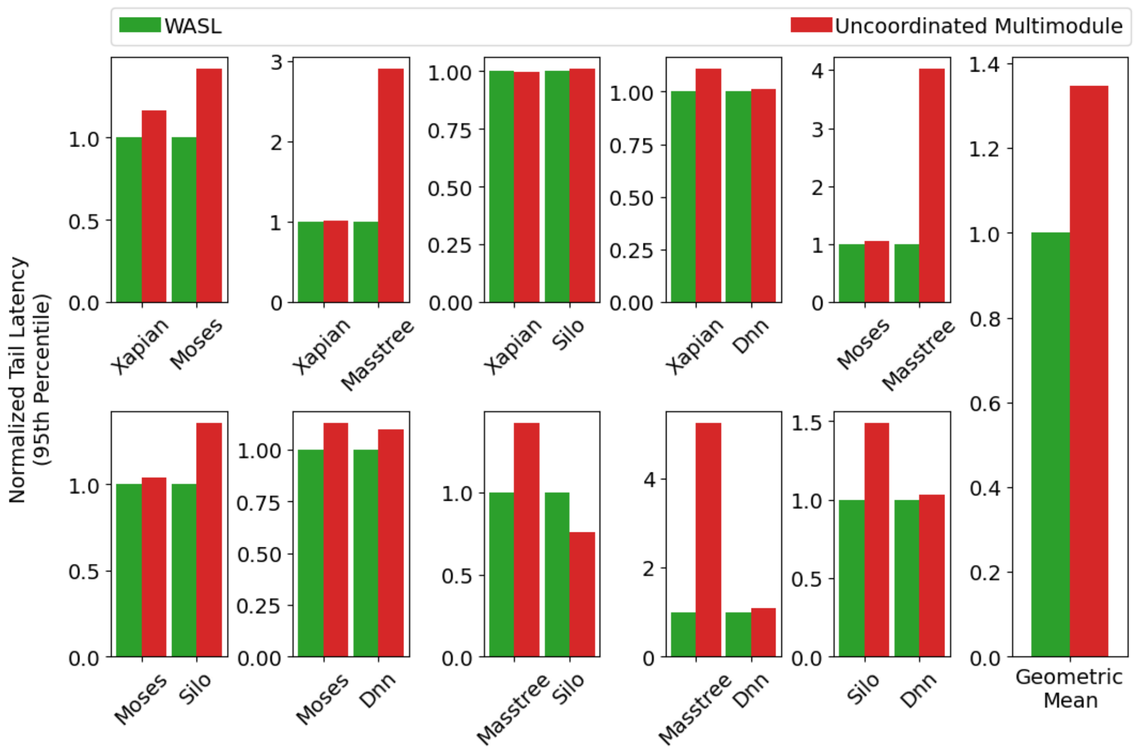


Figure 4.9: Tail latencies of two applications colocated with a system module all using a different module for adaptation.

last long enough for them to recover from those changes by actuating their knobs. However, this assumption is broken when multiple colocated adaptation modules operate naively. WASL reinstates this assumption by manipulating the rate at which adaptation modules change the operating environment of colocated modules. The evidence of this is not only in the recovered performance but also in the operating environment.

However, since there is no direct method to measure the operating environment, we use an alternative metric. We first estimate the tail-latency that can be achieved by setting the application in the most performant configuration. This is the same value that is used to calculate the slowdowns for WASL as mentioned in Section 4.2.3. To measure changes in the operating environment, we measure the standard deviation in the estimated values of the tail latencies for the applications.

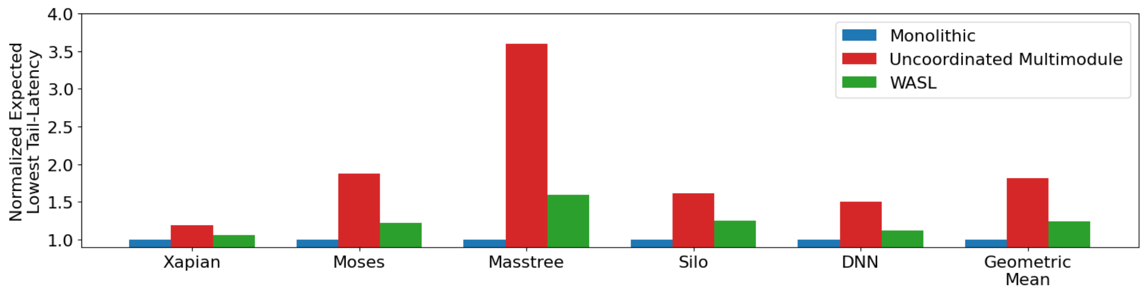
Figure 4.10a shows the standard deviation of our estimates normalized with respect to the estimates from the Monolithic execution. As we can see Uncoordinated Multimodule execution leads to a ∼1.8x increase in standard deviation of the

Figure 4.10 shows the standard deviation of our estimates normalized with respect to the estimates from the Monolithic execution. As we can see, regardless of the module used for adaptation, Uncoordinated Multimodule execution leads to a increase in disturbance of the operating environment compared to the Monolithic execution. However, WASL allows the operating environment to stabilize resulting in a roughly 46%, 95% and 88% reduction in standard deviation of the operating environment when all modules use the Adaptive Control Module, RL Module and PI Module respectively.

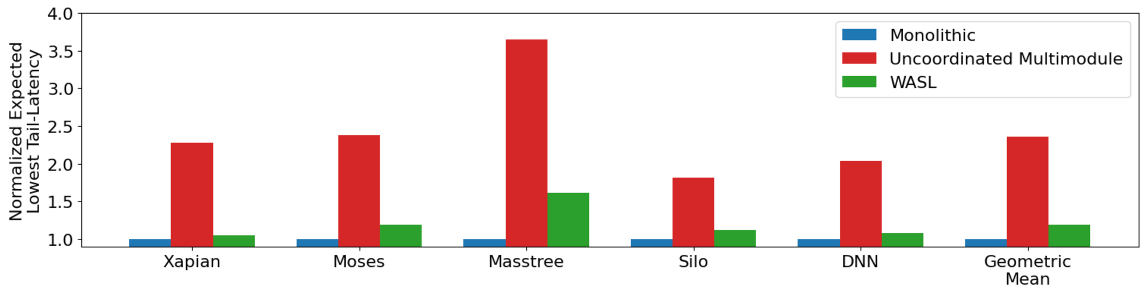
These results show that WASL successfully reinstates the key assumption for the colocated adaptation modules allowing them to meet their respectively goals successfully.

