CHARACTERIZING AND CONTROLLING PROGRAM BEHAVIOR USING EXECUTION-TIME VARIANCE

A Dissertation
Presented to
The Academic Faculty

by

Tushar Kumar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2016

CHARACTERIZING AND CONTROLLING PROGRAM BEHAVIOR USING EXECUTION-TIME VARIANCE

To my dear mother and grandmother,

for all their love, patience and sacrifice

over these long years.

ACKNOWLEDGEMENTS

There are many friends, colleagues and mentors over the many years whose patience, guidance, friendship and kindness touched me and taught me much about life, work and adventure.

Foremost, I'd like to thank with all my heart my advisor, Prof Santosh Pande, for believing in me, giving me a chance to be ambitious once again, showing me how to think big in research and for keeping me very well funded so I would have the freedom to think big. Thank you for giving me one heck of an adventure — intellectual and across Europe and North America.

I have the deepest gratitude for my co-advisor, Prof Sudhakar Yalamanchili, for offering me the best of opportunity, a chance to work on cutting-edge research in our startup and explore my full potential, and for being the nicest, most helpful person I have met.

I would like to thank my PhD committee members — Prof Patricio Vela, Prof Richard Vuduc, Prof Abhijit Chatterjee and Prof Umakishore Ramachandran. In particular, I would like to thank Prof Vela for devoting considerable time and interest in understanding my work in depth as the subject area expert and in helping me express the technical accomplishments much better.

From the early years of my graduate studies I'd like to thank my labmates and close friends Khawar Azad, Jimy Chang, Manuel Benet Navarro, Chris Wood, Weilai Yang, Himanshu Agarwal, Ankur Agrawal, Shridhar Reddy, Deepak Agarwal and Sriram Kishore Rallabhandi.

My years of hiatus at a startup introduced me to Hitesh and Reshmi Patel, Suresh Cheemalavagu, Ajay Jayaraj, Yogesh Chobe, Rick Copeland and Roger Dickerson who all became close friends. I am particularly indebted to Hitesh for introducing me to yoga, and for teaching the mind and body practice at a level I have not experienced with any other teacher.

On my return to graduate studies I was blessed to make many new friends and confidantes — Lakshmi Narasimhan Chakrapani, Pinar Korkmaz, Rodric Rabbah, Balasubramanian Seshasayee, Subramanian and Divya Ramaswamy, Nawaf Almoosa, Jeffrey Young, Veerdhaval Mahajan, Varadraj Vernekar, Stephen Lewis, Aakash Jariwala, Trang Thai, Ismail Faik Baskaya, Shafi and Adria Motiwalla, Manish and Surabhi Goyal, Partha Sarathi Chakraborty and Tanushree Ghosh, Shauvik Roy Choudhary and Tanushree Mitra, Vishakha Gupta, Danesh Irani and Zareen Kasad, Bhuvan Bamba, Vijay Balasubramaniyan, Dulloor Rao, Adit and Smita Ranadive, Raghav and Rakshita Vijaywargiya, Mukil Kesavan, Priyanka Tembey, Farhana Aleen, Kaushik Ravichandran, Martin Levihn, Jonathan Gladin, and Kangqi (Vincent) Ni. In particular, I'd like to thank my close friends and labmates Romain Cledat and Jaswanth Sreeram for our shared startup adventures (since the first startup was evidently not enough for me) and our shared creation, misery, joy and travel. The stimulating intellectual and research discussions with Nawaf, the wonderful time spent with Manish and Surabhi Goyal, the family-like bonds with Partha and Tanu, Shafi and Adria, and Shauvik and Tanu, the shared interest in coffee, food and dessert with Vishakha, Pinar and Nawaf, the late night working sessions fueled by coffee, juice and omelettes with Nawaf, Shauvik and Danesh, and the close bonds forged over tennis and beer with Yogesh, Jaswanth, Adit, Dulloor and Raghav will remain some of my fondest memories.

I would like to thank my fellow graduate students Jayaram Natarajan, Kangqi (Vincent) Ni, Girish Mururu and Vinit Deodhar for their considerable help critiquing my practice talk. They helped improve the final presentation immensely. I am deeply indebted to Partha, Shauvik, both the Tanus, Jeffrey and Nawaf for helping me so much during my last few visits to Atlanta and as I worked remotely to finish the thesis.

Finally, I would like to thank my family, in particular my mother and grandmother, without whose love, support and encouragement from an early age these academic pursuits would not have been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Immersive applications, such as computer gaming, computer vision and video codecs, are an important emerging class of applications with QoS requirements that are difficult to characterize and control using traditional methods. This thesis proposes new techniques reliant on execution-time variance to both characterize and control program behavior. The proposed techniques are intended to be broadly applicable to a wide variety of immersive applications and are intended to be easy for programmers to apply without needing to gain specialized expertise.

First, we create new QoS controllers that programmers can easily apply to their applications to achieve desired application-specific QoS objectives on any platform or application data-set, provided the programmers verify that their applications satisfy some simple domain requirements specific to immersive applications. The controllers adjust programmer-identified knobs every application frame to effect desired values for programmer-identified QoS metrics. The control techniques are novel in that they do not require the user to provide any kind of application behavior models, and are effective for immersive applications that defy the traditional requirements for feedback controller construction.

Second, we create new profiling techniques that provide visibility into the behavior of a large complex application, inferring behavior relationships across application components based on the execution-time variance observed at all levels of granularity of the application functionality. Additionally for immersive applications, some of the most important QoS requirements relate to managing the execution-time variance of key application components, for example, the frame-rate. The profiling techniques not only identify and summarize behavior directly relevant to the QoS aspects related to timing, but also indirectly reveal non-timing related properties of behavior, such as the identification of components that are sensitive to data, or those whose behavior changes based on the call-context.

CHAPTER I

INTRODUCTION

The execution-time variance of programmatic constructs is a largely untapped aspect of programs that offers unique insights for understanding program behavior and a unique approach for controlling program behavior. Immersive applications, such as computer gaming, computer vision and video codecs, are an important emerging class of applications that stand to benefit from the characterization and control of execution-time variance.

Immersive applications attempt to maximize the feature set expressed while maintaining a sufficiently smooth frame-rate. Their Quality-of-Service (QoS) requirements are often best-effort in nature, such as seeking soft-real-time frame-rates and the improvement of multiple application-specific QoS metrics in a balanced manner. Immersive applications are typically large C/C++/Java programs, which lack analyzable language semantics about behavior and timing. Their QoS behavior is typically a complex *emergent* property of the data-set and underlying algorithms. Consequently, QoS tuning becomes a tedious and ad hoc process for immersive applications. The gaming industry is famous for prolonged game-play testing where every possible game scenario is played out on various relevant gaming platforms and then manual tweaks to the game feature-set are made for each scenario and platform. Other application domains, such as video encoding and computer vision, typically a priori fix the feature-set for a limited use-case after extensive trial-and-error — for example, programmers may manually tweak algorithmic parameters until a sufficient frame-rate and accuracy/fidelity is achieved on a required video resolution and video content. The a priori fixed parameters fail to account for local variations in the application behavior over a data set during a single execution of the application and variations in the application behavior across data sets.

1.1 Contributions

1. We create new QoS controllers that programmers may easily apply to a wide variety of immersive application to dynamically tune application-specific parameters and keep QoS metrics at desired values over a single data set, across differing data sets and across differing execution platforms. These controllers combine system identification, adaptive control and optimal control with the particular properties inherent to immersive applications to make the problem of QoS control tractable, whereas traditional controller design techniques prove difficult to apply due to the nature of immersive applications.

2. We create new profiling techniques that rely on detecting variant behavior around repeatedly executed constructs in the application call-structure and summarize the most dominant variant behavior from across all levels of the functional granularity of the application. These profiling techniques are particularly suitable for identifying to the user the application components that are likely to impact the QoS in immersive applications, which is particularly difficult for traditional hot-spot profiling and static analysis to accomplish for immersive applications.

The two contributions help during different phases of the application QoS tuning process. Programmers may use our profiling techniques to discover what parts of their application need either offline or dynamic tuning to maintain QoS. Once the programmers develop an understanding of the application components that impact QoS, either with the help of our profiling techniques or just from their own application/domain knowledge, the programmers may choose to apply our controllers to perform the QoS tuning automatically over a wide-variety of platforms and use-case scenarios.

1.2 Dynamic QoS control

As one part of this work, we create QoS control techniques for immersive applications. The QoS requirements of immersive applications are typically straightforward to express as a variance-minimization problem, creating an opportunity to apply standard model-estimation and control techniques for the automated tuning of QoS during application execution. Immersive applications frequently use parametrically scalable algorithms, where the dynamic tuning of the algorithm parameters would allow control over the expressed QoS behavior of the application, with regards to the frame execution time, accuracy of results, level of detail, etc. However, immersive applications pose the following challenges to the direct application of the standard estimation and control techniques — *i)* a monolithic C/C++/Java implementation of the applications with no analyzable properties for behavior and timing; *ii)* high sensitivity of the application behavior to the data-set, often with rapid variations over the data-set, making the a priori derivation/estimation of a fixed model infeasible; *iii)* behavior that is mostly an emergent property of the data-set and the algorithms, often making the construction of parametric models very difficult, even for application-domain experts; and, *iv)* existence of no common modeling framework with well-defined behavior properties for immersive application.

We create a QoS-control problem formulation that recognizes additional properties common to immersive application. We refer to these properties as the domain assumptions of the class of immersive applications. We create two QoS controllers that rely on the domain assumptions to augment the standard estimation and control techniques, making QoS control tractable for immersive applications. The first controller is *uni-variate* with a very light-weight adaptive-integral control strategy. The second controller is *multi-variate* and builds on estimation and optimal control techniques. The controllers are probabilistic and best-effort in nature — the better a given application satisfies the domain assumptions, the greater the probability that its QoS requirements will be satisfied by the controllers and with increasing tightness.

The QoS controllers are designed to allow an arbitrary immersive application to adapt to a wide variety of operating conditions (such as compute platforms with differing capabilities) and a wide choice of optimization goals for the QoS metrics, so long as the domain assumptions remain valid. We greatly simplify the achievement of QoS for immersive applications by only requiring the programmers and domain experts to verify that their application satisfies the domain assumptions, which is often a simple and intuitive process. In contrast, the current state of the art requires them to either manually tailor algorithmic parameters to only a narrow range of data sets and operating conditions, or have considerable controls expertise to create a custom solution for their particular application, with the vast majority of practitioners settling for the former.

1.3 Profiling

As a second part of this work, we create offline profile-analysis techniques to characterize the variant behavior around repeatedly invoked constructs in programs. Variance characterization creates new opportunities for the programmer to tune program behavior. In particular, variance characterization reveals the program components most likely to impact the typical QoS requirements of immersive applications, and has the ability to detect related behavior across program components due to correlations in their execution-time variance, which is beyond the capabilities of static analysis and hot-spot profiling.

We create the Call-Context Variance Analysis (CCVA) technique to demonstrate the role call-context plays in determining the locations of variant behavior in the program call-structure, and then the more general Dominant Variance Analysis (DVA) technique that succinctly captures the structure of the variant behavior exhibited by an application. DVA is capable of relating variant behavior across application components and finding underlying causes of variance in large C/C++/Java applications where the programming language provides no behavioral or timing semantics, limiting the utility of static analysis. The large code-base and the predominantly data-dependent behavior of immersive applications also limits the utility of regular hot-spot profiling techniques. Hot-spot profiling would simply identify the top-level application functions as being the most compute-intensive, while the performance tuning opportunities for frame-rate and improved QoS typically span functions at multiple levels of granularity. In contrast, DVA will examine the repeated execution of the application components at all levels of functional granularity (such as a frame, and various levels of sub-block processing within a frame in a video encoder) and identify variant behavior at each level. DVA will determine if behavior at one level of granularity is the principal cause of variance at another level. We create a new program representation called Variance Characterization Graph (VCG) capable of flexibly and succinctly extracting structure from the full program call-graph. DVA uses the VCG representation to capture multiple instances of related variant behavior within a common structure for the user to examine. Finally, DVA has the ability to summarize numerous instances of similar behavior occurring across the program call-graph into a succinct VCG representation.

1.4 Thesis Statement

This thesis proposes novel adaptive control techniques capable of tuning the QoS of frame-oriented immersive software applications for which generic controller design techniques and generic programmatic techniques fail to be effective due to these applications' rapidly time-varying and data-dependent nature, and proposes a novel profiling technique capable of summarizing the dominant variant behavior of these applications from across all levels of their functional granularity, in a manner particularly suited for the QoS optimization of typical immersive applications.

1.5 Organization

The remainder of the thesis is organized as follows. Chapter 2 motivates why the new profiling and QoS control techniques are needed to overcome the unique challenges posed by immersive applications. Chapter 3 discusses related work. Chapter 4 describes the uni-variate QoS controller. Chapters 5, 6 and 7 describe the multi-variate QoS controller. Chapter 8 describes the Call-Context Variance Analysis profiling technique. Chapter 9 describes the Dominant Variance Analysis profiling technique. Finally, Chapter 10 concludes the thesis and discusses future work.

CHAPTER II

MOTIVATION

2.1 Nature of Immersive Applications

Immersive applications such as gaming, multimedia and computer vision are becoming prominent compute-intensive applications on consumer desktops and mobile devices. Immersive applications are interactive by nature, and place a premium on using a platform's compute resources to the maximum in order to create as engrossing and visually stunning an experience as possible for the user. Due to their interactive nature, these applications have a bounded window of time (a "frame") to update the simulated world state and its visual rendering based on immediate user inputs. Therefore, the QoS tuning goal for these applications is to pack the most sophisticated features possible into a frame while maintaining a sufficiently smooth and high frame-rate [1]. A video encoder may need to compress a live video-stream in real time without dropping too many frames in the presence of per-frame variations in processing time. The programmers may also desire that an application-specific feature achieves a certain quality, such as the achievement of sufficiently sophisticated Artificial Intelligence (AI) for bots (simulated characters) in a video game without compromising the frame-rate too often with excessive AI computations, essentially striking a balance between AI quality and having a smooth frame-rate.

2.2 Challenges in Tuning QoS

There are four main challenges that programmers face in using existing formal methods to tune the behavior of immersive applications.

The first challenge is that immersive applications are implemented as *monolithic* programs using general-purpose C/C++/Java development flows which provide significant productivity advantages in developing a large complex application. Unfortunately, the use of general-purpose programming languages fails to provide sufficient analyzable information about the execution properties of the application that languages with specialized semantics, such as for real-time [2, 3] or for streaming [4], could.

The second challenge is that the QoS of immersive applications is highly sensitive to the nature of the data being processed and the nature of data may change rapidly during a single execution of the application. An MPEG2 *encoder*, for example, exhibits very different frame times on different parts of the same video stream being compressed, as illustrated by the sequence of frames in Figure 2.1. *Note that this is the case even when the application has exclusive use of system resources, i.e. the frame-time variation is a consequence purely of the characteristics of the application itself with no resource contention issues involved.* Figure 2.2 shows similar variations for the MPEG2 *decoder* application [5].

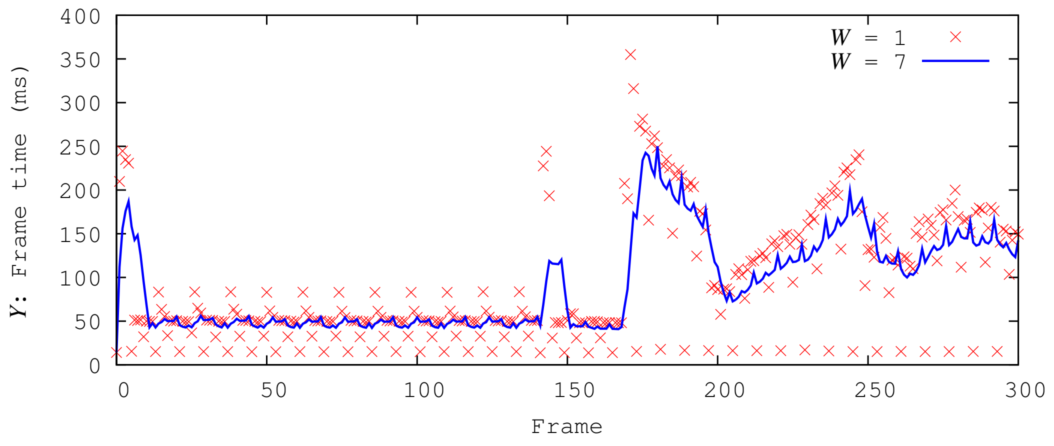


Figure 2.1: A sequence of frame-encoding times for the MPEG2 encoder. $W = 1$ shows the instantaneous frame-times. $W = 7$ shows a moving average of the previous seven frame times.