4.4.4 How much overhead does WASL *incur?*

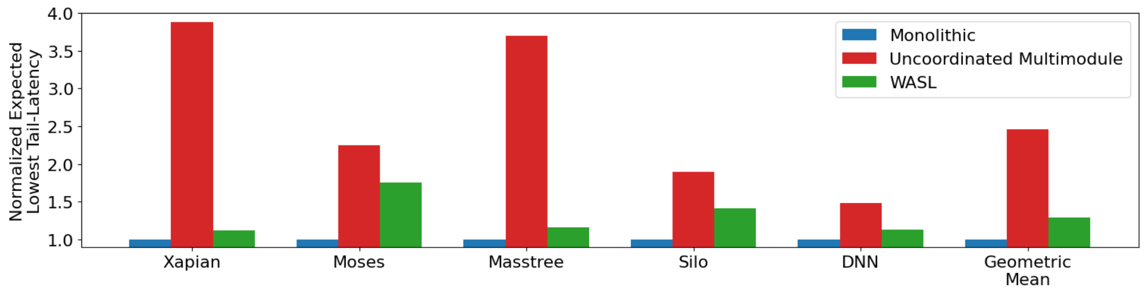
All experiments from the presented evaluation use the WASL algorithm with the Linaer Instantaneous estimator. This means that, overall, WASL computes its return values in $\mathcal{O}(1)$. Similarly, WASL only needs to store four scalar floating point



(a) Normalized Standard Deviation with Adaptive Control Module



(b) Normalized Standard Deviation With RL Module



(c) Normalized Standard Deviation With PI Module

Figure 4.10: Normalized Standard Deviation in the estimated lowest tail-latency.

values, namely, `inacc_1`, `inacc_2`, `fraction_1` and `inaccuracyThreshold`. Hence, the memory overhead of WASL is also in $\mathcal{O}(1)$. We note that, as mentioned in Section 4.2.3, some adaptation modules do not explicitly estimate the lowest achievable tail latency. In this case, the cost of estimation would also be incorporated in WASL's overhead. However, as in the presented evaluation, this can be calculated in $\mathcal{O}(1)$. For all of the presented experiments, we use a Kalman Filter to estimate the lowest tail latency [206].

We define the total time for a single adaptation action as the total time required to perform tail-latency estimation, execute WASL and compute the optimal configuration for the following adaptation window. We found that the time required to estimate the lowest tail-latency along along with time to execute WASL was dominated by time required to compute configurations the configurations. Concretely, based on the geometric mean, the tail-latency estimation and WASL consumed less than 1% of the total time spent to perform a single adaptation action. This is significantly smaller than the nontrivial overhead incurred by prior approaches that need to coordinate information sharing or adaptation delegation between adaptation modules.

4.5 Discussion and Limitations

Our evaluation shows interesting results in which Uncoordinated Multimodule execution perform significantly worse for extreme-low-latency applications. This is understandable because applications with extremely low latencies are impacted greatly by smaller changes in the operating conditions. Furthermore, in such applications a large number of requests will be served between adaptation actions. This is done to amortize the cost of performing adaptation. We found that even for such applications WASL is able to moderate the operating environment effectively to allow them to meet their QoS goals.

The most basic requirement of WASL is that the adaptation module that uses WASL must have a quantifiable way to modify its rate of adaptation. WASL cannot be used for adaptation modules which do not have this capability. However, during our review of principled adaptation modules suggested by prior work we could not find such a module.

WASL allows adaptation modules to cope with changes in operating environment caused by colocated modules. The underlying assumption is that the adaptation modules themselves have the robustness to respond successfully to other external factors. WASL cannot be used in scenarios in which the adaptation modules would not be able to perform successful adaptation even if they were not colocated with any other modules.

4.6 Conclusion

This work motivates the benefits of supporting harmonious execution of multiple colocated adaptation modules with minimal explicit coordination. The requirements for using prior works, such as sharing proprietary information between modules or delegating adaptation, make them sub-optimal for real-world use. To overcome these limitations, this work introduces WASL, a novel framework that allows colocated adaptation modules to achieve their quality-of-service goals without any explicit co-ordination. This allows the stakeholders to develop their applications and modules independently without having to considering the details of the colocated adaptation modules. Since, WASL does not rely on any knowledge of the application or the technique used for adaptation, developers can completely decouple their development from WASL. Furthermore, adding WASL to adaptation modules requires minimal effort. And unlike prior work, using WASL, cloud providers have the freedom to colocate different adaptation modules without having to worry about their compatibility. Our evaluation shows that WASL allows applications to achieve behavior that is comparable to prior works, allowing them to meet their quality-of-service requirements in a range of real-world scenarios. We believe that WASL's generality and robustness make it a feasible solution for use in the real-world.

CHAPTER 5
CONCLUSION

This dissertation addresses the requirement of general-purpose and dynamic adaptation in computing systems.

The first project, DDS, proposes a novel iterative and adaptive protocol for streaming in machine learning based video-analytics applications. DDS' novel adaptation module allows it to handle variations in bandwidth significantly better than the heuristic-based approaches used by prior work.

However, building adaptation for DDS highlighted issues in the existing approach for engineering adaptation modules for systems. The prevailing approach for engineering adaptation for systems treats adaptation as an after-thought rather than a core component of the system. Hence, adaptation is added using system-specific modules once the critical parts of the system have been implemented and its tunables and goals have been finalized. This results in non-reusability of adaptation modules which is fundamentally opposed to good engineering practices in software systems. Furthermore, it requires developers to be experts both in their domain and in principles required to implement robust adaptation modules such as machine learning and control theory. Prior work recognizes and addresses this issue by suggesting frameworks that package pre-built adaptation modules in libraries and language runtimes. However, a major limitation of suggested frameworks is that they focus on adaptation for a specific set of knobs and goals. Hence, frameworks from prior approaches cannot be used to perform adaptation involving different sets of knobs and goals. Furthermore, due to their limited support they do not have any interfaces for the system to interact with any aspect of adaptation once the framework has been instantiated. Thus, they cannot be used to perform any meta-adaptation.

Hence, to overcome these limitations, in the second project we propose, GOAL, a first-of-its-kind framework for general-purpose adaptation in computing systems. GOAL's library API can be used to easily augment systems with adaptation. GOAL uses a virtualized, time-varying adaptation module that is independent of any specific model relating metrics to knobs. Furthermore, GOAL allows developers to declare desired adaptive behavior using a novel domain-specific language and provides interfaces that allows the system to dynamically modify all aspects of adaptation during execution.

Finally we turn our focus towards the deployment of adaptive applications. We motivate the need for colocating adaptive applications to increase resource usage efficiency while recognizing that colocating competing adaptive applications results in negative interference that leads to applications' inability to meet their goals. This phenomenon has been widely studies by prior work which suggested several approaches to mitigate the negative interference between adaptive applications. However, a common feature of suggested approaches is that they impose restrictions on the methods that can be used by colocated applications for adaptation. Similarly, such approaches require details of applications and adaptation to be shared between colocated applications. Some of the suggested approaches even require synthesizing a monolithic adaptation module and expect all colocated applications to delegate the entirety of their adaptation to the monolithic adaptation module. Needless to say, these features of prior work make them sub-optimal for use in the real-world since applications often have proprietary information that stakeholders are seldom willing to share. Furthermore, such approaches limit the system administrators' ability to colocate applications together as it requires all colocatable applications to be enumerated beforehand so that the necessary changes may be made to those applications.

Hence, in the last part of this dissertation we present WASL, a novel easy-to-use and model-free framework that allows harmonious execution of adaptive applications without any explicit coordination. WASL imposes no restrictions on the method that an application uses for adaptation and it can be used by all applications independently without requiring any knowledge of the applications themselves or other applications that may be colocated with them. WASL utilizes information that is already available to an application's adaptation module to modify the rate of adaptation of the modules to ensure that the application continues to meet its goal. Because of its minimal requirements, WASL is the only feasible solution for use in real-world adaptive applications.

We hope that using contributions of this dissertation developers will be able to develop and deploy systems that can fulfil complex adaptation requirements and respond to changing adaptation requirements over the lifetime of systems. We also hope that the presented frameworks provide motivation for further work in building practical solutions for adaptation and meta-adaptation in real-world systems.

5.1 Future Work

We strongly believe that the frameworks presented in this dissertation are a first step towards providing general-purpose robust adaptation to computing systems.

GOAL focuses on performing generalized adaptation for applications executing on a single node. A natural next step would be to propose methods for adaptation of distributed applications on over-provisioned clusters. The solution must be scaled out appropriately so as to perform adaptation with minimal overhead and coordinate adaptation decisions between nodes in a cluster.

The dissertation focuses on performing adaptation from within applications. However, another venue for adaptation is the operating system on which the application executes. Modern operating systems have hundreds of tunable parameters that govern how they manage their virtual memory, file systems and block I/O. As with application parameters, it is well known that setting kernel-level parameters appropriately is essential to ensure high performance of applications running on the OS. Hence, a system that modifies the parameters of the kernel based on the application and its workload would be profoundly interesting. For such a system to be general, instead of relying on information directly from the applications, it would have to use microarchitectural and OS-level counters available in the kernel to identify the workload and set configurations accordingly. Our initial studies show that, when running real-world workloads, data from such counters can often have significant noise due to background kernel level processes. Identifying workload patterns from this data would a valuable contribution and an essential first step in developing a system for adapting kernel parameters.

APPENDIX A
INDIVIDUAL CASE STUDIES FOR GOAL

We now present details of the implementation and execution for each of the applications used in the case studies (Section 3.5.1). We implement 6 case studies with a published adaptive design from the literature and consider a scenario that requires runtime modification of the AdaptSpecs. As mentioned earlier, Table 3.2 summarizes the initial goals, the required meta-adaptation, and the platform for each case study. For each case study, we compare GOAL based version's execution to a version that uses an application-specific AdaptLog as described in Section 3.5.2. We show that for a single AdaptSpec, GOAL*'s more general AdaptLog performs as well as prior work for synthesizing AdaptSpec-specific AdaptLogs; however, when the application performs meta-adaptation,* GOAL *provides large benefits in performance across a range of metrics.*

A.1 Video Object Detection

Consider a CCTV camera augmented with object detection. It has a required frame rate. It should also minimize power when there is no object in the scene. But, when an object of interest appears, it should meet a minimum quality requirement. To do so, we dynamically modify the AdaptSpec to `restrict` the range of certain `knobs` when the object is present.

Knobs. We declare four app-level knobs: `qp`, `subme`, `merange` and `reframes` and two sys-level knobs: the number of cores and core frequency. `qp`, `merange`, `reframes`, and `subme` control throughput/quality tradeoffs.

Measures. We consider throughput, power, and quality.

Goal. We start with `goal`:

```
min(powerConsumption) such that throughput == 20.0
```

Figure A.1 shows the behavior with GOAL and prior work that synthesizes a specialized AdaptLog. Both meet the 20 frame/s goal despite dynamic changes from frame to frame. The difference is quite noticeable, however, when an object is detected (frame 150). Here, the GOAL implementation performs meta-adaptation using `restrict` as follows:

```
if detector.objInFrame() { qp.restrict([10, 20]) }
```

In contrast, prior work continues with the initial AdaptSpec. Figure A.1 shows that between when an object is detected (frame 150) and when it leaves (frame 900), the frames are encoded with a higher quality. While the object of interest is present the GOAL's AdaptLog handles this knob restriction intelligently—using more cores at higher frequencies—so the goal is still met. Here, the runtime increases energy, but it is consistent with the updated AdaptSpec, and is exactly what the developer desires: higher quality at a cost of higher energy when an object is present. The GOAL version provides roughly 22% higher quality while the object of interest is in the frame.

Once the object of interest is gone, the restriction is lifted:

```
if !detector.objInFrame() { qp.control() }
```

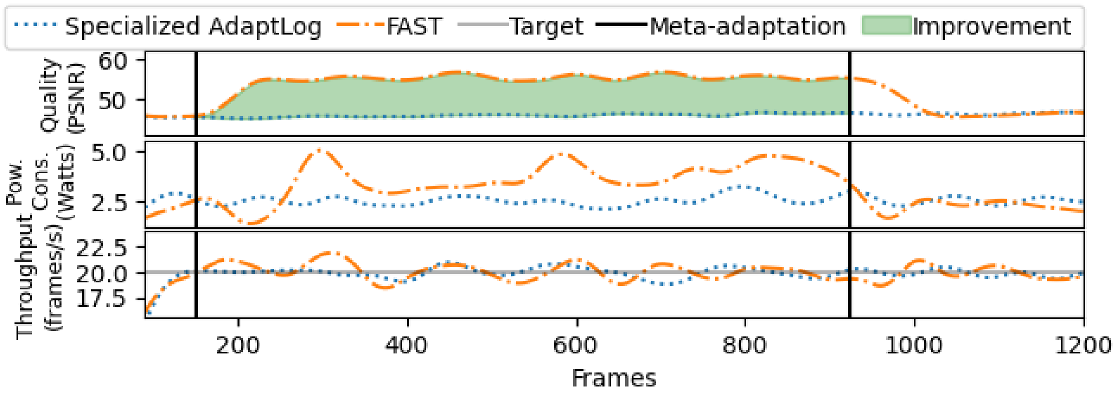


Figure A.1: Time series of the execution of Video Object Detection.

Upon a call to `control` GOAL version seamlessly returns to minimizing power while meeting the throughput constraint.

The limited adaptation support of prior work forces the system to execute with a single AdaptSpec and thus live with sub-optimal behavior, potentially missing important information while the object of interest is in the frame. But the GOAL version allows the system to easily perform meta-adaptation, facilitating behavior that is in line with the requirements.

A.2 Service Oriented Architecture

Consider a provider who has three distinct service implementations [76]. Incoming requests are redirected to any of the three implementations, which, while functionally the same, have different latency, reliability and cost. During high traffic times the priority is meeting a latency target while maximizing reliability. However, during low traffic times the goal is to minimize cost while delivering acceptable reliability.

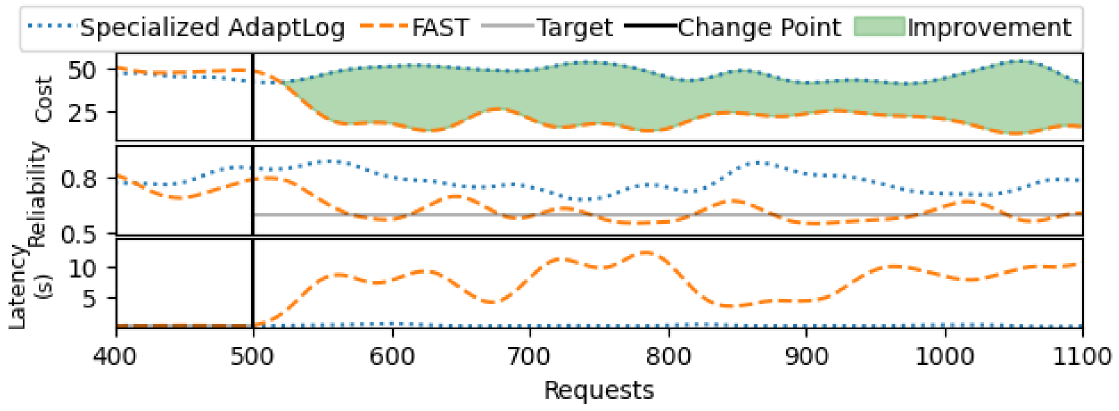


Figure A.2: Time series of the execution of Service Oriented Architecture.