The primary reason for the frame-time variability in the MPEG2 encoder is that the motion estimation algorithm may perform searches of very different sizes across frames depending on how quickly a matching block is found in adjacent video frames [6]. Such data-dependent variability makes the use of any fixed setting of the application features sub-optimal. A secondary reason for the frame-time variability is the use of Group-of-Pictures (GoP) by the video standards [7] — for example, every sequence of seven frames will consist of a pre-determined mix of intra coded, predictive coded and bi-predictive coded frames (called I, P and B frames, respectively). The computational load of motion estimation varies substantially across I, P and B frames, reflecting in substantially different frame-times for adjacent frames in a GoP, as seen in the $W = 1$ plot in Figure 2.1. Since the duration of the GoP is known a priori, and because the instantaneous frame-time variations are not noticeable to an interactive user, the QoS goal for video encoders/decoders typically involves keeping the *moving average of frame-time* within a desired range. The corresponding moving averages over the prior seven frames (the GoP length) are shown in the $W = 7$ plot in Figure 2.1. Despite the averaging, the changing nature of the raw video data produces substantial changes in frame-time over longer sequences of frames, as shown for $W = 7$.

Therefore, QoS controllers designed a priori on fixed application models or on "representative" data sets would fail to account for the large variations possible within data sets (such as the changing amount and speed of motion over the video sequence). Further, different data sets may differ from each other significantly, such as a low versus a high video resolution, producing still larger variations in behavior that would make offline designed controllers ineffective.

The third challenge is that the behavior of immersive applications is largely an emergent property of the algorithms and data sets. Immersive applications tend to be large and complex, involving multiple algorithms that interact and influence each other. The nature of the data may affect the decision logic of an algorithm — say, the number of loop iterations performed until some accuracy threshold is achieved. In a video game, a path-planning algorithm (such as A* [8, 9]) may explore a large search-space of possible paths until a viable solution is found. The path determined may impact other algorithms, such as the AI of a bot that has to follow the path. The game world state, the user's inputs, and the resulting behavior of the algorithms would typically be hard to factor into any simplified model (say, a parametric model), suitable for the construction of model-based controllers (say, using

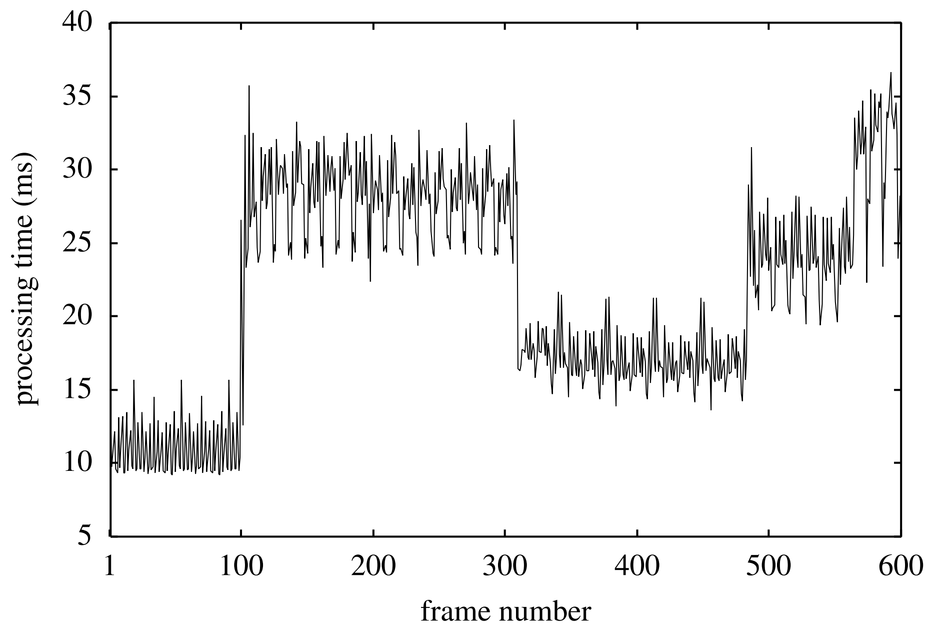


Figure 2.2: Variations in per-frame decoding time for the MPEG2 decoder *(Wüst et al. [5]).*

adaptive control techniques) to tune the QoS of immersive applications (frame-time, accuracy of results, etc.). We contend that new techniques that directly account for the nature of immersive applications are required.

The fourth challenge is that there is generally a lack of modeling frameworks suitable for the broad range of immersive applications. Some specialized applications written by experts, such as video codecs, may have been systematically modeled and their execution-time properties studied in depth, such as with MATLAB or high-level models before the actual C/C++/Java implementation was done [10]. In contrast, the vast majority of immersive applications (e.g., games, computer vision applications) get to have their behavior studied only after they have been implemented [11]. Frequently, various third-party libraries of functionality are used in the application [11, 12, 13, 14]. Consequently, the application programmers may know what QoS metrics they care about and may have an idea of what algorithmic knobs or parameters could be adjusted to impact the QoS metrics, but would typically lack a sufficient understanding of the interactions of the algorithms and of the libraries to understand how the knobs/parameters actually impact the QoS metrics. Instead of expecting the programmer to understand *how* the knobs/parameters impact QoS, we would like in our approach to place a much lighter burden on the programmers — identify which knobs/parameters *could potentially impact* QoS.

2.3 Challenges in Characterizing QoS Performance

Programmers need to understand the behavior of immersive applications in order to tune the QoS performance. In Section 2.2 we motivated that immersive applications are large complex programs, they exhibit behavior differences both within and across data sets, the application behavior is an emergent property of the algorithms and the data set, and there is often a lack of suitable modeling frameworks that programmers may use to understand the application behavior. Here, we motivate the properties needed by a profiling technique to become effective at characterizing the behavior of immersive applications.

1. Given the diversity of immersive applications, it is important that the profiling techniques be *broadly applicable and not rely on application-specific properties.*

2. Given that the functional components in an immersive application are expected to exhibit varying behavior over a given data set, the profiling techniques must be capable of characterizing the *range of behavior of a component.*

3. Given the large code base of an immersive application, the behavior must be characterized at *multiple levels of functional granularity.* Further, it should be possible to present the profiling results to the user in a highly summarized manner, yet without losing the details of lower-level components when those are key to understanding the application behavior (infeasible for hot-spot profiling, which will tend to emphasize the top-level components).

4. Given that immersive applications tend to be frame-oriented, and their components often process streams of data (such as sub-blocks within a video frame), the characterization of *repeatedly invoked components* is paramount.

5. Given that general-purpose programming languages, such as C/C++/Java, tend to express the application hierarchically in terms of functions, it is important to preserve

information about the call-structure under which particular behaviors occur (*the call context of the behavior*), as the programmer will rely on the call structure to relate the profiling results to the application code-base.

6. The profiling technique must *summarize similar behavior* across the program call-tree as much as possible, for varying degrees of "similarity" desired by the programmer.

7. Given the large size of the application, profiling results should use the call-context in reporting results only when the behavior of components varies based on their call-context, and only use the *minimum call-context necessary for distinguishing differing behaviors.* Further, similar behavior of a component under multiple call-contexts should be merged together and identified collectively using the minimal amount of call-context that can distinguish from other call-contexts with dissimilar behavior.

8. Given that an immersive application should be characterized at multiple levels of granularity, it would be very helpful to the programmer if the *behavior found at different levels of granularity could be related*, particularly with regards to identifying the *underlying causes of behavior* across components.

Most traditional profiling techniques are oriented towards minimizing program execution time, such as profiling for hot-spots or hot-paths. Other techniques detect phases in an application's execution where each phase has distinct characteristics with regards to stall cycles, cache miss-rates and instructions-per-cycle executed. Another set of techniques, collectively called *worst case execution time* (WCET) analysis, attempt to place bounds on the execution times of components in safety-critical applications. While all these techniques aid the programmer in debugging different aspects of an application's performance, they lack the characteristics desired for profiling immersive applications.

To achieve the above mentioned desired properties, we are required to innovate on the following ideas.

- We propose the notion of *dominant behavior*, to filter out what behavior is important across all levels of granularity, whether exhibited by a single component in the call-structure, or dominant only in aggregate across the application call-structure.

- We create mechanisms to determine the sensitivity of behavior to call-context, and mechanisms to minimally represent the distinguishing call-context for the behavior.
- We create a structural representation to express the context sensitivity of behavior whenever context matters, and for capturing relationships between behavior at multiple levels of granularity.
- We develop statistical mechanisms to determine "similarity" of behavior, particularly to aid the summarization of behavior.
- We create a structural representation that can merge and summarize similar behavior across the application call-structure, including the relevant call-contexts and the cause-effect relationships.

We develop the above ideas for behavior defined in terms of the *mean and variance* of the *execution-time* of application components. However, the demonstrated ideas can be generalized to diverse aspects of the application execution, such as the mean and variance of the number of memory accesses, cache misses, network accesses, etc. By defining behavior in terms of mean and variance, we can not only compare the behavior of two components for similarity, but can also flexibly combine similar behaviors into either a collective mean and variance, or retain the details of the original behaviors as a Gaussian mixture model (GMM). Use of the mean and variance provides representational and computational simplicity when the spread is small relative to the mean, while the GMM provides generality to approximate arbitrary probability distributions. Finally, mean and variance, and probability distribution models such as GMMs are natural choices to summarize variations in behavior over the multiple invocations of a repeatedly-invoked component.

Chapter 8 introduces the CCVA profiling technique to distinguish behavior based on the minimal call-context. Chapter 9 describes the DVA profiling technique that introduces the VCG structural and behavior representation. DVA combines the ideas of dominant behavior, the merging of similar behavior, and the minimal distinguishing call-context to operate on the VCG representation.

Benefits for Application Performance Tuning. With immersive applications, the proposed profiling techniques can identify and characterize application components whose variant behavior is likely to impact the QoS of the application, thereby providing the programmer with succinct and pertinent information to guide application tuning.

Non-immersive applications also benefit from variance analysis. During the parallelization of a sequential application, programmer expertise or compiler analysis is traditionally used to identify application components that are *safe* to execute in parallel. Profiling techniques are used to determine which groups of components are *beneficial* to execute in parallel with regards to producing speedup gains. However, traditional hot-spot profiling fails to take the variations in the execution times of components into account. A safe set of components, with each component exhibiting a large and roughly equal execution time on average, may appear highly suitable for parallel speedup. However, if the execution time of these components is highly variant in an out-of-phase manner, then in any parallel execution of the components, one component is likely to have a much larger execution time than the other components, thereby significantly reducing the parallel speedup. Therefore, variance analysis is beneficial for determining which candidate safe set of components is likely to produce a large parallel speedup.

It is important to study variance at all levels at which it occurs in the program. For example, in a packet-routing application, it may be important to characterize the variance in both the per-packet routing time at each port, as well as in the overall packet-routing rate for the router. Such an analysis would help designers/programmers determine how much buffer space should be provided at the port for transmission/reception, and the amount of global memory needed for temporarily storing received packets. This can also facilitate a trade-off between the sizes of the port buffer space and the global memory. A video-compression application would similarly benefit from a characterization of the variance in frame-compression time at the high level, and the per-image-block variations in processing time within a video frame for diagnosing which types of image blocks or types of processing at the low level are responsible for the high-level variance in frame-compression time.

CHAPTER III

RELATED WORK

3.1 Dynamic QoS Optimization

Control theory There is extensive literature on the design of controllers for *linear-time-invariant* (LTI) *systems* [15], which is not directly applicable due to the non-linear and highly time-varying nature of immersive applications. In particular, one common technique called the Ziegler-Nichols method [16, 17, 18], tunes the gain parameters of a PID controller to make the controller suitable for use with an LTI system with an unknown model. The tuning technique involves offline excitation of the system with particularly crafted inputs, such as sinusoids that are gradually varied in frequency and the system response observed to determine the controller gain parameters. The extensive offline tuning involved is not suitable for use with immersive applications, where the application behavior can vary significantly across data sets and even over a few frames within a data set.

Adaptive control [19] deals with systems with partially-unknown or time-varying characteristics. The well-established adaptive techniques are classified as model-reference adaptive control (MRAC), self-tuning regulator (STR), gain scheduling (GS) and model-identification adaptive control (MIAC). Broadly speaking, adaptive techniques are intended for dealing with uncertainty in unknown constant or slowly changing parameters. With immersive applications, the behavior can change rapidly and by large amounts, requiring additional innovation and customization over the general adaptive techniques. Generally, *robust control* techniques are used to deal with disturbances and rapid parameter variations. Our controllers incorporate parameter estimation techniques that build up metrics over long sequences of frames to create robustness, while retaining the ability to detect rapid changes in behavior by relying on the domain properties. However, as one major contrast, our controllers are designed to improve the *probabilistic properties* of deviant application behavior, rather than the traditional goals of achieving Lyapunov stability [20, 21, 22, 23], improving settling time, or minimizing overshoot [24]. As a second major contrast, immersive software applications do not exhibit the typical general properties of physical systems that can be relied upon to achieve convergence, such as the dissipation of total system energy that would ultimately lead to equilibrium. Instead, we identify *new properties of immersive applications*, which we refer to as the domain assumptions, that our controllers rely upon to control application behavior. As a third major contrast, the control law in several traditional adaptive approaches relies on components of the plant model to exhibit various mathematical properties — e.g., the positive definiteness of a state-transform matrix. Instead, our controllers must work with extremely noisy model estimates over insufficient online data where it is not possible to reliably establish mathematical properties that would hold true for the underlying application behavior. The controllers rely on the domain assumptions, in particular certain probabilistic properties, to achieve effective control under these circumstances.

Next, we contrast against the specific adaptive techniques. MRAC [25, 26] and STR [27, 28, 29] require *parametric models* of the application to be available, usually in an LTI form or in particular non-linear forms. When a plant model is not already available, the control engineer assumes a model form, and experimentally fits/verifies the model (offline model identification), leaving some parameters to be *adaptively estimated* at runtime. As an example of STR applied to software applications, Lu et al. [30] experimentally validate that the QoS of their non-linear application, a proxy server cache, can be approximated as a second-order parametric LTI model whose parameters change periodically. A model estimator periodically re-estimates the model parameters from system performance, and pole-placement techniques produce a new controller for the estimated model. The MRAC adaptive approach explicitly tracks the error between the outputs of a *reference plant model* and the actual plant, and adjusts controller parameters to drive the error to zero. Both STR and MRAC require a plant model of the application to be available. Consequently, MRAC and STR are not well-suited to immersive applications where it is generally not feasible for programmers to provide a parametric plant model, due to the rapidly time-varying and emergent nature of the application behavior.

Our uni-variate controller [31] broadly falls in the category of GS [32, 33, 34]. The uni-variate controller directly adjusts the feedback gain based on runtime metrics estimated by the controller, with no application behavior model provided a priori or estimated at runtime. The runtime metrics characterize properties that every immersive application must possess, based only on the domain assumptions. In particular, the domain assumptions help identify the granularity of QoS deviations that would be perceptible to the interactive user of the application, and quantify the gain adjustment required to suppress the perceptible QoS deviations. In contrast, the traditional GS techniques make assumptions about the form and properties of the underlying unknown non-linear model [33]. These form and properties assumed are not suitable for modeling immersive applications, and in particular, do not correspond to the domain assumptions of immersive applications. Therefore, we create our uni-variate controller as an alternative approach to gain scheduling based on the domain assumptions.

The multi-variate controller falls in the category of MIAC [35]. Broadly speaking, the MIAC approach is used when there is insufficient knowledge to create an a priori model, such as with MRAC, but there is enough knowledge about the system to estimate a model. As the main contribution, our controller relies on domain-specific knowledge about immersive applications to make model identification tractable. The controller performs online model identification to linearly approximate a model of the *current* application behavior and uses optimal control techniques to construct a regulator dynamically. The controller uses a *hybrid* of the *certainty equivalent adaptive* and *cautious adaptive* approaches guided by the domain assumptions to achieve the dual objectives of replacing the active model with a new model as soon as the application behavior changes, as well as discarding freshly estimated "noisy" models when the existing active model is estimated to be more accurate. Additionally, the domain assumptions help define the problem structure and the cost functions involved in the model identification problem and the regulator construction problem.