The system needs to quickly switch between goals because taking longer to perform this meta-adaptation would result in the queue changing again, leading to a different goal.

Knobs. We use: `p1`, `p2`, and `sl`. `p1`, `p2` dictate the probability of the request being allocated to the services and `sl` dictates the request's service level.

Measures. Response time, cost and reliability.

Goals. The application starts with a goal to maximize reliability with a response time of 0.5 seconds:

```
max(reliability) such that latency == 0.5
```

During low-traffic times the goal switches to:

```
min(cost) such that reliability == 0.6
```

Implementing this meta-adaptation using GOAL requires a single call to `intend`. However, using prior work that synthesizes a specialized AdaptLog, the system is

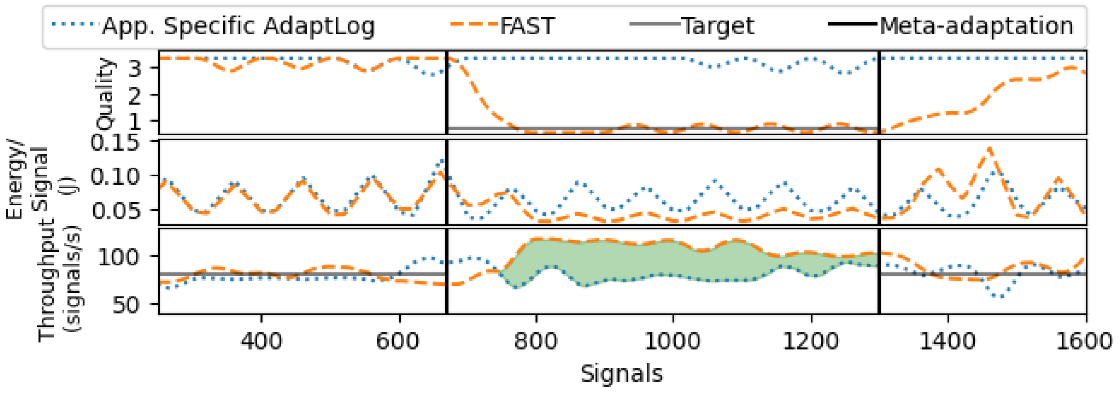


Figure A.3: Time series of the execution of Synthetic Aperture Radar.

forced to execute using a single AdaptSpec throughout the execution [76].

Figure A.2 shows the results of both GOAL and prior work. Both maintain an average latency of 0.5 seconds while the length of queue is greater than the threshold. When the queue length drops below the threshold the goal is changed. The GOAL-based system adjusts the goal using a single call to `intend`, reducing the cost by roughly 63%

The limited support by existing frameworks forces the system to continue execution using a higher cost during a time when the reliability and latency can be relaxed to reduce the cost. Hence, GOAL allows the system to exhibit optimum behavior according to the developer's requirements.

A.3 Synthetic Aperture Radar

Consider a Synthetic Aperture Radar (as previously implemented with compile-time adaptation [188]). In normal operation mode, the SAR should produce the highest quality results while meeting a throughput constraint. However, when its current field has already been scanned or is deemed unimportant by a human operator it should maximize throughput while meeting a quality constraint that is enough for the human operator to determine when to change the operating mode.

Knobs. We use filter sizes (Coarse Decimation Ratio, Fine Decimation Ratio), spatial granularity (Number of Beams, Number of Ranges), core number, and frequency.

Measures. The throughput and quality (as energy above threshold) of the signal processing.

Goals. The system starts with an initial goal of maximizing quality while meeting a throughput constraint:

```
max(quality) such that throughput == 80.0
```

But as the plane flies over an unimportant region, the application changes the goal to:

```
max(throughput) such that quality == 0.7
```

Figure A.3 shows the execution of the GOAL and prior work [76] versions. Both versions maintain an average throughput of 80 signals per second with an average quality of 0.7 (70% above threshold). However, when the SAR passes over a region that is deemed unimportant by the human operator, the GOAL version changes the adaptation goal using a single call to `intend`, as shown above. However, the prior work based version is forced to continue execution with the initial goal that produces low throughput.

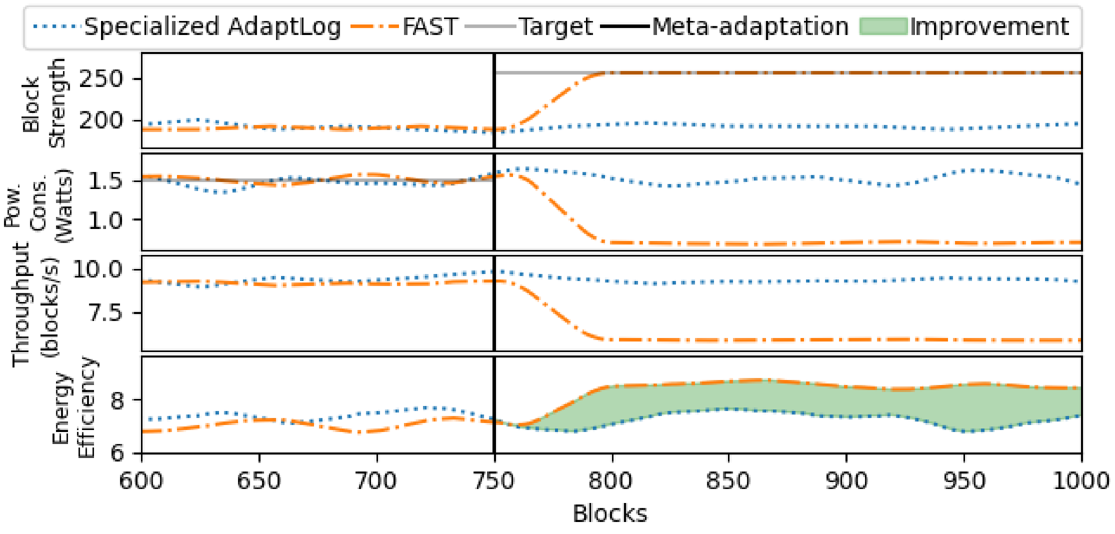


Figure A.4: Time series of the execution of AES Encryption

While over the already scanned region, GOAL version provides 1.44× higher throughput than the existing framework based version (as illustrated using the green region in Figure A.3), allowing it to quickly pass over the area while still providing high enough quality for the operator to make accurate decisions.

A.4 AES Encryption on Mobile Systems

When plugged in, the system wants maximum throughput and high power consumption is tolerable. But when unplugged, efficiency (throughput / power consumption) should be maximized. Additionally, certain critical data requires a higher level of encryption and a longer key. We consider a scenario in which critical data needs to be encrypted and the phone is unplugged simultaneously.

Thus, the system must perform meta-adaptation to ensure the highest security and energy efficiency. Here, energy efficiency can be declared as a separate metrics or it can be derived from metrics that GOAL already tracks. We do the latter to demonstrate the expressiveness of GOAL's interface. This expressiveness allows developers to express goals compoased of arithmetic expressions over the metrics defined in the AdaptSpec without changing the core implementation.

Knobs. We use `keySize`, cores, and core frequency.

Measures. Throughput, power and encryption strength.