Self-adaptive software There is recent work on applying adaptive control theory in *a generic manner* to a wide-variety of software applications. Brun et al. [35] survey the increasing need for self-adaptive feedback loops in software applications and the various adaptive techniques that could be applied. Hoffman et al. [36] have similar goals as our uni-variate controller — check point an application's execution-rate on key events referred to as "heart beats" (e.g., every loop-iteration or frame) and dynamically adjust a single parameter to bring the heart beats within a desired range. The adjusted parameter could be an internal application parameter that directly adjusts application algorithms or an external parameter (such as number of cores given to the application). Extensions of the technique allow the power-states of the processor to be dynamically adjusted (DVFS) to achieve the desired "heart rate" [37]. Most recently, Filleri et al. [38] show how adaptive controllers can be created for software application with control-theoretic guarantees related to performance (tracking, settling and overshoot) and robustness. These works have the same goal we have — make it easy for non-experts to incorporate QoS control into their software applications. However, all of these techniques *expect an offline model* for the application behavior to be available for the controller design step, with the model possibly constructed via offline model identification on suitably representative reference data-sets. In contrast, both our uni-variate and multi-variate controllers *explicitly avoid offline model construction* as immersive applications are shown to be highly variant, often with a rather large range of behavior possible within a single application execution, and with behavior varying significantly over data sets. For these reasons, the behavior cannot be captured effectively by a fixed linear model, or even a general-form parametric model (which the cited self-adaptive software techniques do not attempt). Consequently, for immersive applications, our controllers significantly outperforms these more generic techniques. In particular, our controllers are capable of tracking the frame-rate (or any other QoS outcome) *within a tight window* (instead of just being effective at keeping the outcome above a desired minimum), and are capable of tuning the *more instantaenous behavior* of the application. For example, with video encoding applications, our controllers track the moving average of the last 7 frames ([31], Chapter 4 and Chapter 7) instead of the last 40 frames by Hoffman et al. [36].

Additionally, none of the prior self-adaptive software techniques *control multiple application parameters or explicitly track multiple outcomes*, which is a capability our multi-variate controller is the first to achieve.

Therefore, our reliance on properties specific to immersive applications allows our controllers to achieve much tighter and more instantaneous control on immersive applications and without the use of offline models (which are typically infeasible for immersive applications). However, the prior techniques are not limited to immersive applications, and may be very effective at the control of other software applications, for example, applications whose behavior is unknown but not highly time-variant, and where the application behavior on reference data-sets (for training) is highly representative of behavior over any online data-set. As a final contrast, some of the prior techniques provide control-theoretic guarantees on the constructed controllers, which our controllers do not. However, the guarantees only apply when the offline model is a sufficiently accurate representation of the application behavior (with some additional leeway possible with the incorporation of robust control techniques). Since any fitted model is likely at best a very approximate or a very temporary representation of an immersive application's behavior, the control-theoretic guarantees on the model fail to translate to guarantees on the application behavior. Instead, our controllers are inherently *probabilistic best effort* in nature, as the domain properties of immersive applications that we have identified translate easily to probabilities, and implicitly allow the QoS goals to be met more tightly and on a greater fraction of the frames when the application behavior more strictly satisfies the required domain assumptions. By this approach, our controllers possess the following performance and robustness properties for immersive applications in contrast with the prior techniques.

1. High frequency of application frames that satisfy the QoS goal if the specified QoS goal is attainable.

2. High frequency of application frames that minimize the QoS error if the specified QoS goal is unattainable.

3. Robustness in the form of graceful degradation of QoS performance when the application behavior does not strictly satisfy the domain assumptions.

4. Robust estimators that filter noise over a long sequence of frames, while staying responsive to sudden behavior changes due to the reliance on the domain assumptions.

Real time Mejia-Alvarez et al. [39] allow optional tasks in the task-graph representation of an application. The optional tasks are executed for improved *total system value* whenever there is *slack to deadline*. In contrast, the tuning of the QoS of immersive applications requires a feature setting to be picked *prior* to frame invocation, needing the controller to *predict* which feature settings will likely achieve the desired frame time. The feature settings may impact the execution time of algorithms that execute early in the computation structure of the frame. Such algorithms would fail to be tuned if optional work could be discarded only towards the end of a frame's computation. More generally, complex applications such as games and the increasingly sophisticated computer vision use-cases, typically employ a symbolic representation of the world state [40, 41, 42, 43]. The update of a game object in the world state may trigger the invocation of particular algorithms depending on the attributes associated with the object. Therefore, a particular algorithm of interest may be invoked on-demand multiple times over the frame computation, with no possibility of grouping all the invocations into a single or a few tasks that could be tuned or discarded together. Therefore, comparatively few immersive applications would be structured to benefit from the dropping of optional work towards the end of the frame. Additionally, as explained in Chapter 2, immersive applications are typically not amenable to implementation as real-time task-graphs. More generally, task-based techniques may only re-order, schedule and eliminate parts of the computation; they cannot alter the inherent computation, which our controllers can do by tuning the feature settings every frame. Lastly, our controllers may only be applied to applications that (approximately) satisfy the domain assumptions of immersive applications, while task infrastructures may have different requirements, such as periodic execution of a set of tasks, that make them more suitable for other non-immersive applications.

Cucinotta et al. [44] and Block et al. [45] use feedback control to fine-tune the allocation of compute resources to periodic soft-real-time tasks that can exceed deadlines. Their methodology requires the prediction of task workloads. They assume that subsequent task workloads can be predicted when suitable stochastic models are available. When stochastic models are not available, they assume that either a moving-average model or a PI controller can adequately predict task workloads based on previously observed workloads. Block et al. additionally allow tasks to have multiple *service levels*, each delivering a user-specified level of QoS with differing workloads. They require the specification of a function that translates workloads between the service levels. In contrast, the data-dependent and time-varying nature of immersive applications typically *precludes the existence* of a well-defined relationship between workload (frame time) and service level (feature setting), much less the ability of the programmer to *specify a function* capturing the relationship. Even the QoS delivered for a fixed service level (feature setting) tends to be a time-varying entity with immersive applications. Further, Block et al. require the user to specify gain parameters for their PI controller. Overall, it is infeasible for the users or the programmers of immersive applications to provide much of the above required specifications, either because of the applications' time-varying nature or the difficulty in determining analytical/stochastic models (Chapter 2). Instead, our uni-variate adaptive controller derives and fine-tunes gain parameters directly from the observed application response, while our multi-variate controller determines appropriate frames during the execution of the application to reconstruct a model of the application behavior ("system identification") to determine how best to adjust the feature setting based on the current behavior of the application. As a final difference, these works tune system resource allocation and scheduling to improve missed soft-real-time deadlines without directly changing application properties; they don't attempt to tune application-specific algorithmic parameters to improve application-specific QoS metrics.

de Niz et al. [46] schedule and allocate resources *between* applications for QoS gains. Our controllers are complementary to their approach — the controllers tune the QoS of an *individual* application by tuning parameters that impact the application's algorithms. As future work, the two aspects can potentially be combined to allow the joint tuning of the algorithmic parameters and the resources allocated across applications to achieve the best QoS across concurrently executing applications.

Application-specific techniques Work by Roitzsch et al. [47] and Huang et al. [48] on predicting the frame-execution times of an MPEG2 *decoder* relies on extracting the video control sequence from the bitstream using light-weight techniques. Since the control sequence precisely dictates the set of computations needed for decoding each frame, it is possible to accurately predict the per-frame execution time. In contrast, we target the MPEG2 *encoder*, that processes raw video-frame-image data, and no simple pre-computed metrics are available to assist in the prediction of per-frame processing time. Work by Wüst et al. [5] dynamically scaled the complexity of an MPEG2 *decoder* among four levels based on markov decision processes and reinforcement learning. However, the models and the metrics in that work were created specifically for the MPEG2 decoder. In contrast, our work is not application-specific and benefits the broader class of frame-oriented applications, for example the MPEG2 encoder whose frame times cannot be cheaply predicted ahead of time.

The more recent video encoding standards, such as H.264, employ rate-distortion optimization (RDO) [49, 50], which often involves application-specific feedback control to dynamically tune QoS. The new standards allow the encoder to dynamically select from a large space of possible block-sizes, motion-vectors and quantization step-sizes to achieve the best trade-off between picture quality and compression bit-rate, based on the properties of the raw video content. The larger space of possibilities allows the encoder to potentially find much better macro-block matches and customized quantization step-sizes for the transform data that result in lower residual error (the "distortion"), that would require fewer bits to encode. However, additional bits must be expended to represent the more detailed possibilities for the motion vectors, block sizes and quantization steps (the "rate"). Often a feedback loop is employed to determine whether increasing the rate is currently justified by a sufficient reduction in the distortion. Therefore, RDO may involve feedback control carefully crafted by video experts after careful modeling for a single purpose. In contrast, our two controllers can be be very widely deployed by programmers who are not experts to model and craft their own controllers for their particular applications. Most importantly, many immersive applications exhibit complex, dynamically changing behavior that is often very hard to model, even for experts.

Function Estimation Our uni-variate controller does not require any function estimation. The multi-variate controller needs to rely on some type of function-estimation techniques to discover the application's multi-variate input-output relationship based on only very limited sampling of the application's response characteristics. Linear least-squares estimation (LLSE) [51] is a very commonly used function-estimation technique capable of fitting a *linear model* even on relatively few data samples. LLSE is tolerant to significant "noise" in the sample data (in our case, noise corresponds to all transient behavior in the application that is not controllable through the available control parameters). Radial-basis-function networks (RBFN) [52] are considered universal approximators of arbitrary *non-linear multi-variate continuous functions* on a compact subset of $\mathbb{R}^n$. That is, the coefficients in an *appropriately structured* and *sufficiently large* RBFN can be trained to fit any given non-linear continuous function defined over a bounded domain.

When the estimation's objective is to *interpolate* a function over a *grid of sample points*, RBFNs are considered both straightforward to apply and highly accurate [53, 54]. However, given that our objective is to achieve effective QoS control over an application with rapidly time-varying characteristics, our controller can sample *only a few points* from a high-dimensional input space based on which the controller must adequately model the application behavior. Therefore, determining an *accurate model* is not a feasible objective for our controller. Instead, our primary objective with function estimation is to determine the *sensitivities* of various outputs to individual inputs. Therefore, in our controller, we have chosen to use LLSE to fit a linear model to the observed application behavior, despite the potential of RBFNs to model arbitrary application behavior. The justifications for this choice are as follows: **i)** RBFNs require model fitting to be performed in a higher-dimensional space [55] compared to the dimensionality of the sample data to achieve a good fit, requiring estimation of a much larger set of coefficients compared to performing a linear fit with LLSE, **ii)** for a given dimensionality for model-fitting, RBFNs employ a larger number of "structuring parameters" to define the parametric form of the non-linear model compared to a linear model that will be estimated using LLSE, **iii)** feedback control has the potential to work well with even approximate models (such as a linear model) of the plant to be controlled, since feedback control can continue fine-tuning the control inputs over multiple control steps based on the observed response of the plant, **iv)** we have identified special characteristics (common across applications in our target domain) that would allow *linear approximations* of the application's non-linear behavior to be sufficient for achieving good feedback control, and **v)** optimal-control theory provides well-developed controller design techniques for linear system models, such as linear quadratic regulators (LQR) [56], making the use of controller design techniques over arbitrary non-linear models [57] unnecessary for our purposes.

Hence, we expect to achieve good control performance at a much lower computational cost by fitting approximate linear models, compared to fitting more accurate non-linear RBFN models.

3.2 Offline Characterization Techniques

Existing application-profiling techniques look for program hot-spots and hot-paths [58, 59, 60]. These techniques attempt to find performance bottlenecks in an application, and do not attempt to characterize patterns of variant behavior.

Calder et al. have used statistical techniques to characterize large-scale program behavior in terms of just a few recurrent intervals of code [61], and to identify phase-change points during the dynamic execution of a program [62]. However, their work does not attempt to characterize the variant behavior in terms of the *functional decomposition* of the application. In particular, they seek out intervals [61] consisting of closely-matching sets of dynamic basic-blocks. The behavior captured by these intervals does not directly relate to the behavior of functions. The behavior of the intervals is also not sensitive to the call context under which the behavior occurs.

Variability characterization curves (VCCs) [63, 64] and approximate VCCs [65] have been used to characterize the variability in the workloads of multimedia applications. Such analysis techniques require domain-specific knowledge of the application before they can be applied. Similarly, there are custom techniques for improving the QoS of a very limited type of applications, such as the techniques by Roitzsch et al. [47] that develop a higher-level model of a generic MPEG decoder, and use this model to predict the video-decoding times with high accuracy. In contrast, our framework characterizes the application's variant behavior in a completely domain-independent manner, with no assistance from the user.

For applications written using formal real-time abstractions such as tasks and deadlines, there are established formal techniques [66, 67] that analyze the real-time characteristics of the application and enforce the real-time requirements. For monolithic applications written without the use of these abstractions, our framework is unique in its ability to characterize their soft-real-time behavior.

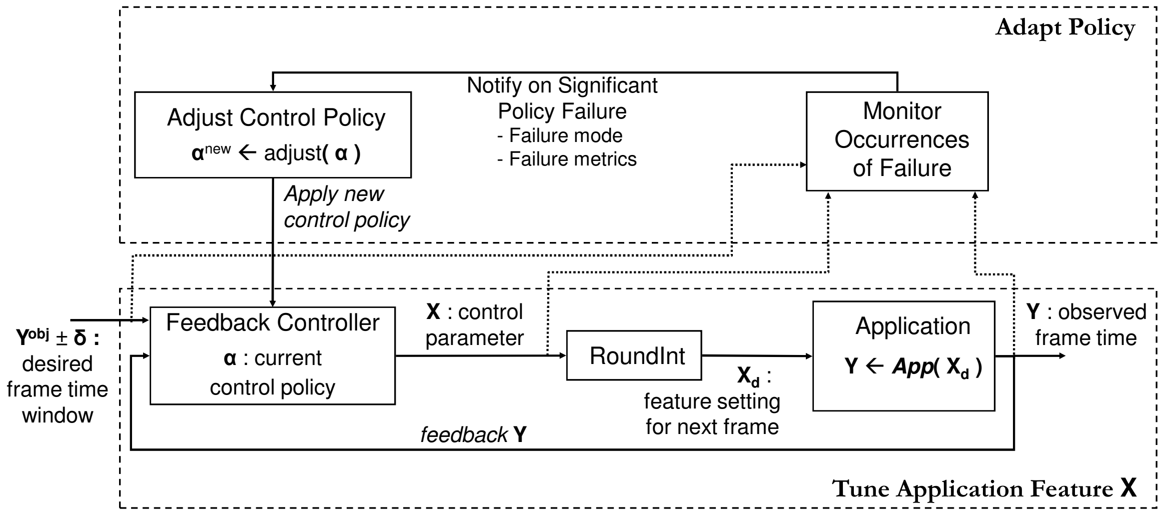
Worst-case-execution-time (WCET) [68] is an analysis methodology applicable to monolithic applications, and has been incorporated into commercial products such as those from AbsInt [69]. However, for non-safety-critical, compute-intensive applications like gaming and video, the knowledge of the *likely range* of real-time behavior is more important for driving design-optimization than the knowledge of the worst-case behavior. The likely range (detected by our technique) can be substantially removed from the worst case, thereby diminishing the utility of characterizing the worst-case behavior for such applications.

CHAPTER IV

UNI-VARIATE QOS CONTROL

In this work [1, 31] we develop a QoS controller that caters to the unique nature of immersive applications. We design an *adaptive feedback controller* based on a *system-identification* strategy. The controller is intended to be incorporated into applications to dynamically adjust a single algorithmic parameter of the programmer's choosing to control the QoS of a single programmer-identified QoS metric, often the frame-rate. The remainder of the chapter assumes that the QoS metric is frame-rate, even though the technique is not limited to frame-rate.

A system-identification based adaptive controller does not require a model of the application and is also tolerant to dynamically changing application characteristics. The adaptive aspect of the controller detects when the current control scheme is failing to sufficiently achieve the specified frame-rate objective, and adjusts the feedback-control policy accordingly. The controller consists of two layers, as illustrated in Figure 4.1.



Adapt Policy
Notify on Significant Policy Failure
- Failure mode
- Failure metrics
Adjust Control Policy
$\alpha^{new} \leftarrow$ adjust(α)
Monitor Occurrences of Failure
Apply new control policy
$Y^{obj} \pm \delta$: desired frame time window
Feedback Controller
α : current control policy
X : control parameter
RoundInt
X_d : feature setting for next frame
Application
$Y \leftarrow App(X_d)$
Y : observed frame time
feedback Y
Tune Application Feature X

Figure 4.1: Block diagram of the adaptive feedback controller.

The lower layer tunes the value of the control parameter X for the next frame of the application based on the measured execution time Y of the current frame. *App*() represents the *unknown* response of the application's frame time Y to the application feature setting. The programmer's or user's desire is to keep the frame time Y in the *objective window* $[Y^{obj} - \delta,\ Y^{obj} + \delta]$. The controller increases or decreases the value of X based on the observed error $Y - Y^{obj}$. A discretized value X_d (X rounded to the nearest integer) is applied as the application feature setting for the next frame. If Y fell inside the desired objective window, then the frame is considered a *success* and no correction is made to X. Otherwise, the frame is considered a *failure* and X is adjusted for the next frame. The success of the feedback controller is measured as the **satisfaction ratio** (SR), which is defined as the fraction of the frames whose execution time fell inside the objective window when executing the application on a given test data set.

Adjusting the generic control input X_d would correspond to adjusting the values of application-specific algorithmic parameters, such as the motion-estimation *search-window-size* in the MPEG2 encoder application. The controller itself is unaware of the nature of the algorithmic parameter, except that adjusting X_d is expected to simultaneously increase or decrease both the quality of subsequent computational results and the frame times exhibited. When incorporating our controller in their applications, we require programmers to bind specific integer values of X_d to values taken by the algorithmic parameters. In doing so, *we eliminate the need for the programmer to make uninformed guesses about the response characteristics of the application.* In particular, we do not require the programmer to specify functions that relate values of an algorithmic parameter X to the resulting frame time Y or computational quality. Nor do we require programmers to provide any knowledge about the magnitude of the correction ΔX needed to the current X in order to bring about a desired correction ΔY in the subsequent frame time Y. Indeed, as we expressed in the challenges in Section 2.2, the lack of such modeling knowledge by the programmer is what creates a need for our technique. Instead, we require the programmer to pick a sampling of parameter values over a range sufficiently large to exercise a wide range of frame times and computational quality. For example, in the MPEG2 encoder, it is relatively easy for the programmer to pick the following list of values for the search-window-size parameter without needing much, or any, knowledge of expected input data sets: $30, 20, 15, 10, 5, 2, 1, 0$. The values on the left can be expected to generally produce a larger frame time and computational quality than the values on the right. The programmer binds the values to consecutive integers $X_d = 0, 1, \ldots, 7$. Note that in the absence of our controller, the MPEG2 encoder is always invoked with a *fixed value* for the search-window-size parameter. The programmer utilizes application-specific knowledge to establish that a fixed search-window size of 30 produces extremely large encoding times, with values of $20, .., 5, .., 0$ progressively shortening the per-frame encoding times, producing a large range of frame times. Specific knowledge about the raw input video characteristics or the choice of the objective window is not needed in picking these samples. While it can be expected that a very careful choice of samples may further improve the QoS delivered by our controller, we demonstrate that substantial improvements in QoS can be achieved simply by picking any scattering of samples over a large range as exemplified above, so long as the impact of the chosen samples on the corresponding frame-time QoS metric is *monotonic*.

The extremely limited prescription from the programmer makes the construction of a feedback controller a non-trivial task. However, the interactive nature of the gaming and multimedia domains allows us to make good assumptions about how long (i.e., number of frames) the characteristics of *App*() can be expected to *stay steady*. By staying steady, we refer to Y remaining mostly unchanged over a sequence of frames when X_d is held fixed. Over different "regions", i.e., sequences of consecutive frames, Y may hold steady at different values for the same fixed X_d value. The length of the regions may vary. However, the nature of the domains allows us to make reasonable assumptions about the minimum length of a steady region. Under the assumption of a steady region, we can derive tests which indicate whether the current control scheme (denoted by α in Figure 4.1) is working *as well as possible* in keeping Y within the objective window. Therefore, we can define specific *failure modes*, detect when a failure mode occurs on the current α, and adjust the control policy to α^{new} so that the failure mode ceases to occur on subsequent frames. This is the task performed by the upper layer in Figure 4.1. This layer maintains *failure* *metrics* corresponding to each failure mode. When specific metrics fail certain tests, the corresponding failure mode is said to have occurred at that point in the frame sequence. The failure metrics also carry the *quantitative information* necessary for formulating a corrected control policy α^{new} that will be put into effect from the next frame onwards.

Such a strategy for controller design, where the control policy is constructed and adapted based solely on the observed behavior of the application without relying on application-specific models, is called system identification.

While it would be possible to enhance our technique with application-specific knowledge, such as examining transform coefficients or the nature of the motion vectors in MPEG2, we restrict the scope of this work to examining how far we can push the system-identification approach. Such a restriction allows our work to be applied to not just well-studied applications like MPEG2, but to a broader class of applications including emerging ones that are not yet well understood.

4.1 Contributions

This work makes the following research contributions:

- We illustrate the hitherto untapped potential of applying control theory with system identification to the problem of achieving high QoS in frame-oriented interactive applications. We make the case that such an approach is a simple, practical and broadly applicable approach for QoS optimization in applications that defy traditional formal treatment.

- Using the gaming and video domains as important representatives of frame-oriented interactive applications, we show that *just three domain observations* (scalable algorithms with monotonic parameters, sliding window of user perception W, and the likelihood that the application behavior will remain stable over at least W consecutive frames) prove sufficient for driving the system-identification approach and allow the construction of a broadly applicable adaptive controller that produces large QoS improvements without using application-specific knowledge.

- As a first step, we chose to use the simplest possible feedback law, namely a linear proportional controller (P controller). We demonstrate how the domain observations above can be used to analytically derive failure modes and failure metrics over a P controller, and thereby derive an adaptation policy for the controller. The simple nature of a P controller greatly simplified the determination of failure modes and metrics thereby allowing us to provide robust justification. We experimentally demonstrate that our adaptation strategy applied over even a P controller delivers substantial QoS improvements for real-world applications.

- We motivate that incorporating our adaptive controller into a new application places a very low burden on the programmer. The programmer does not need to provide any information about the dynamic response characteristics of the application and does not need to intelligently pick sample values for the algorithm scaling parameter. The programming effort needed to tie the application frame rate and the algorithm scaling parameters to the controller's inputs/outputs is expected to be quite low.

4.2 Domain Observations

We make the following observations about the common characteristics of frame-oriented video and gaming applications. We use sophisticated real-world applications in realistic scenarios to provide illustrative examples from each domain: MPEG2 encoder for video, and Torque for gaming.

1. **Monotonic effect of algorithmic parameters** The key compute-intensive algorithms within the video and gaming domains tend to be scalable by nature. Typically, these algorithms are heuristic-driven and their runtime complexity can be adjusted over a wide range based on tuning a handful of parameters. When the parameters are set to maximize runtime complexity, these algorithms tend to produce the highest quality results. Conversely, when set for low runtime complexity, they tend to produce a low-quality result, but very quickly. The algorithmic parameters often control the

number of iterations the algorithm spends refining a result. Or, an algorithmic parameter may limit the search space explored by the algorithm. Therefore, we make the observation that the runtime complexity and the corresponding quality-of-result for such algorithms is very often a *monotonic* function of their corresponding algorithmic parameters.

In subsequent discussion, we limit ourselves to applications where the programmer can find algorithmic parameters that monotonically impact the application QoS, in particular the frame time. The monotonicity requirement is easily satisfied by two real-world applications we use to validate our technique, and we seek to illustrate that the requirement would also be easily satisfied by additional interactive applications. It is well established in existing research literature [70] that the search-window-size parameter in MPEG2 has the most significant effect among all available parameters on the frame-encoding times, and this effect is monotonic due to the nature of the search algorithm. Our second application is a commercial game engine called Torque [71]. Based on the game-engine documentation and the user forums, we were quickly able to recreate a common game-play scenario where two teams of simulated enemies are involved in combat. The behavior of each character (bot) in these teams is determined by the *artificial intelligence* (AI) algorithms that run periodically. There are a number of clearly defined parameters that control **i)** how frequently each bot "thinks", and **ii)** the range of visual information (about the adjacent terrain, and the locations of friends and enemies) that the bot incorporates in its thinking for determining its next goals (escape, fight, seek ammunition) and for the path planning to achieve those goals. These *think-interval* and *visual-range* parameters also monotonically scale the AI algorithms.

Once the algorithmic parameters are identified, they are straightforward to tie to X_d and have the feedback controller adjust them dynamically. Given the monotonicity of all the parameters involved, we expect the programmer to combine all the used parameters into a *single formal parameter* X_d, which takes positive integral values starting from 0 up to a maximum value N defined by the programmer. Therefore, the adaptive controller will only adjust X_d, and the new values for the underlying algorithmic parameters will be determined from a *fixed mapping* defined by the programmer from integers $0, \ldots, N$ to tuples of values over the underlying parameters. Ultimately, $Y \leftarrow \mathit{App}(X_d)$ is an unknown, and, quite likely, a time-varying function due to the significant data-sensitive nature of these applications. However, *App*() can still be assumed monotonic in X_d at any given time instant.

2. **Perceived frame rate** A typical video stream can be expected to exhibit similar characteristics over short sequences of frames. For example, the sequence may correspond to the video camera panning horizontally. Or the sequence may have captured objects moving across a fixed background image with relatively uniform velocity. In either case, the computational requirements for encoding each of the frames in such a sequence can be expected to be similar. Given a typical frame-rate of $20 - 40$ fps in video, it is likely that such sequences are quite common and their length is at least 20 frames or substantially more. Similarly, in a fast first-person-shooter game such as Torque, it can be expected that the game-world state does not change too quickly within a sequence of frames. At a typical frame-rate of $30 - 60$ fps, such sequences should be of non-trivial length as well. This observation allows us to expect that for video and gaming applications, *App*() is only a *slowly* time-varying function. Therefore, if a desirable value of X_d (that keeps Y in the objective window) is found early in the sequence, it can be expected to work for the rest of the sequence, thereby delivering a high satisfaction ratio.

   Unfortunately, a video encoder like MPEG2 performs different types of computation on adjacent frames. Typically, treats a video stream as a recurring pattern (called GOP) of I-P-B-B-P-B-B frames (one common pattern). The pattern is imposed oblivious of the data characteristics of the raw video frames. Motion estimation is not applied to the I frames, applied uni-directionally to the P frames, and applied bi-directionally (effectively twice) to the B frames. Similarly, a sophisticated game like Torque has an event-scheduling loop at its heart that can cause adjacent frames to vary considerably in the amount of computation scheduled for them.

However, even though adjacent frames may vary substantially in the computation performed in them, our observation of slow variation in *App*() holds *on average*. For example, Figure 2.1 (from Chapter 2) shows that the frame time can vary quickly between adjacent frames in the MPEG2 encoder (here the algorithmic parameter, the search-window size, is kept fixed). However, the frame time as a moving average over the previous *seven* frames (the length of our GOP pattern) varies slowly except for moments of major discontinuity in the video scene. The Torque game engine exhibits similar behavior.

Most importantly, the *instantaneous* frame rate is not perceptible by the user. For example, the instantaneous frame rate could occasionally drop significantly below the desired rate and the user would not notice the drop provided the frame rate recovered quickly. Therefore, the *perceived frame rate* is determined by the current *moving average* of the frame times of the previous few frames. Fast-action games and high-quality video feeds typically have a frame rate of around 30 frames-per-second. The perceived frame rate can be estimated as a moving average of the previous five-to-ten frames, depending on the specific attributes of the application. We refer to the length of this moving-average window as the **sliding window**, W, for that application. Therefore, rather than trying to keep the instantaneous frame times in the objective window, our controller instead attempts to keep the perceived frame times within the objective window. This allows us to assume that *App*() varies only slowly with time. *Henceforth, Y will refer to the moving average of the previous W frame times.* Further, W also serves to define the *minimum duration of time that is perceptible to the user*, and is therefore useful to the adaptive part of the controller in deciding when a series of failures has gone on *too long* (i.e., the failures may become perceptible to the user) and the control policy ought to be corrected.

3. **Response sensitivity** Depending on the nature of the data being processed, the ΔX change needed in X to produce the same ΔY correction in Y may vary. We refer

to this as the response sensitivity of *App*(). The response sensitivity fundamentally affects the control policy α that ought to be used in the lower layer of the feedback controller in order to achieve a good satisfaction ratio. The following are important causes for the variation in response sensitivity:

- **Global**: The video encoder has $100\times$ more pixels-per-frame to compress in a 1600×1200 video compared to a 160×120-resolution video. Even if the two videos otherwise have similar characteristics, Y can be expected to be scaled correspondingly. Similarly, in a game, the amount of game-world state to be processed per frame grows with the number of objects and bots set up in the game world. Therefore, the response sensitivity to two different data sets may be vastly different depending on the *invariant characteristics* of the data sets. A similar global effect will show up between running the application on fast hardware versus slow hardware.

- **Time-varying**: While the response sensitivity may not change as much over time within the same data set, it varies sufficiently that α must be fine tuned occasionally. The faster an object moves across the screen in a video stream, the further motion estimation has to search to find a matching image block in an adjacent frame. Therefore, if the number of fast moving objects in the scene changes substantially, the response sensitivity may change as well. In a game, if many bots are playing closer together, then the AI computation for each bot will have to take into account the positions of more bots. These effects will dissipate when the number of objects in motion in the video changes or there is a scene-cut, and similarly when the bots move far away from each other. Such regional variations are significant factors in the relative magnitude of ΔY versus the corresponding ΔX. Based on the scenarios described above, we should expect that the regional variations appear over much longer sequences of frames. For example, if the speed of a fast object in a video changes, adjusting X may be sufficient to correct Y without adjusting α. However, if the number of fast moving objects doubles, a change in α may be required. The speed of an object may stay fixed for *tens* of frames, but it may take *hundreds* of frames for persistent moving objects to be added or removed from the scene.

The global scenario suggests the need for determining a large adjustment to α very soon after the application starts execution on a new data set on an unknown hardware platform. Such a large adjustment is only rarely expected to be needed again in further processing of the data set. The time-varying response sensitivity is expected to appear repeatedly over a data set, each time requiring a fine-tuning of α.

4.3 Adaptive Feedback Controller

Our adaptive feedback controller can be added to an application as a library. The programmer needs to specify the following to the controller:

- N integer: X is allowed to vary in the range $[0, N]$.
- W: A suitable length for the sliding window.
- $[Y^{obj} - \delta,\ Y^{obj} + \delta]$: the objective window to keep Y in.

Our convention is that $X_d = 0$ represents the *most* compute-intensive algorithmic setting, which produces the largest Y and causes the application's algorithm to produce the highest quality computational result. Conversely, $X_d = N$ should produce a low-quality computational result in the shortest frame time Y. The programmer should select W to be short enough that the moving average of frame times Y over the previous W frames can show a response as a consequence of any changes in X_d before the user perceives a change in the frame rate. That is, if W is too long, then a drop in the instantaneous frame rate over multiple frames may not affect Y sufficiently for the adaptive controller to notice and attempt corrective action. Instead, the user will perceive the dropped frame rate before the adaptive controller may fix it. Therefore, a large W introduces *feedback lag* in the system where the controller always responds too late. On the other hand, W should be chosen large enough to smoothen frame-to-frame variations that occur spuriously as described under Section 4.2 (the GOP pattern in the MPEG2 encoder; the unbalanced scheduling of events over adjacent frames in Torque). The programmer can try different values of W within a range allowed by the above considerations, and pick the one that produces the best satisfaction ratio on test inputs. Additionally, the programmer needs to tie the values of X_d to the parameter values used by the scalable algorithms in the application.

In this work, we choose to keep the feedback control policy as simple as possible so that it would be easier to adapt based on observed application behavior. Here the control policy α represents a single scalar parameter whose value is adjusted when the control policy is changed. We use a simple proportional controller as follows:

$$\Delta X \leftarrow \frac{1}{\alpha} \times \Delta Y \tag{1}$$

In other words, $\frac{1}{\alpha}$ is the *feedback gain* of the control system. ΔX is the amount by which X should be changed in the next frame given that the observed frame-time Y of the current frame deviated from the center of the desired objective window by $\Delta Y \leftarrow Y - Y^{obj}$. While X and ΔX take continuous values inside the controller, the value of X passed to the application is the closest integral value (X_d).

Since we control ΔY using ΔX, instead of controlling Y using X, and because the gain α is adapted to the application behavior, the controller is an instance of adaptive-integral control.

Note that even though the control policy is that of linear feedback, the application, in general, may be highly *non-linear* and *time-variant*. Therefore, the overall closed-loop feedback system cannot be analyzed as a linear control system.

4.4 Adaptation of Control Policy

In Section 4.2 we made limited assumptions about the manner in which application characteristics hold steady and how they may vary over time. The assumptions now allow us to detect sub-optimal modes of operation of the controller which indicate that some adjustment to the control policy is likely to deliver significant improvements in the `SR`. We contrast this to the situation where the `SR` is poor but adjusting the control policy is not likely to improve it much. We would refer to the former sub-optimal modes as failure modes since the controller could do better, but the latter situation is not a failure mode since it is not indicated that another control policy would do better.