Goals. At the beginning when the phone is plugged in the program uses the following goal:

```
max(throughput) such that powerConsumption == 1.5
```

Figure A.4 compares the GOAL application to prior work that builds a customized AdaptLog for managing the original AdaptSpec. The critical data arrives at block 700. At this point, GOAL, using the `intend` function, allows dynamic construction of more complicated goals and adapts to both the battery and security changes (note the objective is a function of two measures):

```
if batteryState() == .BatteryStateCharging  {
    intend(to: .maximize,
    objective: "throughput/powerConsumption",
    suchThat: [(measure: "blockStrength",
        is: .equalTo, goal: 256.0)])
}
```

When the system is processing critical data while unplugged, the GOAL based version increases the energy efficiency by roughly 18% while still meeting the strict security requirements. But the version based on prior work suffers with low energy efficiency and high power consumption during a time when power is an important concern.

A.5 Search Engine

We now examine the execution of our final application the Search Engine. The search engine example is a simplified Swift port of swish++ [143] that runs index-based search on books from Project Gutenberg [16].

Consider a scenario in which the search engine provider maintains a target throughput. But when certain terms that indicate a critical event are encountered in large numbers, the application needs to perform meta-adaptation to return more thorough results; e.g., providing more results for searches related to developing public safety issues.

Knobs. We use `maxDocs`, number of cores, core frequency.

Measures. The throughput and power.

Goals. The application starts out with the goal of meeting a throughput target while minimizing power:

```
min(powerConsumption) such that throughput == 18
```

When more thorough results are required, we call `restrict()` to force GOAL's to use large values of `maxDocs`.

Figure A.5, shows the executions using a specialized AdaptLog and GOAL. Both meet the throughput requirement of 18 queries per second while minimizing power

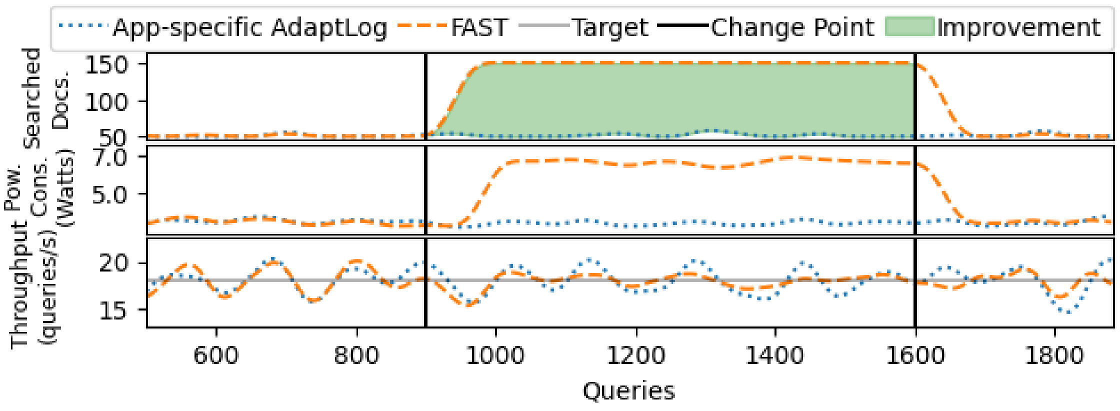


Figure A.5: Time series of the execution of Search Engine.

consumption. However, only the GOAL version responds appropriately when the heuristic determines that more thorough search results are needed (query 900) by making a call to `restrict`:

```
if triggerSearchTermsMetric() > threshold {
    maxDocs.restrict([150, 170])
}
```

After a call to restrict, the GOAL version immediately starts producing results based on higher number of searched documents (a proxy for quality). During this time, GOAL version produces results that are roughly 3× higher in quality than results from prior work version.

Once the critical situation has passed a call to `control` lifts the restrictions on `maxDocs`.

A.6 Optical Character Recognition

We now examine the execution of our final application an Optical Character Recognition pipeline. Using this pipeline, a client submits images to a server which then extracts text from them using `libtessarct`[19] and sends it back to the client. The server can resize images to control the time required to extract text from a particular image thereby impacting the quality of the results.

Consider a scenario in which the server is required to a target throughput while minimizing power. After a period of time the user requests the server to maximize the quality of results for a certain set of images while maintaining the target throughput.

Knobs. We use `imageSize`, number of cores, core frequency.

Measures. The throughput, power and quality.

Goals. The application starts out with the goal of meeting a throughput target while minimizing power:

```
min(powerConsumption) such that throughput == 8
```

During this time the server chooses to perform recognition on images with a reduced size, lowering the quality of the images.

Figure A.6, shows the executions using a specialized AdaptLog and GOAL. Both meet the throughput requirement of 8 images per second while minimizing power consumption. However, only the GOAL version responds appropriately when the client requests the server to maximize the quality of results (query 700) by making a call to `intend`:

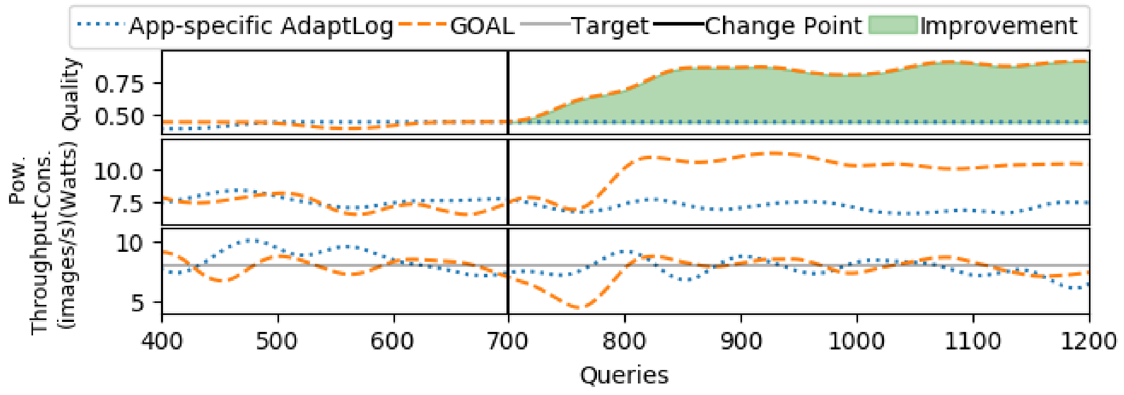


Figure A.6: Time series of the execution of Optical Character Recognition.

```
intend(to: .maximize,
    objective: "quality",
    suchThat: [(measure: "throughput",
      is: .equalTo, goal: 8.0)])
```

After an adjustment period the GOAL version starts producing the higher quality results while still meeting the throughput target. However, during this period, the power consumption of the GOAL version also increases. During this time, GOAL version produces results that are roughly 1.9× higher in quality than results from prior work version. x

BIBLIOGRAPHY

[1] Amazon deeplens cameras. `https://aws.amazon.com/deeplens/`.

[2] Are we ready for ai-powered security cameras? `https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/`.

[3] Arlo: Wire-free hd and hdr smart home security cameras.

[4] Benchmarking videos used in dds. `https://github.com/ddsprotocol/dds`.

[5] Can 30,000 cameras help solve chicago's crime problem? `https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html`.

[6] Cloud-based video analytics as a service of 2018. `https://www.asmag.com/showpost/27143.aspx`.

[7] Dashjs. `https://github.com/Dash-Industry-Forum/dash.js`.

[8] Dds: Machine-centric video streaming. `https://github.com/dds-research-project/dds`.

[9] Google kubernetes engine.