Failure Modes. The only design parameter in the feedback controller is α. Figure 4.2 shows the `SR` achieved using the feedback controller, but with the adaptive layer disabled. Here α is a priori fixed to a given value and not altered during the execution of the application. The figure shows that for any given data-set, the value of α must be chosen from a narrow range that is specific to that data-set. If α is not in this narrow range, the `SR` of the feedback-controlled system drops significantly. Therefore, offline profiling on representative data-sets is unlikely to train α well for an as yet unknown data-set. *Online adaptation of α is the only option.*

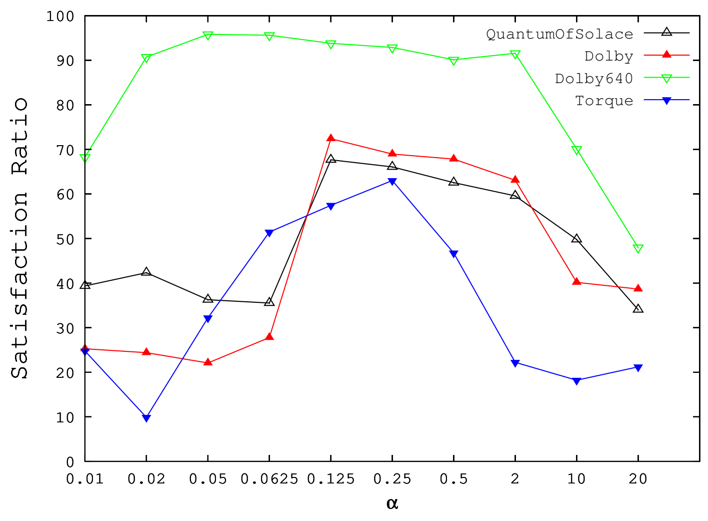


Figure 4.2: Dependence of α on the application and the data set.

Since the controller is a simple P (proportional) controller with a single parameter α, the following three cases provide a complete spectrum of ways in which the policy can be adapted in any situation:

- **Global failure**: gets α into the correct order-of-magnitude range.

- **Oscillation failure**: fine-tunes α by increasing it.
- **Sluggishness failure**: fine-tunes α by decreasing it.

When the application starts execution, the controller initializes with an arbitrary and very small value for α. The Global failure mode determines a large multiplicative correction for α within the first few frames of the application's execution. Subsequently, the Oscillation and Sluggishness failure modes determine much more fine-tuned corrections to α. They are also invoked for further fine-tuning when the application encounters regions of the data with differing response sensitivities. The following subsections elaborate on these failure modes. Each failure mode continually maintains metrics (failure metrics). When a failure metric crosses a threshold, it indicates that the corresponding failure mode has occurred. Additional associated metrics indicate how to correct α to dispel further occurrence of that failure mode.

4.4.1 Global failure mode

The occurrence of this failure mode causes α to reach its correct order-of-magnitude range in a single adjustment of the control policy (the range with high `SR` in Figure 4.2). Section 4.2, in the global response sensitivity paragraph, describes scenarios where such an adaptation is necessary (adjust for video resolution, speed of hardware, etc.). This failure mode occurs when the ΔX corrections due to Equation 1 are so large that they quickly cause X to move out of the $[0, N]$ range on either side. In such a situation, the feedback controller no longer has effective control over Y. This failure mode will occur when α is so small that the typical magnitudes of ΔX are continuously greater than 1.0 when Equation 1 is applied to the currently observed values of ΔY, thereby not letting the controller exercise any single value of X for even a short duration of frames.

The following metrics are maintained for this failure mode, and updated after every frame:

- β: The running average of $|\Delta X|$'s observed so far.
- $\gamma \leftarrow c * \gamma + |\Delta X|$

γ represents a weighted sum of absolute deviations $|\Delta X|$. $0 < c < 1$ is a convergence factor, where the contribution of older frames is de-emphasized with weight c. The main idea here is that if the absolute deviations $|\Delta X|$ are observed to be consistently much larger than 1 (the minimum separation between values of X_d) then it is indicated that the controller is continuously and quickly exploring the full-range of X (i.e., $[0, N]$) or it is consistently exceeding this range altogether. If Y fell inside the objective window, then ΔX is 0.0 for that frame. Therefore, a series of non-zero $|\Delta X|$'s can occur only when the current control policy has a series of failures in keeping Y within the objective window.

The failure mode is considered to occur when $\gamma > \frac{1}{1-c} \times 1.0$ and $\beta > 1.0$. Let's choose $c = 0.9$. The $\frac{1}{1-c}$ term is simply the convergence value of the geometric series sum $\sum_i c^i$. This implies that a single large spike in ΔX would have to exceed $\frac{1}{1-c} \times 1.0$ or approx 10.0 for this failure mode to be triggered. On the other hand, a sustained ΔX of magnitude slightly greater than 1.0 would trigger the failure mode after about $10's$ of frames. Therefore, the above condition allows a global-correction of α to be either triggered quickly by a very large ΔX or by a ΔX of moderate magnitude sustained over multiple frames. This filters out intermittent large spikes in ΔX from triggering the failure mode unless they are extremely large. More generally, $c = 1 - \frac{1}{W}$ would be a better justified choice, as this would suggest that sustained ΔX's of magnitude > 1 would have to occur for about W consecutive frames for a policy failure to have occurred. Slight anomalies in frame-times that are sustained for less than W consecutive frames would not be perceptible to the user, and hence should not be detected as policy failures.

When this failure mode occurs, α is corrected as follows: $\alpha^{new} \leftarrow \beta \times \alpha$. Therefore, when values of ΔY of similar magnitude as before the correction are observed, the resulting values of ΔX produced by Equation 1 would be of a smaller magnitude ≈ 1.0 (instead of average magnitude β). Therefore, the controller would now be able to exercise valid settings of X instead of exceeding range on either side. Further fine-tuning of α may occur subsequently via the other two failure modes. After the correction is applied, the failure metrics are reset to 0.0 and begin updating as usual.

Since we do not provide a failure mode that could make a correspondingly large correction to α when α is too large, we ensure that at application start-up time the controller is initialized with an exceedingly small value for α. Therefore, an additional failure mode that decreases the value of α by orders-of-magnitude is not needed.

4.4.2 Oscillation failure mode

This failure mode occurs if the value of α is sufficiently low, leading to a high feedback gain that causes "under-damped" oscillations of large magnitude or frequency. The crests and troughs of these oscillations need to fall outside the objective window on either side in order for this failure mode to occur. This corresponds to the situation where the ΔX corrections (from Equation 1) being applied to X are so large that they cause Y to swing from one side of the objective window to the other, but not stay confined within the objective window. Figure 4.3 illustrates this phenomenon. On the left is a single half-cycle oscillation with high amplitude. On the right is a sequence of half-cycle oscillations. Note that there is no half-cycle after the H_3 label as Y reaches its crest within the objective window.

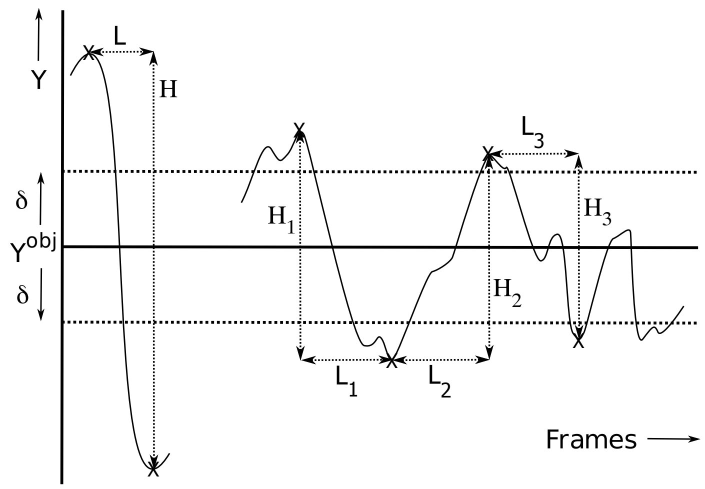


Figure 4.3: Metrics for Oscillation failure mode

In order to detect this failure mode, we have to assume that the application is currently in a steady-state where an appropriate fixed value for X_d can keep Y within the objective window. Then, the only cause for the oscillations would be that the current value of α is causing X to vary faster than the application can respond to a change in X. Therefore, we get oscillations as X is continually over-corrected in each direction. Therefore, the failure mode should adjust α so that the magnitude of ΔX is appropriately reduced, and the feedback controlled application has a chance to settle into steady state.

If the above assumption of an achievable steady state is valid, we can also assume that the steady-state lasts for at least W frames. This is justified because W is a bound for the minimum perceptible length of frames, and in Section 4.2 we motivated that a sequence of frames of fixed characteristics (hence steady state) are likely to be of a duration that is perceptible to the user. Hence we use W to determine if the frequency of oscillations is sufficiently high to justify triggering the failure mode.

If the assumption of steady-state is not valid over the current region of frames and the Oscillation failure mode is incorrectly triggered, we can rely on the Sluggishness failure mode being subsequently triggered to negate the correction made by this failure mode. Therefore, we just need to create a high barrier against this failure mode being triggered inappropriately, not eliminate the chance completely.

The following metrics are maintained for this failure mode:

- L: A half-cycle represents a transition of Y from one side of the objective window to the other. The L metric counts the number of frames involved in the current half-cycle.
- H: This captures the crest to trough or vice versa height of the current half-cycle.
- η: This metric is a weighted sum that provides a combined measure of the frequency and magnitude of previously observed half-cycles. At the end of each half-cycle, this is updated as follows: $\eta \leftarrow d * \eta + H \times \dfrac{W}{L}$, where $0 < d < 1$ is a convergence factor. Again the older frames are de-emphasized with weight d.

The Oscillation failure mode is triggered when $\eta > \tau$ where threshold $\tau = \frac{1}{1-d} \times 2\delta \times 1.0$. Note that 2δ is the height of the objective window. The basis for this condition is that in the absence of oscillations there should be on average at most one half-cycle per W frames. Once the failure mode is triggered, α is corrected as follows: $\alpha^{new} \leftarrow \alpha \times \dfrac{\eta}{\tau}$. We chose

$d = 0.66$ so that a single half-cycle of magnitude or frequency at least three times greater than one half-cycle per W frames can trigger the failure mode, or that a sustained set of reasonably large half-cycles can trigger a correction within the occurrence of $3 - 4$ such half-cycles. This allows α to be fine-tuned rapidly at the onset of a new region with a different response sensitivity (Section 4.2).

4.4.3 Sluggishness failure mode

This failure mode occurs when α is sufficiently high that Equation 1 produces ΔX of low magnitude, leading to a sluggish "over-damped" response. The failure manifests itself as a continuous series of ΔY values of the same sign, indicating that Y is continuously falling outside the objective window on the same side. Additionally, X is not changing quickly enough to produce a faster correction in Y.

As with detecting the Oscillation failure, we make an assumption of steady state, with W serving as the minimal length of frames after which the application response can be considered as sluggish. Any inappropriate triggering of the Sluggishness failure mode is likely to be counter-acted by a subsequent Oscillation failure mode.

The following metrics are maintained for this failure mode:

- K: This keeps track of the current number of contiguous frames whose Y's have all occurred outside the objective window on the same side of the objective mean (i.e. all Y's are either too small or all too large).
- λ: This metric accumulates ΔX over the last K frames.

This failure mode is triggered when $K > W$ and $\lambda < 1.0 \times \mu$ where $\mu = \frac{K}{W}$. The general idea is that the response is sluggish if the cumulative change in X (λ) was less than 1.0 per W frames of one-sided failure. Due to X_d being integer-valued, 1.0 is the minimal change in X on average that can produce a change in behavior in Y. The failure mode corrects α as follows: $\alpha^{new} \leftarrow \alpha \times \frac{\lambda}{\mu}$.

In summary, the metrics for each failure mode are updated and tested after each frame for the occurrence of the corresponding failure mode. Note that the nature of the metrics and tests for the Oscillation and Sluggishness failure modes make it impossible for both these failure modes to occur simultaneously. This is because Y cannot be simultaneously oscillating rapidly on either side of the objective window as well as persisting on only one side of the objective window. However, the Global and Sluggishness tests could both detect failure at the same frame. If so, the Global failure is given precedence as its detection corresponds to the occurrence of large magnitude anomalies in Y, which can be largely corrected in one step. In contrast, the Sluggishness failure metrics do not account for the magnitude of errors in Y, and can only bring about minor corrections in α each time the Sluggishness failure mode is detected.

4.4.4 Illustration of Failure Modes

Figure 4.4 shows a frame sequence for the MPEG2 encoder application modified to use our adaptive controller, and a corresponding frame sequence for the unmodified application that operated with a fixed setting for X. We choose $X = 2$ for the latter since this produced the best `SR` among all possible fixed settings for X. The two horizontal lines in the top part of the figure demarcate the boundaries of the objective window. It is clear visually that the feedback controlled system stays within the objective window for significantly more frames than the fixed choice case. This observation is also borne out by comparing the ongoing satisfaction ratios plotted in the lower part of the figure.

The upper part of the figure shows variations in Y. The lower part of the figure shows the corresponding X that was applied. For the unmodified application case, X is always 2. At frame 9, α is corrected from the default initial value of 0.0002 to the new value of 0.09764 (an orders of magnitude correction) in a single step. The cumulative `SR` shown as a black dotted line in the lower part of the graph immediately gets a boost after this correction is applied. In contrast, the gray dotted line showing the cumulative `SR` for the fixed $X = 2$ run, stays quite low indicating a high rate of failure at that point. The Global failure mode is denoted with `G` and the new corrected value of α is shown at the point of correction. At frames $45, 141, 155, 181$ under-damped oscillations of large magnitude and/or high frequency appear, and are corrected within one or two half-cycles of the start of the

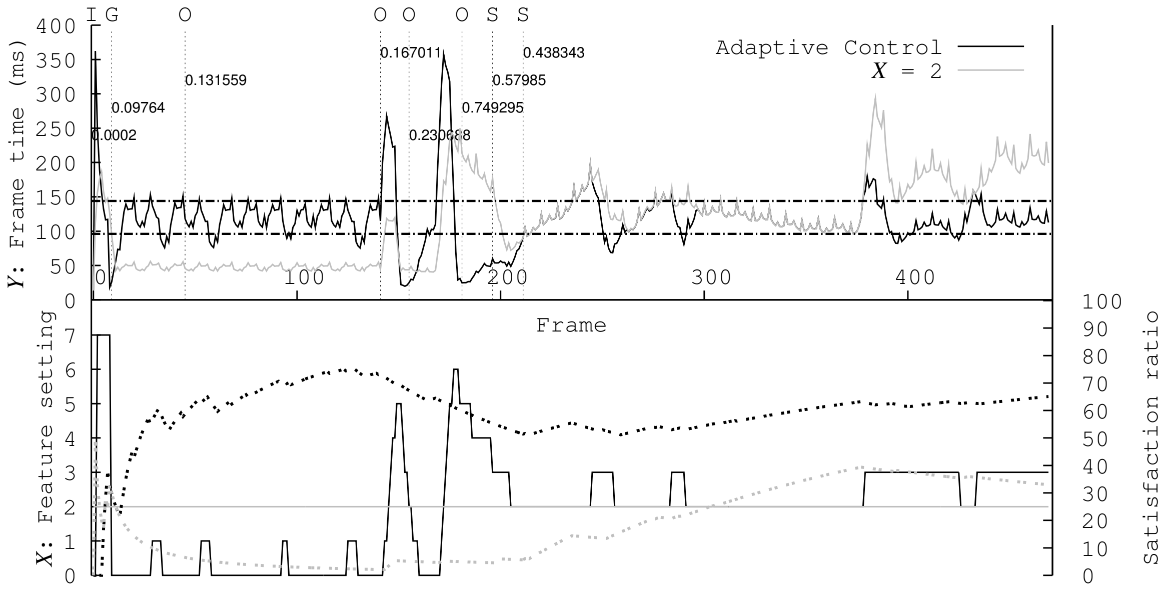


Figure 4.4: Frame Sequence for the MPEG2 encoder on the `Quantum of Solace` video: Adaptive versus fixed $X = 2$. $X = 0$ uses largest Motion Estimation search window, $X = 7$ uses smallest search window. G, O and S mark frames where global, oscillation and sluggishness corrections are made to α, with the corresponding multiplicative factors annotated.

oscillations. The occurrence of an Oscillation failure mode is denoted with `O` and the new corrected value of α is shown at the point of correction. The Sluggishness failure mode is seen to occur at frames 196 and 211 where the frame execution time stays outside the objective window for extended periods of time while the feedback system does not adjust X fast enough to bring Y back into the objective window. These occurrences are denoted with `S` and the new corrected value of α is shown at the point of correction.

4.5 Experimental Validation

We use the MPEG2 encoder from the Mediabench II video [70] benchmark suite, and the Torque game engine to validate our methodology. We use three different raw video sequences to test the MPEG2 encoder. The video sequences are called `QOS`, `dolby`, and `dolby640`. The first is a prefix of a trailer from the *Quantum of Solace* action movie, consisting of 470 frames at a 320×192 resolution. `dolby` and `dolby640` are derived from the commonly used video test-sequence *dolbycity*. They are both 799 frames long with resolutions of 320×192 and 640×480, respectively. To drive the Torque game engine, we recorded a sequence of movements that the player executes at specific frame counts within the game world. This produces a sequence of 900 game frames in each run. We do not directly control the behavior of the bots in the fighting teams. Their behavior is controlled by the randomized AI algorithms, for which we control basic parameters affecting AI intensity and vision range. We use the satisfaction ratio (SR) metric to measure the QoS performance of an application run on a given data set. For reasons explained in Section 4.2, we measure the SR over the sliding-window-averaged frame times rather than the instantaneous frame times. In the MPEG2 encoder, X was given a range of *eight* values $(0 - 7)$ corresponding to the following values of the search-window-size parameter used for motion estimation: $30, 20, 15, 10, 5, 2, 1, 0$. We ran all experiments on a Core2 Quad Q6600 2.4GHz CPU machine with 2GB of RAM.