[10] How ai based video analytics is benefiting retail industry. `https://www.lanner-america.com/blog/ai-based-video-analytics-benefiting-retail-industry/`.

[11] Hp r0w29a tesla t4 graphic card - 1 gpus - 16 gb. `https://www.amazon.com/HP-R0W29A-Tesla-Graphic-Card/dp/B07PGY6QPT/`. Accessed: 2020-1-29.

[12] Jetson agx xavier. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/`. Accessed: 2020-1-29.

[13] Nvidia jetson systems. `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/`.

[14] Nvidia tesla deep learning product performance. `https://developer.nvidia.com/deep-learning-performance-training-inference`. Accessed: 2020-1-29.

[15] Official implementation of efficient cascading residual network for sr. `https://github.com/nmhkahn/CARN-pytorch`.

[16] Project gutenberg. https://www.gutenberg.org/.

[17] Smraza raspberry pi 4 camera module 5 megapixels 1080p. https://www.amazon.com/Smraza-Raspberry-Megapixels-Adjustable-Fish-Eye/dp/B07L2SY756/. Accessed: 2020-1-29.

[18] Tensorflow detection model zoo. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

[19] Tesseract ocr.

[20] Video meets the internet of things. https://www.mckinsey.com/industries/high-tech/our-insights/video-meets-the-internet-of-things.

[21] Video surveillance: How technology and the cloud is disrupting the market. https://cdn.ihs.com/www/pdf/IHS-Markit-Technology-Video-surveillance.pdf.

[22] Vision meets drones: A challenge. http://www.aiskyeye.com/.

[23] Wi-fi vs. cellular: Which is better for iot? https://www.verypossible.com/blog/wi-fi-vs-cellular-which-is-better-for-iot.

[24] x264 open source video lan. https://www.videolan.org/developers/x264.html.

[25] Data generated by new surveillance cameras to increase exponentially in the coming years. http://www.securityinfowatch.com/news/12160483/, 2016.

[26] Namhyuk Ahn, Byungkon Kang, and Kyung-Ah Sohn. Fast, accurate, and lightweight super-resolution with cascading residual network. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 252–268, 2018.

[27] Frederico Alvares, Gwenaël Delaval, Eric Rutten, and Lionel Seinturier. Language support for modular autonomic managers in reconfigurable software components. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 271–278, 2017.

[28] Ganesh Ananthanarayanan, Victor Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath Sivalingam, and Sudipta Sinha. Real-time video analytics – the killer app for edge computing. *IEEE Computer*, October 2017.

[29] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: Online autotuning through local competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 91–100, New York, NY, USA, 2012. ACM.

[30] Apple. Swift. `https://developer.apple.com/swift/`, 2019.

[31] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research amp; Development in Information Retrieval*, SIGIR '14, page 103–112, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th International ACM SIGIR Conference on Research Development in Information Retrieval*, SIGIR '14, page 103–112, New York, NY, USA, 2014. Association for Computing Machinery.

[33] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[34] Baochun Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, 1999.

[35] S. Barati, F. A. Bartha, S. Biswas, R. Cartwright, A. Duracz, D. Fussell, H. Hoffmann, C. Imes, J. Miller, N. Mishra, Arvind, D. Nguyen, K. V. Palem, Y. Pei, K. Pingali, R. Sai, A. Wright, Y. Yang, and S. Zhang. Proteus: Language and runtime support for self-adaptive software development. *IEEE Software*, 36(2):73–82, March 2019.

[36] Saeid Barati, Ferenc A. Bartha, Swarnendu Biswas, Robert Cartwright, Adam Duracz, Donald Fussell, Henry Hoffmann, Connor Imes, Jason Miller, Nikita Mishra, Arvind, Dung Nguyen, Krishna V. Palem, Yan Pei, Keshav Pingali, Ryuichi Sai, Andrew Wright, Yao-Hsiang Yang, and Sizhuo Zhang. Proteus: Language and runtime support for self-adaptive software development. *IEEE Software*, 36(2):73–82, 2019.

[37] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* 2013.

[38] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[39] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007.

[40] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), dec 2016.

[41] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 215–224, New York, NY, USA, 2010. ACM.

[42] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237, New York, NY, USA, 1995. ACM.

[43] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<t>: A first-order type for uncertain data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 51–66, New York, NY, USA, 2014. Association for Computing Machinery.

[44] John Boyd. The essence of winning and losing. Online document, 1995.

[45] John Raymond Boyd. Destruction and creation. 1976.

[46] John Raymond Boyd. The essence of winning and losing. 1996.

[47] Stephen P. Bradley, Arnoldo C. Hax, and Thomas L. Magnanti. *Applied mathematical programming.* Addison-Wesley, 1977.

[48] Jake D. Brutlag, Hilary Hutchinson, and Maria Stone. User preference and search engine latency. In *JSM Proceedings, Qualtiy and Productivity Research Section.*, Alexandria, VA, 2008.

[49] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 217–232, New York, NY, USA, 2017. ACM.

[50] Anthony Canino, Yu David Liu, and Hidehiko Masuhara. Stochastic energy optimization for mobile gps applications. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 703–713, New York, NY, USA, 2018. ACM.

[51] Liyu Cao and Howard M. Schwartz. Analysis of the kalman filter based estimation algorithm: an orthogonal decomposition approach. *Automatica*, 40(1):5–19, 2004.

[52] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[53] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 316–329, New York, NY, USA, 2017. ACM.

[54] H. Chen, M. Song, J. Song, A. Gavrilovska, and K. Schwan. Hears: A hierarchical energy-aware resource scheduler for virtualized data centers. In *2011 IEEE International Conference on Cluster Computing*, pages 508–512, 2011.

[55] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168. ACM, 2015.

[57] Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. Adascale: Towards real-time video object detection using adaptive scaling. *arXiv preprint arXiv:1902.02910*, 2019.

[58] Sandeep P Chinchali, Eyal Cidon, Evgenya Pergament, Tianshu Chu, and Sachin Katti. Neural networks meet physical networks: Distributed inference between edge devices and the cloud. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2018.

[59] High Efficiency Video Coding and ITUT Rec. H. 265 and iso, 2013.

[60] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. *Microsoft Azure: Planning, Deploying, and Managing Your Data Center in the Cloud.* Apress, USA, 1st edition, 2015.

[61] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[62] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013.

[63] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[64] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 473–488, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Christina Delimitrou and Christos Kozyrakis. Bolt: I know what you did last summer... in the cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[66] Jiankang Deng, Jia Guo, Xue Niannan, and Stefanos Zafeiriou. Arcface: Additive angular margin loss for deep face recognition. In *CVPR*, 2019.

[67] Yixin Diao, J.L. Hellerstein, S. Parekh, R. Griffith, G.E. Kaiser, and D. Phung. A control theory foundation for self-managing computing systems. *IEEE Journal on Selected Areas in Communications*, 23(12):2213–2222, 2005.

[68] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 39–52, New York, NY, USA, 2019. Association for Computing Machinery.

[69] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 39–52, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 362–373. ACM, 2011.

[71] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems.* Prentice-Hall, Inc., USA, 9th edition, 2000.

[72] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 557–570, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Mirko D'Angelo. Decentralized self-adaptive computing at the edge. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, page 144–148, New York, NY, USA, 2018. Association for Computing Machinery.

[74] John Emmons, Sadjad Fouladi, Ganesh Ananthanarayanan, Shivaram Venkataraman, Silvio Savarese, and Keith Winstein. Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 27–32, 2019.

[75] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. *SIGARCH Comput. Archit. News*, 40(1):301–312, March 2012.

[76] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM.

[77] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM.

[78] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 13–24, New York, NY, USA, 2015. ACM.

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

[80] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, page 48–63, New York, NY, USA, 1999. Association for Computing Machinery.

[81] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[82] Andrei Frumusanu. Improving the exynos 9810 galaxy s9: Part 1. https://www.anandtech.com/show/12615/improving-exynos-9810-galaxy-s9-part-1, 05 2018.

[83] Andrei Frumusanu. The samsung galaxy s9 and s9+ review: Exynos and snapdragon at 960fps. https://www.anandtech.com/show/12520/the-galaxy-s9-review/5, 03 2018.

[84] Archana Ganapathi, Yi-Min Wang, Ni Lao, and Ji-Rong Wen. Why pcs are fragile and what we can do about it: A study of windows registry problems. Technical Report MSR-TR-2004-25, June 2004.

[85] Jim Gao. Machine learning applications for data center optimization, 2014.

[86] Shilpa George, Junjue Wang, Mihir Bala, Thomas Eiszler, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards drone-sourced live video analytics for the construction industry. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 3–8. ACM, 2019.

[87] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. Technical report, 1998.

[88] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[89] J. Goodman, A. P. Dancy, and A. P. Chandrakasan. An energy/security scalable encryption processor using an embedded variable voltage dc/dc converter. *IEEE Journal of Solid-State Circuits*, 33(11):1799–1809, Nov 1998.

[90] Google. Android Power Management: Battery Saver. `https://developer.android.com/about/versions/pie/power#battery-saver`, 2020.

[91] Guang-Liang Li. An analysis of impact of workload fluctuations on performance of computer systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 256–264, 1995.

[92] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[93] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[94] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[95] Joseph L. Hellerstein. Engineering autonomic systems. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, page 75–76, New York, NY, USA, 2009. Association for Computing Machinery.

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

[97] João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-speed tracking with kernelized correlation filters. *IEEE transactions on pattern analysis and machine intelligence*, 37(3):583–596, 2014.

[98] Congrui Hetang, Hongwei Qin, Shaohui Liu, and Junjie Yan. Impression network for video object detection. `https://arxiv.org/pdf/1712.05896.pdf`, Dec 2017.

[99] H. Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 223–232, July 2014.

[100] Henry Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214, New York, NY, USA, 2015. ACM.

[101] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 259–264, New York, NY, USA, 2012. Association for Computing Machinery.

[102] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 259–264, New York, NY, USA, 2012. Association for Computing Machinery.

[103] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 199–212, New York, NY, USA, 2011. Association for Computing Machinery.

[104] Petr Jan Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001.

[105] Y. Hsu, K. Matsuda, and M. Matsuoka. Self-aware workload forecasting in data center power prediction. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 321–330, 2018.

[106] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

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

[108] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017.

[109] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[110] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. Poet: a portable approach to minimizing energy under soft real-time constraints. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 75–86, April 2015.

[111] Connor Imes, Lars Bergstrom, and Henry Hoffmann. A portable interface for runtime energy monitoring. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016,

page 968–974, New York, NY, USA, 2016. Association for Computing Machinery.

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

[113] Connor Imes, Steven Hofmeyr, and Henry Hoffmann. Energy-efficient application resource scheduling using machine learning classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[114] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[115] Samvit Jain, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, and Joseph E. Gonzalez. Scaling Video Analytics Systems to Large Camera Deployments. In *ACM HotMobile*, 2019.

[116] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 29–42, 2018.

[117] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266. ACM, 2018.

[118] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN)*, 22(1):326–340, 2014.

[119] Kinjal A Joshi and Darshak G Thakore. A survey on moving object detection and tracking in video surveillance system. *International Journal of Soft Computing and Engineering*, 2(3):44–48, 2012.

[120] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, page 117–126, New York, NY, USA, 2009. Association for Computing Machinery.

[121] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive resource provisioning for virtualized servers using kalman filters. *ACM Trans. Auton. Adapt. Syst.*, 9(2), July 2014.

[122] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[123] Aman Kansal, Scott Saponas, A.J. Bernheim Brush, Kathryn S. McKinley, Todd Mytkowicz, and Ryder Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 661–676, New York, NY, USA, 2013. ACM.

[124] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610, 2015.

[125] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[126] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[127] D. H. K. Kim, C. Imes, and H. Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *ICCPS*, 2015.

[128] Hyunji Kim, Ahsan Pervaiz, Henry Hoffmann, Michael Carbin, and Yi Ding. Scope: Safe exploration for dynamic computer systems optimization, 2022.

[129] Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian. Xtune: A formal methodology for cross-layer tuning of mobile embedded systems. *ACM Trans. Embed. Comput. Syst.*, 11(4), January 2013.

[130] Karl Krauth, Stephen Tu, and Benjamin Recht. Finite-time analysis of approximate policy iteration for the linear quadratic regulator. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8514–8524. Curran Associates, Inc., 2019.

[131] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[132] Sanjay Krishnan, Adam Dziedzic, and Aaron J Elmore. Deeplens: Towards a visual data management system. *arXiv preprint arXiv:1812.07607*, 2018.

[133] Robert Laddaga. Guest editor's introduction: Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14(3):26–29, May 1999.

[134] Guang-Liang Li and P. Dowd. An analysis of network performance degradation induced by workload fluctuations. *IEEE/ACM Transactions on Networking*, 3(04):433–440, jul 1995.

[135] Robert LiKamWa and Lin Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 213–226. ACM, 2015.

[136] Ji Lin, Chuang Gan, and Song Han. Tsm: Temporal shift module for efficient video understanding. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 7083–7093, 2019.

[137] Xue Lin, Yanzhi Wang, and Massoud Pedram. A reinforcement learning-based power management framework for green computing data centers. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 135–138, 2016.

[138] Mason Liu and Menglong Zhu. Mobile video object detection with temporally-aware feature maps. `https://arxiv.org/pdf/1711.06368.pdf`, Mar 2018.

[139] Weiyang Liu, Yandong Wen, Zhiding Yu, Ming Li, Bhiksha Raj, and Le Song. Sphereface: Deep hypersphere embedding for face recognition. In *CVPR*, pages 6738–6746, 2017.

[140] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[141] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[142] Yao Lu, Aakanksha Chowdhery, and Srikanth Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 57–70. ACM, 2016.

[143] Paul J. Lucas. Swish++. `http://swishplusplus.sourceforge.net/`.

[144] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 129–144, Carlsbad, CA, October 2018. USENIX Association.

[145] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann. Self-adaptive video encoder: Comparison of multiple adaptation strategies made simple. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 123–128, May 2017.

[146] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4), December 2012.

[147] Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4), dec 2012.

[148] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Automated control of multiple software goals using multiple actuators. In *Proceedings of the 12th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2017. ACM, 2017.

[149] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Automated control of multiple software goals using multiple actuators. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 373–384, New York, NY, USA, 2017. Association for Computing Machinery.

[150] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 197–210, New York, NY, USA, 2017. Association for Computing Machinery.

[151] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[152] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[153] Marwa Meddeb. *Region-of-interest-based video coding for video conference applications*. PhD thesis, Telecom ParisTech, 2016.