Figure 4.2 shows that the range of values of α that produce high SR are highly application and data-set dependent. This unpredictability in the high-SR range for α emphasizes the importance of training α online during each run of the application. Figure 2.1 (from Chapter 2) illustrates the importance of having a sliding window to satisfy the regions of stable behavior requirement. The $W = 1$ case shows the instantaneous frame times for the MPEG2 encoder on a prefix of the QOS video sequence. There is a clear banding of the frame times in each GOP into I, P, and B frames. Computing moving averages of length *seven* is shown by the $W = 7$ case. The latter case clearly illustrates the presence of sequences of frames with steady characteristics in the video sequence.

Figure 4.5 demonstrates the importance of choosing the correct W for each application to achieve the best SR across all data sets. We see that $W = 7$, derived analytically to match the length of the MPEG2 Group-Of-Pictures (GOP), gives the best overall SR across the different video data sets. For Torque we empirically discover that $W = 7$ works the best.

Figure 4.6 shows the impact on SR of picking different frame-time objectives for each application. Evaluating a range of frame objectives also *simulates* the effect of evaluating

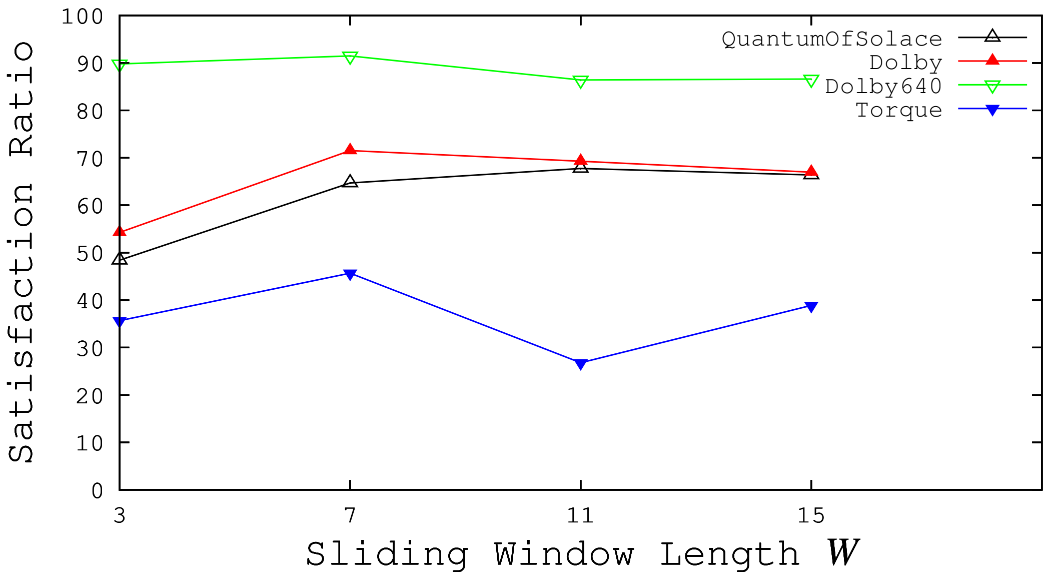


Figure 4.5: Variations in satisfaction ratio (`SR`) against sliding-window size (W).

the application with a *fixed frame objective* over a *range of hardware* with varying compute capabilities. The size of the objective window is kept at 20% on either side of the frame-time objective (i.e., $\delta/Y^{obj} = 0.20$). Experiments are run for various fixed settings of X (i.e., the unmodified application with different configuration settings), and a run of the application modified to use the adaptive controller. For the `QOS` and `dolby` data sets no fixed choice produces a good `SR`. But the adaptive-controller case delivers substantially better `SR` for these data sets. This illustrates that the controller is able to correct X to the value best suited for each sequence of frames. For `dolby640`, $X = 7$ and $X = 5$ deliver extremely high `SR` for $Y^{obj} = 0.08$ and 0.12 seconds respectively. The two fixed choices just happened to work well on our machine for the given data set for these specific frame-time objectives. $Y^{obj} = 0.02$ and 0.04 seconds turn out to be impossible objectives as even a motion-estimation search-window of size 0 cannot achieve them. Note that `dolby640` is a much higher resolution video than the other video samples, leading to correspondingly larger per-frame encoding times.

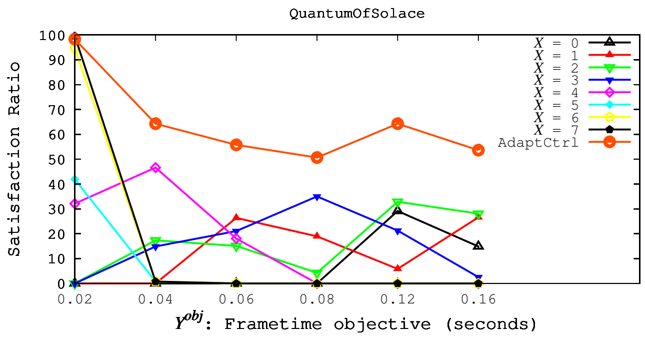
The adaptive case can significantly outperform every fixed X, rather than simply following the envelope of the fixed-X cases (e.g., Figure 4.6(b) at $Y^{obj} = 0.12sec$ has adaptive

SR $\approx 70\%$ while all the fixed-X cases have SR $\leq 40\%$). This is because the response characteristics of our highly time-variant applications can change after sub-sequences of just $10's - 100's$ of frames, making any given fixed X suboptimal over the full frame sequence. Therefore, dynamic tuning has room to significantly outperform every fixed X by choosing the best X appropriate for each sub-sequence.

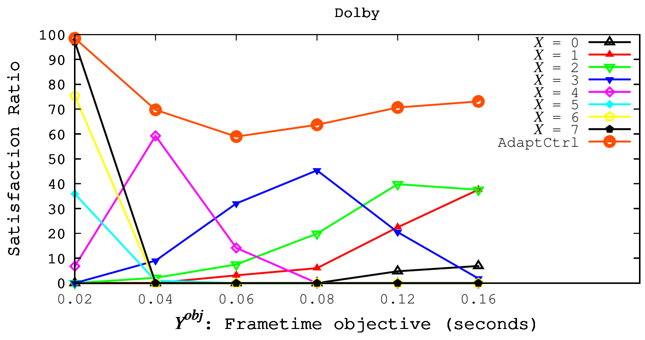
Figure 4.7 shows the spread of the observed frame times Y about the desired mean objective Y^{obj} for Torque. The distortion is calculated as follows:

$$Distortion = \sqrt{\frac{\sum_{i=1}^{M}(Y_i - Y^{obj})^2}{M}}.$$

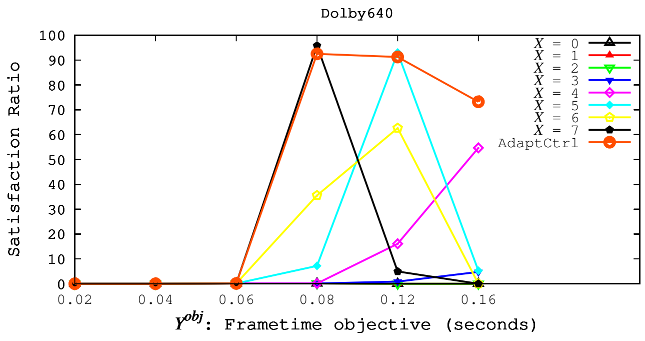
Here, Y_i's are the observed frame times. The distortion is a metric for the overall variation in the observed frame times. Figure 4.7 shows that the adaptive controller produces the lowest, or very close to the lowest, distortion compared to any fixed-X case. For $Y^{obj} = 0.06secs$, the fixed $X = 6$ case has slightly lower distortion in Figure 4.7 but substantially worse SR in Figure 4.6(d) compared to the adaptive case. A similar observation holds for the distortion and the corresponding SRs at $Y^{obj} = 0.12secs$. Except for these two borderline anomalies, the distortion of the adaptive case is always lower than any fixed case. The distortion metric shows that our adaptive controller not only improves the probability of keeping the frame times within the objective window (measured as SR), but also reduces the overall variation in the frame time.



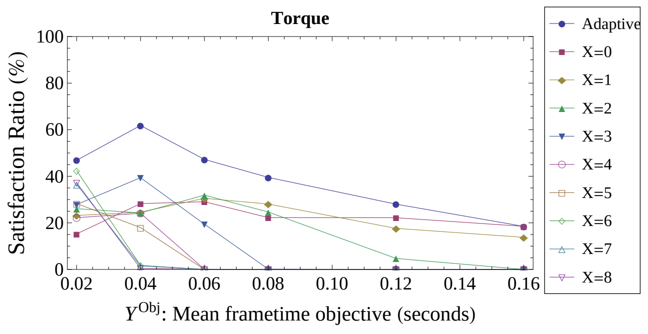
(a) QOS



(b) dolby



(c) dolby640



(d) Torque

Figure 4.6: Variations in SR against Y^{obj} (mean frame-time objective), for fixed-X and adaptive cases.

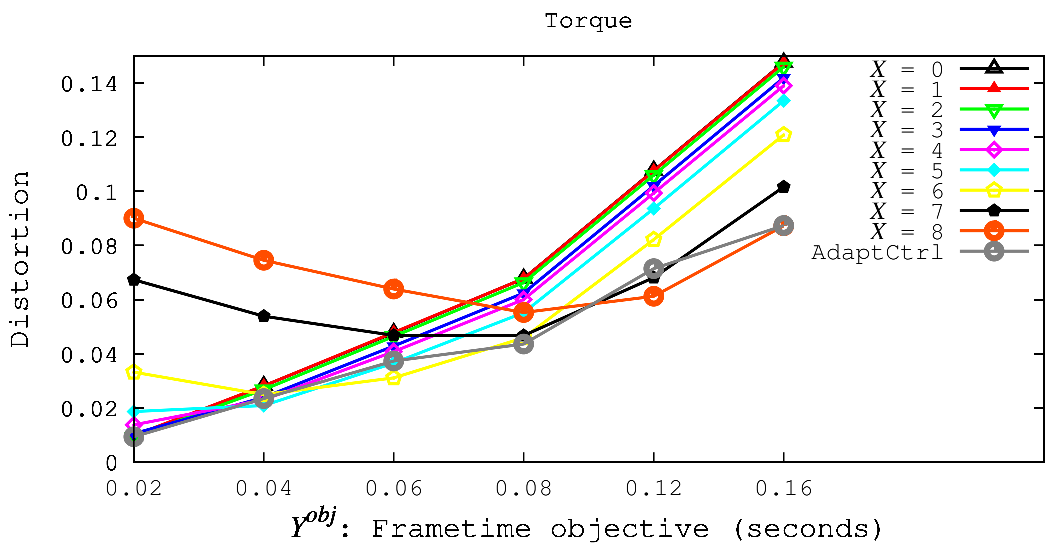


Figure 4.7: Distortion in the frame times for Torque.

Our adaptive controller does not directly address other aspects of the application's QoS, such as the quality of the computed results (let's call it the *computational-QoS*). In the MPEG2 encoder, the computational-QoS corresponds to the peak signal-to-noise ratio (PSNR) for the compressed frames. In Torque, the computational-QoS corresponds to the intelligence exhibited by the bots, which can be estimated as the average amount of time spent on AI computations per frame. Instead, the controller only directly attempts to maximize the *frame-QoS*, i.e., the satisfaction ratio. For these applications, maintaining a high and smooth frame rate trumps other considerations of computational-QoS, as a poor frame rate directly makes a game unpleasant and jerky to play or a video stream uncomfortable to watch. However, our adaptive controller not only achieves the best frame-QoS, but also indirectly achieves a similar or better computational-QoS for the application compared to the best-case execution of the application that did not use the adaptive controller. For the MPEG2 encoder, we measure the PSNR averaged over all the frames. The encoder is set to produce a constant-bit-rate file, so all the output-file sizes are almost equal. The averaged PSNR does not vary by much across the fixed-X runs and the adaptive-control runs. There is no case where the fixed-X run produces both a better SR and a better averaged PSNR compared to the adaptive case. The Torque game application shows a much larger variation between runs in the average AI-time per frame. The adaptive case always has similar or much better AI per frame compared to the fixed-X case with the best `SR`. In particular, when $Y^{obj} = 0.04secs$ and $\delta/Y^{obj} = 0.20$, the adaptive case spends $24.13ms$ on AI per frame, whereas the fixed case with the best `SR` (for $X = 3$) spends $14.21ms$ on AI per frame. Therefore, our adaptive controller significantly enhances the frame-QoS across data sets and frame-time objectives, without compromising (and sometimes enhancing) the computational-QoS.

The total runtime overhead of our controller was less than 0.05% of the total frame time, in all cases.

4.6 Conclusion

In this work we introduced a system-identification approach to design an adaptive feedback control system for frame-oriented gaming and video applications that are implemented without the use of real-time constructs, and whose highly data-dependent, time-varying nature makes it difficult to establish analytical models relating algorithmic complexity with frame-execution times. We demonstrate that the proposed adaptive controller trains to the characteristics of the application and the current data set based purely on the observed behavior of the application, without requiring any specific knowledge about the application or the data set. We have demonstrated that our controller substantially smoothens the frame rate, and keeps the frame times within a user-specified objective window with probability matching or often significantly exceeding any fixed setting of the application's feature set. Further, our controller achieves this without compromising other aspects of the application's QoS, such as PSNR or game-play intelligence.

CHAPTER V

MULTI-VARIATE QOS CONTROL: APPROACH AND PROBLEM DEFINITION

5.1 Motivation

In Chapter 4 we described an adaptive controller that significantly improved the frame-rate QoS (the expressed "feature-set") of frame-oriented interactive applications. While the adaptive controller was applicable to a large variety of frame-oriented interactive applications that satisfied three broad *domain assumptions*, the controller dynamically adjusted only a *single* application parameter X in order to keep only a *single* objective, typically the observed frame-time Y, within a desired range.

Here we introduce a generalization to *multiple*-X, *multiple*-Y, applicable to a similarly large variety of interactive applications. The multiple-Y's allow the simultaneous optimization and trade off between a larger set of QoS metrics beyond just frame time. Adjusting multiple-X's allows for a much more fine-tuned optimization of the multiple-Y's, both individually and collectively.

The X's represent *application-specific algorithmic parameters*, whose adjustment causes the application's *emergent* QoS characteristics to vary. While it is relatively easy for programmers to describe the desirable ranges for QoS metrics (Y's) and identify which application parameters (X's), if varied, are likely to significantly affect the QoS metrics, it is very difficult for programmers to establish any relationship about *how* the X's ought to be varied to bring about a desired change in the Y's. The reasons for this difficulty are the same as those motivated for the single-X, single-Y problem. Namely, the relationship between the multiple-X's and multiple-Y's in an interactive application can be *i)* highly data-dependent, *ii)* time-varying and *iii)* non-linear in general.

Hence, an *automated, non-application-specific controller* that can dynamically discover the X-Y relationships, and adjust the X's to keep the Y's within desired ranges, *will* *tremendously simplify the work of building interactive applications and achieving high QoS.*

5.2 Contributions

1. We propose a *statistical problem specification* that is suitable for capturing the QoS requirements of immersive applications, and is amenable to being effectively solved via an online controller operating under a bounded compute budget.

2. We use powerful multi-variate *offline* system-identification and optimal control techniques, LLSE and LQR, respectively, in a novel manner that makes them amenable for an *online* controller. In particular, we create techniques to overcome some well-known limitations of LQR, which allow our controller to be far more robust and deliver higher QoS performance even with highly approximate system-identification.

3. We create a number of *probabilistic and estimation techniques* necessary to tackle the following challenges specific to the online system-identification and control of immersive software applications:

   - almost no designer "intuition" is available to guide the selection of various parameters used by LLSE and LQR,
   - the training data for model estimation is typically extremely limited and very noisy,
   - and the application behavior typically changes too rapidly and too significantly to assume a fixed underlying model (even implicitly) in the solution technique.

5.3 Overview

This chapter provides the problem specification and an outline of the controller design. We discuss how LLSE and LQR apply to the online controller. Then we identify a number of challenges with LLSE and LQR that are traditionally solved with designer insight on a fixed use-case in a offline-design setting, but must now be addressed solely from the problem specification without specific knowledge about any use-cases.