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

[155] A. Moustafa and M. Zhang. Learning efficient compositions for qos-aware service provisioning. In *2014 IEEE International Conference on Web Services*, pages 185–192, 2014.

[156] J-R Ohm. Advances in scalable video coding. *Proceedings of the IEEE*, 93(1):42–56, 2005.

[157] Frans P. B. Osinga. *Science, Strategy and War: The strategic theory of John Boyd*. Routledge, 2007.

[158] A. Padovitz, S. Loke, and A. Zaslavsky. Awareness and agility for autonomic distributed systems: Platform-independent publish-subscribe event-based communication for mobile agents. In *2012 23rd International Workshop on Database and Expert Systems Applications*, page 669, Los Alamitos, CA, USA, sep 2003. IEEE Computer Society.

[159] Ahsan Pervaiz, Yao Hsiang Yang, Adam Duracz, Ferenc Bartha, Ryuichi Sai, Connor Imes, Robert Cartwright, Krishna Palem, Shan Lu, and Henry Hoffmann. Goal: Supporting general and dynamic adaptation in computing systems. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, page 16–32, New York, NY, USA, 2022. Association for Computing Machinery.

[160] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.

[161] Barry Porter, Roberto Rodrigues Filho, and Paul Dean. A survey of methodology in self-adaptive systems research. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 168–177, 2020.

[162] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. Yukta: Multilayer resource controllers to maximize efficiency. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 505–518, 2018.

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

[164] Amir M. Rahmani, Bryan Donyanavard, Tiago Mück, Kasra Moazzemi, Axel Jantsch, Onur Mutlu, and Nikil Dutt. Spectr: Formal supervisory control and coordination for many-core systems resource management. *SIGPLAN Not.*, 53(2):169–183, March 2018.

[165] Redhat. Tuned: Tuning Profile Delivery Mechanism for Linux. `https://tuned-project.org/`, 2020.

[166] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, pages 779–788, 2016.

[167] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[168] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[169] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6):1137–1149, 2017.

[170] Eric Rutten, Nicolas Marchand, and Daniel Simon. Feedback control as mape-k loop in autonomic computing. In Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, editors, *Software Engineering for Self-Adaptive Systems III. Assurances*, pages 349–373, Cham, 2017. Springer International Publishing.

[171] M. Révay and M. Líška. Ooda loop in command control systems. In *2017 Communication and Information Technologies (KIT)*, pages 1–4, Oct 2017.

[172] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), may 2009.

[173] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[174] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

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

[176] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[177] Homeland Security. Cctv technology handbook. Online Documen, 2013.

[178] Michael Seufert, Sebastian Egger, Martin Slanina, Thomas Zinner, Tobias Hossfeld, and Phuoc Tran-Gia. A survey on quality of experience of http adaptive streaming. *IEEE Communications Surveys & Tutorials*, 17(1):469–492, 2015.

[179] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. Mete: Meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 13–24, New York, NY, USA, 2011. Association for Computing Machinery.

[180] Haichen Shen, Seungyeop Han, Matthai Philipose, and Arvind Krishnamurthy. Fast video classification via adaptive cascading of deep models. *arXiv preprint*, 2017.

[181] Stepan Shevtsov and Danny Weyns. Keep it simplex: Satisfying multiple goals with guarantees in control-based self-adaptive systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 229–241, New York, NY, USA, 2016. Association for Computing Machinery.

[182] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: A platform for os-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 289–302, New York, NY, USA, 2009. Association for Computing Machinery.

[183] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system

for perpetual systems. In *In Proceedings of The Fifth International ACM Conference on Embedded Networked Sensor Systems (SenSys '07), Syndey*, 2007.

[184] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. *SIGPLAN Not.*, 43(9):253–264, September 2008.

[185] Akshitha Sriraman and Thomas F. Wenisch. µtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[186] Staff. Cloud computing is still dangerously underutilized, Nov 2017.

[187] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 145–158, USA, 1999. USENIX Association.

[188] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 607–621, 2016.

[189] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803. USENIX Association, November 2020.

[190] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 328–339. IEEE, 2017.

[191] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. A case for coordinated resource management in heterogeneous multicore platforms. In Ana Lucia Varbanescu, Anca Molnos, and Rob van Nieuwpoort, editors, *Computer Architecture*, pages 341–356, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[192] Konstantinos Tovletoglou, Lev Mukhanov, Dimitrios S. Nikolopoulos, and Georgios Karakonstantis. Harmony: Heterogeneous-reliability memory and qos-aware energy management on virtualized servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 575–590, New York, NY, USA, 2020. Association for Computing Machinery.

[193] Jilin Tu, Ana Del Amo, Yi Xu, Li Guari, Mingching Chang, and Thomas Sebastian. A fuzzy bounding box merging technique for moving object detection. In *2012 Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS)*, pages 1–6. IEEE, 2012.

[194] Stephen Tu and Benjamin Recht. Least-squares temporal difference learning for the linear quadratic regulator. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5005–5014, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[195] David Vengerov. A reinforcement learning framework for utility-based scheduling in resource-constrained systems. *Future Generation Computer Systems*, 25(7):728 – 736, 2009.

[196] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[197] Jóakim von Kistowski, Hansfried Block, John Beckett, Klaus-Dieter Lange, Jeremy Arnold, and Samuel Kounev. Analysis of the influences on server power consumption and energy efficiency for cpu-intensive workloads. 02 2015.

[198] Chengcheng Wan, Muhammad Santriaji, Eri Rogers, Henry Hoffmann, Michael Maire, and Shan Lu. ALERT: Accurate learning for energy and timeliness. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 353–369. USENIX Association, July 2020.

[199] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. Residual attention network for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2017.

[200] Han Wang, Yuan Hong, Yu Kong, and Jaideep Vaidya. Publishing video data with indistinguishable objects. In *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*, 2020.

[201] Hongbign Wang, Xin Chen, Qin Wu, Qi Yu, Xingguo Hu, Zibin Zheng, and Athman Bouguettaya. Integrating reinforcement learning with multi-agent techniques for adaptive service composition. *ACM Trans. Auton. Adapt. Syst.*, 12(2), May 2017.

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

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

[204] Yanzhi Wang and Massoud Pedram. Model-free reinforcement learning and bayesian classification in system-level power management. *IEEE Transactions on Computers*, 65(12):3713–3726, Dec 2016.

[205] Yiding Wang, Weiyan Wang, Junxue Zhang, Junchen Jiang, and Kai Chen. Bridging the edge-cloud barrier for real-time advanced vision analytics. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[206] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, USA, 1995.

[207] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating*

Systems Principles, SOSP '13, page 244–259, New York, NY, USA, 2013. Association for Computing Machinery.

[208] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, Carlsbad, CA, October 2018. USENIX Association.

[209] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *Acm computing surveys (CSUR)*, 38(4):13, 2006.

[210] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 149–163, New York, NY, USA, 2003. Association for Computing Machinery.

[211] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252. ACM, 2018.

[212] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.

[213] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *SIGPLAN Not.*, 51(4):545–559, mar 2016.

[214] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multi-task cascaded convolutional networks. *CoRR*, abs/1604.02878, 2016.

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

[216] Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438. ACM, 2015.

[217] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in dash. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[218] Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, Pradeep Padala, and Kang Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 43(1):62–69, January 2009.

[219] Yuhao Zhu and Vijay Janapa Reddi. Greenweb: Language extensions for energy-efficient mobile web computing. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 145–160, New York, NY, USA, 2016. ACM.