Chapter 6, first provides the context for our work among system-identification-based control techniques and elaborates on the necessary design trade-offs involved given the nature of the problem. Then, the chapter details the solutions to the identified challenges. Next, Chapter 7 provides experimental results and case-studies with immersive applications using our controller. Finally, Appendix B lists all the controller algorithms at a high-level, allowing the easy reproduction of the work.

5.4 Problem Definition

The programmer uses an API to identify n application-specific parameters and m QoS metrics. Callback functions are registered for each parameter and metric. The controller is notified of an application frame transition via an API call, at which point the controller uses the callback functions to read the QoS metrics for the current frame t and then apply new parameter values for the next frame $t + 1$. Additionally, the programmer needs to certify that the application's behavior over the identified parameters and QoS metrics exhibits four key properties common to immersive applications, referred to as the *domain assumptions*. The problem instance registered by the programmer, the domain assumptions and the controller's optimization goals are summarized below.

Registered problem instance.

1. QoS output variables $\vec{y}$, the corresponding target values $\vec{\bar{y}}$, tolerances $\vec{\delta}$, and (optional) relative importances $\vec{s}$.
2. Control input variables $\vec{x}$, taking integral values over $-\vec{N}$ to $\vec{N}$.
3. User-perception window W.
4. Per-frame controller overhead allowed b.
5. (Optional) input and output model-orders: x_{order}, y_{order}.

Domain Assumptions.

1. Monotonic response between $\vec{x}$ and $\vec{y}$.
2. Tolerance to QoS deviations within sliding windows of W frames.

3. Stable application-response behavior over durations much greater than W frames with high probability.

4. Range of $\vec{y}$ values exhibited does not change dramatically or frequently over a given application run.

Optimization Goals. The following τ_t metric defines the QoS performance of the controller on frame t of the application. The application executes as a series of frames $t = 1, 2, \ldots$.

$$\tau_t = \frac{1}{m} \sum_{i=1}^{m} s_i \frac{|y_{i|t} - \tilde{y}_i|^2}{\delta_i^2} \tag{2}$$

The controller has two optimization goals.

1. Keep $\tau_t \leq 1.0$ for as many frames as possible. The *satisfaction ratio* metric `SR` is defined as the fraction of application frames with $\tau_t \leq 1.0$.

2. Minimize τ_t whenever $\tau_t > 1.0$. The *mean squared error QoS* metric `MSEQ` is the average τ_t observed over the application frames.

We use the `SR` and `MSEQ` metrics to characterize the performance of the controller in achieving the two optimization goals.

Performance and Robustness. The lack of an analytical model and the changing application behavior makes it infeasible to establish traditional control-theoretic performance properties, such as Lyapunov stability [20, 21, 22, 23], or overshoot and settling time minimization [24]. Instead, the inherently probabilistic nature of the domain assumptions, and the probabilistic nature of the estimation techniques employed, confer the following performance and robustness properties to the controller.

1. High frequency of goal satisfaction among the application frames if the specified QoS goal is attainable.

2. High frequency of application frames that minimize the QoS error if the specified QoS goal is unattainable.

3. Robustness in the form of graceful degradation of QoS performance when the application behavior does not fully satisfy the domain assumptions.

4. Robust estimators that tolerate noise over long sequence of frames, while staying responsive to rapid changes in application behavior due to the reliance on the domain assumptions.

5.5 Problem Definition: Discussion

Let $\vec{y}$ be a vector representing the m scalar QoS metrics $y_1, y_2, \cdots, y_m$ that need to be optimized for the given application:

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}. \tag{3}$$

Let $\vec{x}$ be a vector representing the n scalar parameters $x_1, x_2, \cdots, x_n$ that are to be adjusted by the controller in order to optimize the QoS metrics $\vec{y}$:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{4}$$

with $-N_j \leq x_j \leq N_j$, where N_j are integers specified by the programmer ("input bounds" constraint). The programmers must map the integral values for x_j produced by the controller to values for the actual parameters taken by algorithms within their application. Similarly, the programmers must map the actual output metrics in their application in some way to the real-valued y_i's. The mappings must be done in a manner that produces a monotonic relationship between the x_j's and y_i's (monotonic response domain assumption).

The execution of the application consists of a series of frames indexed by time $t = 1, 2, \ldots$, where the control input $\vec{x}_t$ is applied prior to the execution of frame t, and QoS metrics $\vec{y}_t$ are observed immediately after the execution of frame t.

Nature of Monotonic Relationship. In general, the relationship between each x_j and y_i could be monotonic increasing or monotonic decreasing. This relationship is not assumed to be known. As a special case, some x_j's may have no impact on certain y_i's.

Extension of Problem Definition to support Maximization Objectives. The problem definition covered so far allows a *range objective* to be specified for each y_i — so far the goal has been to maintain $|y_i - \tilde{y}_i| \leq \delta_i$ for programmer-specified $\tilde{y}_i$ and δ_i. However, the controller tolerates the range objective to be changed every few frames (with some runtime overhead to adjust internal metrics and re-design the regulator). We exploit the controller's ability to tolerate changing range objectives to build support for a maximization objective — maximize y_i. With this extension, each y_i can be specified as either a range objective $|y_i - \tilde{y}_i| \leq \delta_i$, or, a maximization objective $[(\tilde{y}_{i,1}, b_{i,1}), (\tilde{y}_{i,2}, b_{i,2}), \cdots, (\tilde{y}_{i,k}, b_{i,k})]$ with $\sum_{p=1..k} b_{i,p} = 1.0$ and $\tilde{y}_{i,1} < \tilde{y}_{i,2} < \cdots < \tilde{y}_{i,k}$. With a range objective the programmer specifies the optimal "center-value" $\tilde{y}_i$ to achieve for QoS metric y_i with a tolerance given by δ_i. With a maximization objective the programmer specifies that if the QoS metric y_i has exceeded a value $\tilde{y}_{i,p-1}$, then there is an additional benefit of $b_{i,p}$ to attempt to have y_i match or exceed $\tilde{y}_{i,p}$. The maximization objective captures the desire to maximize y_i over the range $[\tilde{y}_{i,1}, \tilde{y}_{i,k}]$ as a series of *progressive range objectives* $|y_i - \tilde{y}_{i,p}| < \tilde{y}_{i,p} - \tilde{y}_{i,p-1}$. The controller converts a maximization objective into a series of range objectives that differ incrementally over time.

Specifying the Relative Importances of the QoS metrics. The programmer can specify *importance* s_i (> 0) for achieving QoS metric y_i. $s_i = 1.0$ by default unless specified to be a different value.

The following definition of the *instantaneous performance of the controller* at frame t, τ_t, generalizes Eq 2 to accommodate the maximization-objective extension. For simplicity, subsequent discussion will only use range objectives and Eq 2.

$$\tau_t = \frac{1}{m}\sum_{i=1}^{m} s_i \frac{|y_{i|t} - \tilde{y}_i|^2}{\delta_i^2} r_i \qquad . \tag{5}$$

Here $y_{i|t}$ represents the value of y_i observed at frame t. For a range objective, we use $r_i = 1$. For a maximization objective, when $\tilde{y}_{i,p-1} < y_{i|t} \leq \tilde{y}_{i,p}$, we have $\tilde{y}_i = \tilde{y}_{i,p}$, $\delta_i = \tilde{y}_{i,p} - \tilde{y}_{i,p-1}$ and $r_i = 1 - \sum_{q=1}^{p-1} b_{i,q}$.

Overall QoS Performance. We define two performance metrics to characterize the overall performance of the controller over all the application frames. The first performance metric, satisfaction ratio (SR), is defined as the *fraction* of the frames $t = 1, 2, \ldots$ that exhibit $\tau_t \leq 1.0$. That is, for these frames the QoS metrics were overall satisfied in accordance with the importance placed on them by the programmer. The second metric, mean-squared-error-QoS (MSEQ), captures the *average deviation over all the frames* of the QoS metrics. Hence, if the application executed over a sequence of T frames, then

$$\texttt{MSEQ} \triangleq \frac{1}{T}\sum_{t=1}^{T} \tau_t \,. \tag{6}$$

We will compare the SR and MSEQ metrics between *i)* a run of the application that incorporates the controller, against *ii)* a run of the application that uses the best *fixed* hand-tuned settings for $\vec{x}$ without using the controller, for a given input data set. In this manner, we would seek to establish that case-*i)* consistently delivers better SR and MSEQ over case-*ii)* over a range of input data sets. Such a comparison will establish whether the controller achieves significant QoS improvement for a given application or not.

5.6 Nature of Immersive Applications

Achieving good performance on an unknown application without relying on application-specific behavior models requires us to make some broad assumptions about the domain from which the applications are drawn. In this sense, our controller is *domain specific* rather than application specific. We make four broad observations about frame-oriented applications coming from the gaming, multimedia, interactive visualization and computer vision domains, which we broadly refer to as the *domain of immersive applications.* Thus, immersive applications constitute the domain for which we have designed our controller.

In this section we motivate that any immersive application is likely to conform to the following four observations, suggesting that our controller applies to a wide variety of immersive applications. In the remainder of this chapter and in the next chapter *we show that it is possible to construct a QoS controller solely on the assumption that these observations apply to a given application, with no additional application-specific knowledge provided.*

#1 Parameters that produce a monotonic response in QoS are fairly common. Computer-vision and video-encoding applications typically consist of object- or motion-tracking algorithms that constitute a significant portion of the per-frame execution time. Such tracking algorithms are often heuristic in nature, consisting of a variety of parameters that can be tweaked to achieve desirable application-specific tracking capabilities within a limited window of time. Even though tracking is a common theme across these applications, the unique nature of the object being tracked/recognized (face, person, arbitrary template, etc.) varies considerably with each application, along with differing standards for *tracking accuracy* (specific object for computer vision, versus, finding any closely matching macroblock in video encoding) and differing standards for acceptable computational complexity depending on the compute platform (ranging from small low-power embedded platforms to high-end computers augmented with compute-GPUs). Consequently, the algorithms involved are typically *highly scalable* in their QoS characteristics and their corresponding execution-time complexity. Therefore, it can be expected that such applications have many algorithmic parameters (X's) whose adjustment has a monotonic effect on the QoS characteristics exhibited by the application (Y's), including per-frame execution time and detection-accuracy metrics. Similarly, gaming applications consist of a large number of path-planning, graph-walking, and physics-simulation algorithms (respective examples

— artificial intelligence; a scene-graph representing relationships between game-world entities; and collision detection, special effects, and realistic mechanics). Each of these types of algorithms have a large number of parameters that make trade-offs between the *realism exhibited*, the *modeling granularity* of the game world, and the computational complexity. Therefore, parameters that have a dominant, monotonic effect on various QoS characteristics can again be expected to be common in gaming.

Benefit: the monotonic response is vital for a feedback control scheme to be applicable.

#2 Existence of a sliding window within which QoS deviations are not perceptible. The interactive user of immersive applications may not perceive a degradation in QoS if the degradation happens only for a short sequence of frames. Consider, for example, the frame rate in a fast-action game or video playback. The instantaneous frame rate is not perceptible to the user. The instantaneous frame rate could occasionally drop significantly below the desired rate and the user would not notice this provided the frame rate recovered quickly. The *perceived frame rate* is determined by the current moving average of the frame times of the previous few frames. Fast-action games and high-quality video feeds typically have a frame rate around 30 frames-per-second. The perceived frame rate can be estimated as a moving average of the previous *five-to-ten* frames, depending on the specific attributes of the application. Let's refer to the length of this moving-average window as the *sliding window*, W, for that application.

Typical video encoding schemes divide the frames into groups of *intra-coded* I, *predicted* P, and *bidirectionally-predicted* B frames, each with widely different computation complexity that guarantees very large frame-to-frame variations in frame-time. For example, MPEG2 encoding commonly uses a repeating Group-of-Pictures (GoP) of seven frames — I-P-B-B-P-B-B. Therefore, we use $W = 7$ with our MPEG2 encoding benchmark.

Apart from the frame rate, there could be other QoS characteristics that are perceptible to the user. In a video game with a lot of simulated combatants (bots), the level of *intelligence* exhibited by the bots would be perceptible to the player. Again, W places a limit on how quickly any change is perceptible to the user. Therefore, per-frame variations in exhibited artificial intelligence would not be perceptible in durations less than W frames. In fact, only the cumulative intelligence exhibited over a duration of W^{AI} frames, where W^{AI} is significantly longer than the W required to smoothen perceived frame-rate, would be noticed by the player. Generally speaking, there can be different window lengths W_i associated with different QoS metrics y_i (i.e., y_i is actually a moving average of a QoS metric over a window of W_i frames). However, for simplicity we restrict our discussion to a single W, the shortest of the various W_i, which would typically be dictated by the frame-rate. Depending on the nature of the application and the intended use-case, we expect the programmer to specify a suitable W to the controller at the start of application execution. Applications with no sliding window would have $W = 1$.

Benefit: knowledge of W helps the controller distinguish between frame-to-frame "noise" (which may be impossible for a controller to limit — like over an MPEG2 GoP sequence, and is not perceptible to the user anyway), and the actual deterioration or improvement in the QoS performance based on specific changes made by the controller in its control policy.

#3 With high probability, the application's $\vec{x} - \vec{y}$ response characteristics remain stable for durations of frames $\gg W$. Immersive applications have the goal of captivating the interactive user in a rich and engrossing simulated world. The richness of the simulated world may come not just from visual fidelity or rendering detail — the richness may be associated with the complexity and the realism of the interactions between the game-world objects (say, realistic distortions and rebound effects when objects made of different materials collide in a game, requiring sophisticated physics modeling; or, sophistication of artificial intelligence during bot interactions). Regardless of the form of the richness, the richness of the simulated world unfolds in a smooth continuous manner to the interactive user over a sequence of frames. *Our third observation is that immersive applications typically don't undergo dramatic changes to their simulated-world state representations in time frames that are too short to be perceptible to the interactive user.* The implication of this observation is that the simulated-world state, and therefore the application's $\vec{x} - \vec{y}$

response characteristics, are unlikely to change dramatically over frame sequences of length less than W. In other words, we can expect to frequently encounter long sequences of frames (of length $\gg W$) over which the application characteristics stay mostly unchanged (stable).

It is certainly possible to implement an immersive application whose characteristics change more rapidly than W frames. However, we do not see this in practice in any number of gaming, video, and computer vision applications that we have examined. Further, such an implementation often reflects a poor application design as the simulated world's transitions experienced by the interactive user are no longer smooth and continuous.

Benefit: The controller incurs overheads in estimating and tracking the current $\vec{x}$-$\vec{y}$ response characteristics of the application. In particular, the controller has to sacrifice getting high QoS on a few frames in order to explore drifts in the application's behavior. The controller can deliver higher QoS performance if the sacrificed frames are mitigated over a longer sequence of frames exhibiting the same application behavior.

#4 Range for $\vec{y}$ values remains fixed over long durations of the application execution. While the x_j values are bounded by the problem definition, y_i may take arbitrary values determined by the application's algorithms and the particular data set. During a single execution of the application, we expect the same set of algorithms to be deployed over long sequences of frames and expect the data being processed to be similar frame-to-frame. Of course there can be frame-to-frame differences, even large differences like with the MPEG2 GoP processing structure. However, we expect the overall ensemble of algorithms and data to remain similar over sufficiently long sequence of frames. For example, on a given compute platform, encoding 640×480 resolution video would exhibit a particular range of computational complexity (frame-time) very distinct from the computational complexity with 320×240 resolution video.

Benefit: Knowledge of the typical $\vec{y}$ range enables the efficient implementation of a number of estimation techniques within the controller. The controller is able to detect the transitions and act appropriately if significant changes to the range occur only after long durations of frames.

Practical implications. Programmers need to certify that their applications are indeed immersive applications in order to expect our controller to deliver QoS improvement. The above four observations correspond to the four domain assumption requirements (Section 5.4) that the programmer has to certify apply to the application before using the controller. The controller can tolerate a *soft satisfaction* of these requirements, as follows.

- Satisfied with moderate to high frequency — for example with the stable response characteristics requirement.
- Satisfied to a lesser extent — for example the programmer-specified W may be a rough estimate, monotonicity may be violated to a minor extent over sub-ranges of some x_j.

We intend the controller to be **robust** — the QoS performance should only deteriorate gradually over local sequences of frames when the requirements are only soft satisfied, and only to the degree the requirements are violated. The controller relies on *randomization* and uses *noise-tolerant metrics* to achieve robustness.

5.7 Use of LLSE and LQR

Here we discuss how LLSE is used for model estimation and LQR for regulator design using the estimated model. However, both LLSE and LQR rely on a number of *structuring parameters* that in their traditional offline-use setting are chosen by the human designer. Structuring parameters for LLSE include the regularization parameter λ, and for LQR include time-horizon N and cost-matrices Q, Q_f and R. Additionally, LLSE relies on suitable training data to be already available. Appropriate choice of the structuring parameters and suitable generation of the training data prove to be vital for achieving high QoS performance.

Section overview. First, Section 5.7.1 motivates a linear model form suitable for capturing an application's $\vec{x} - \vec{y}$ response characteristics. Second, Section 5.7.2 summarizes the linear least-squares estimation technique. Third, Section 5.7.3 demonstrates how LLSE can be applied to estimate our linear model form using the $\vec{x}$ and $\vec{y}$ data samples observed over the past few frames. Fourth, Section 5.7.4 summarizes the LQR controller-design methodology for any given linear dynamical system model. Finally, Section 5.7.5 shows how the estimated $\vec{x}-\vec{y}$ linear model can be converted to a dynamical system model, and a feedback controller determined for it.

Section 5.8 identifies the challenges that must be addressed for an effective controller to be constructed for immersive applications based on LLSE and LQR, with Chapter 6 providing detailed solutions to the identified challenges.

5.7.1 Linear Model of Application

The application executes as a sequence of frames denoted $t = 1, 2, \ldots$. A new linear model for the application's $\vec{x}-\vec{y}$ response characteristics is periodically estimated at certain frames. At frame $t = t_0$, we estimate the model using the values of $\vec{x}$ and $\vec{y}$ recorded from the previous frames, a *history of application-response behavior* $\mathcal{H}$. $|\mathcal{H}|$ denotes the *history length* of the application response retained for model-estimation purposes. The model to be estimated has the following affine form:

$$\hat{y}_{i|t_0} = \sum_{j=1}^{n} \sum_{r=0}^{x_{ord}} g_{ijr}\ x_{j|t_0-r} + \sum_{j=1}^{m} \sum_{r=1}^{y_{ord}} g'_{ijr}\ y_{j|t_0-r} + c_i\,. \tag{7}$$

The above model is an *affine estimator* of y_i at $t = t_0$ in terms of the past observed values of $\vec{x}$ and $\vec{y}$. The $\hat{y}_{i|t_0}$ notation represents the value *predicted by the model* for y_i at $t = t_0$. In contrast, the value actually *observed* for y_i at $t = t_0$ is denoted by $y_{i|t_0}$.

Next, let us rewrite Eq 7 to allow construction of a model form more amenable to LLSE. Define sequences:

$$\vec{q_i}^{\mathrm{T}} \triangleq [\ g_{ijr} \mid 1 \le j \le n,\ 0 \le r \le x_{ord}\ ] \mathbin{++} \left[\ g'_{ijr} \mid 1 \le j \le m,\ 1 \le r \le y_{ord}\ \right] \mathbin{++} [\ c_i\ ]\ , \tag{8}$$

$$\vec{p}_{i,t_0}^{\mathrm{T}} \triangleq \left[\ x_{j|t_0-r} \mid 1 \le j \le n,\ 0 \le r \le x_{ord}\ \right] \mathbin{++} \left[\ y_{j|t_0-r} \mid 1 \le j \le m,\ 1 \le r \le y_{ord}\ \right] \mathbin{++} [\ 1\ ]\ . \tag{9}$$

Here, $\vec{q_i}$ is a column vector consisting only of the model coefficients to be estimated, and

$\vec{p}_{i,t_0}$ is a column vector consisting only of observations from $\vec{x}$ and $\vec{y}$ that are relevant for estimating $y_{i|t_0}$. The $+\!+$ operator above denotes *sequence concatenation.*

Therefore, Equation 7 can be rewritten as the following *dot-product*:

$$\hat{y}_{i|t_0} = \vec{p}_{i,t_0}^{\,\mathrm{T}} \, \vec{q}_i = \vec{p}_{i,t_0} \cdot \vec{q}_i \,. \tag{10}$$

The estimate at $t = t_0$ for the entire vector $\vec{y}$ can now be expressed as

$$\hat{\vec{y}}_{t_0} = \underbrace{\begin{bmatrix} \vec{p}_{1,t_0}^{\,\mathrm{T}} & 0 & 0 & \cdots & 0 \\ 0 & \vec{p}_{2,t_0}^{\,\mathrm{T}} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & \vec{p}_{m,t_0}^{\,\mathrm{T}} \end{bmatrix}}_{P_{t_0}} \underbrace{\begin{bmatrix} \vec{q}_1 \\ \vec{q}_2 \\ \vdots \\ \vec{q}_m \end{bmatrix}}_{\vec{q}} . \tag{11}$$

Therefore, the **linear model for the application response** can be written as

$$\boxed{\hat{\vec{y}}_{t_0} = P_{t_0} \, \vec{q}} \,. \tag{12}$$

Note that P_{t_0} consists entirely of relevant past observations of $\vec{x}_t$ and $\vec{y}_t$ (over appropriate $t \leq t_0$), whereas $\vec{q}$ consists of all the model coefficients to be estimated, and does not vary with t_0.

5.7.2 Linear Least-Squares Estimation (LLSE)

Given a $k \times l$ matrix A specifying the following *transform*:

$$\vec{v} = A \, \vec{u} \,, \tag{13}$$

and given a desirable result $\vec{v} = \vec{v}_{\texttt{des}}$, the linear least-squares estimation problem is to determine a suitable $\vec{u} = \vec{u}_{ls}$ that will *minimize* the squared-error cost $\|\vec{v}_{\texttt{des}} - A \, \vec{u}_{ls}\|^2$.

The solution to the LLSE problem is given by $\vec{u}_{ls} = A^{\dagger} \, \vec{v}_{\texttt{des}}$. $A^{\dagger}$ is called the *Moore-Penrose pseudo-inverse* of A, and is defined as $A^{\dagger} = (A^T A)^{-1} A^T$. However, the pseudo-inverse is only defined if A has *full column-rank.* Full column-rank is equivalent to the columns of A being linearly independent (which implies $k \geq l$). In general, A may not satisfy this requirement, hence the LLSE problem is extended to the *regularized LLSE* to allow for arbitrary A.

The regularized LLSE problem takes the following form:

$$\underbrace{\begin{bmatrix} \vec{v} \\ 0 \end{bmatrix}}_{\vec{v'}} = \underbrace{\begin{bmatrix} A_{k\times l} \\ \sqrt{\lambda}\; I_{l\times l} \end{bmatrix}}_{A'_{(k+l)\times l}} \vec{u}\,. \tag{14}$$

Or,

$$\vec{v'} = A'\vec{u}\,.$$

Here, $I_{l\times l}$ is the $l \times l$-dimensional identity matrix and $A_{k\times l}$ is an arbitrary matrix with no restrictions placed on it. Since $I_{l\times l}$ has full column-rank, the extended matrix $A'_{(k+l)\times l}$ will also have full column-rank, regardless of choice or dimensions of $A_{k\times l}$.

Now, the solution to the regularized LLSE problem is given by

$$\vec{u}_{ls} = (A')^{\dagger} \begin{bmatrix} \vec{v}_{\mathtt{des}} \\ 0 \end{bmatrix}. \tag{15}$$

This solution minimizes the squared-error cost $\|\vec{v}_{\mathtt{des}} - A\ \vec{u}_{ls}\|^2 + \lambda\|\vec{u}_{ls}\|^2$. Minimizing this cost allows us to prioritize finding a *small-magnitude solution* $\vec{u}_{ls}$ with priority specified by $\lambda > 0$.

5.7.3 Using LLSE to estimate Linear Model for Application Response

We would like to estimate the linear model for the application response ($\vec{q}$ in Equation 12) using the observed history data $\mathcal{H}$. For this we construct the following *estimation form*:

$$\underbrace{\begin{bmatrix} \vec{y}_{t_0} \\ \vec{y}_{t_0-1} \\ \vdots \\ \vec{y}_{t_0-h} \end{bmatrix}}_{\vec{v}_{\mathtt{des}}} = \underbrace{\begin{bmatrix} P_{t_0} \\ P_{t_0-1} \\ \vdots \\ P_{t_0-h} \end{bmatrix}}_{A} \vec{q}\,. \tag{16}$$

Here, h is chosen $\leq |\mathcal{H}| - \max\{x_{ord},\ y_{ord}\}$. Hence, we get the form $\vec{v}_{\texttt{des}} = A\,\vec{q}$, where $\vec{v}_{\texttt{des}}$ is constructed from past observations of $\vec{y}$, and A is constructed from past observations of $\vec{x}$ and $\vec{y}$, as shown above. We now apply the regularized LLSE technique from the previous section to determine a $\vec{q} = \vec{q}_{ls}$ that minimizes the *model-fit-error* of estimation at $t = t_0$: $\epsilon_{t_0} = \|\vec{v}_{\texttt{des}} - A\,\vec{q}_{ls}\|^2 + \lambda\|\vec{q}_{ls}\|^2$. An appropriate choice of λ is vital for robust model estimation. Traditional techniques use manual offline analysis to determine a suitable λ [72, 73, 74]. Section 6.2.6 covers how we can tune λ automatically in an online controller.

Hence, our estimated linear model becomes $\hat{\vec{y}}_t = P_t\ \vec{q}$, with $\vec{q} = \vec{q}_{ls}$ estimated using regularized LLSE as described above.

5.7.4 LQR Regulator Design

Consider the following general representation of a *discrete-time linear dynamical system*:

$$\vec{s}_{t+1} = A_{k\times k}\ \vec{s}_t + B_{k\times l}\ \vec{u}_t\,. \tag{17}$$

Here, $\vec{s}_t$ represents the k-dimensional current state of the system (for $t = 0, 1, 2, 3, \ldots$). A is the *state-transition* matrix, and B is the *input-sensitivity matrix*.

The LQR design problem for a discrete-time linear dynamical system given by Eq 17 is as follows. Given:

1. an arbitrary initial state $\vec{s}_0$,
2. a *state-cost* matrix Q with $Q^T = Q \geq 0$,
3. a *final state-cost* matrix Q_f with $Q_f^T = Q_f \geq 0$, and
4. an *input-cost* matrix R with $R^T = R > 0$,

determine a sequence of inputs $\vec{u}_0, \vec{u}_1, \cdots, \vec{u}_N$ over a *horizon* of N time-steps, that drives the final-state $\vec{s}_N$ close to *zero*, while minimizing the following *quadratic cost function*:

$$J = \sum_{t=0}^{N-1} \left(\vec{s}_t^T\, Q\, \vec{s}_t + \vec{u}_t^T\, R\, \vec{u}_t\right) \ +\ \vec{s}_N^T\, Q_f\, \vec{s}_N\,. \tag{18}$$

The solution to the LQR problem [56] consists of the input sequence $\vec{u}_t := -K_t\vec{s}_t$ for $t = 0, 1, \cdots, N$. Here $K_t := \left(R + B^T W_{t+1} B\right)^{-1} B^T W_{t+1} A$. The W_t matrices are computed as a *backwards recursion in time* as follows (Riccati equation):

$$W_t := A^T \left[W_{t+1} - W_{t+1}B\left(B^T W_{t+1} B + R\right)^{-1} B^T W_{t+1}\right] A + Q\,, \tag{19}$$

for $t = N-1, \cdots, 0$, starting with $W_N := Q_f$.

Let's make the following observations about the LQR controller-design process:

1. All the W_t matrices (for $t = 0, 1, \cdots, N-1$) have to be computed *in advance* before the input for the first time-step u_0 can be computed. In subsequent time-steps, the input for that time-step, u_t, can be computed relatively efficiently since all the W_t's are already computed. Hence, if LQR were to be done online (i.e., during application execution), it would load up the first time-step with the compute-intensive recursion on W_t for all N time-steps, instead of amortizing this load over multiple time-steps.

2. Eq 19 is known to typically converge rapidly. Hence, for longer horizons N, it would be possible to bound the cost of computing W_t's, rather than incurring a cost proportional to N.

The above discussion summarizes the LQR controller-design process when the goal is to drive the system state $\vec{s}_t$ to zero in N time-steps. However, as will be seen in the next subsection, our goal is to have *parts* of the system state converge to a *desired state-trajectory.* This goal is achieved by employing a generalization of LQR called **trajectory-tracking LQR**.

The trajectory-tracking LQR controller-design problem attempts to minimize the *tracking error* between the observed system state $\vec{s}_t$ and a *desired state-trajectory* $\vec{r}_t$, $t = 0, 1, \cdots, N$. Strictly speaking, only a subset of the elements in the state vector $\vec{s}_t$ may be relevant for computing cost. For example, a missile's state vector would include elements for position, velocity and acceleration. However, only position (for targeting) and acceleration (for fuel burn) may be relevant towards minimizing cost. Hence, the trajectory-tracking problem allows for a *transform of the state* $C\vec{s}_t$ to track trajectory $\vec{r}_t$. Note that

$C\vec{s}_t$ may be a vector of far lower dimension than $\vec{s}_t$, corresponding to dropping dependence on a large number of terms in $\vec{s}_t$. The dimension and elements of $\vec{r}_t$ will now correspond to the elements of vector $C\vec{s}_t$. The trajectory-tracking problem seeks to determine an input sequence $\vec{u}_0, \cdots, \vec{u}_{N-1}$ that minimizes the following quadratic cost function:

$$J = \sum_{t=0}^{N-1} \left[(C\vec{s}_t - \vec{r}_t)^T Q \, (C\vec{s}_t - \vec{r}_t) + \vec{u}_t^T R \, \vec{u}_t \right] \; + \; (C\vec{s}_N - \vec{r}_N)^T Q_f \, (C\vec{s}_N - \vec{r}_N) \, . \tag{20}$$

Note that the cost matrices Q and Q_f are of dimensions that correspond to the transformed state $C\vec{s}_t$, and the costs are applied to the *tracking error* $(C\vec{s}_t - \vec{r}_t)$ rather than to the transformed state values themselves.

The solution to the trajectory-tracking LQR problem has a somewhat more complicated form [56], summarized below.

$$K_t := (B^T W_{t+1} B + R)^{-1} B^T W_{t+1} A \tag{21}$$

$$W_t := A^T W_{t+1} (A - BK_t) + C^T Q C \, , \qquad \text{with } W_N := C^T Q_f C \tag{22}$$

$$\vec{v}_t := (A - BK_t)^T \vec{v}_{t+1} + C^T Q \vec{r}_t \, , \qquad \text{with } \vec{v}_N := C^T Q_f \vec{r}_N \tag{23}$$

$$K_t^v := (B^T W_{t+1} B + R)^{-1} B^T \tag{24}$$

$$\vec{u}_t := -K_t \vec{s}_t + K_t^v \vec{v}_{t+1} \tag{25}$$

Here, $\vec{u}_t$ consists of not just the *negative state feedback* via K_t, but also a *feed-forward gain* K_t^v applied to a function of the desired trajectory $\vec{r}_t$. Despite the more complicated forms involved, the observations made earlier still hold true: **i)** the *backwards recursion* (now on W_t *and* $\vec{v}_t$) for the entire $t = N-1, \cdots, 0$ has to be fully computed at $t = 0$, and **ii)** the recursion on W_t tends to converge rapidly (it's essentially identical to Equation 19). Note that the cost of recursing on $\vec{v}_t$ is low in comparison due to matrices being multiplied only with vectors.

5.7.5 Application QoS Control using LQR

LLSE was used in Subsection 5.7.3 to estimate the application's $\vec{x}-\vec{y}$ response characteristics at any time $t = t_0$ in the following form (Equation 12):

$$\hat{\vec{y}}_{t_0} = P_{t_0}\, \vec{q}\,.$$

This linear model for the $\vec{x} - \vec{y}$ response characteristics can rewritten as:

$$\hat{\vec{y}}_{t_0} = L \begin{bmatrix} \vec{y}_{t_0-1} \\ \vdots \\ \vec{y}_{t_0-y_{ord}} \\ \vec{x}_{t_0} \\ \vec{x}_{t_0-1} \\ \vdots \\ \vec{x}_{t_0-x_{ord}} \\ 1 \end{bmatrix}, \tag{26}$$

for an appropriate matrix L constructed using elements of $\vec{q}$.

Note the following about Equation 26 as a contrast with Equation 12: the past observations of $\vec{x}_t$ and $\vec{y}_t$ are now explicit instead of being arranged in P_{t_0}, and the model coefficients are arranged in L instead of $\vec{q}$.

Let us rewrite Equation 26 at time-step t as follows:

$$\hat{\vec{y}}_t = L \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \\ \vec{x}_t \\ \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \\ 1 \end{bmatrix} = \left[\begin{array}{c|c|c|c} L_1 & L_2 & L_3 & L_4 \end{array}\right] \left[\begin{array}{c} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \\ \hline \vec{x}_t \\ \hline \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \\ \hline 1 \end{array}\right]$$

$$= L_1 \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \end{bmatrix} + L_2\, \vec{x}_t + L_3 \begin{bmatrix} \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \end{bmatrix} + L_4 \begin{bmatrix} 1 \end{bmatrix}$$

$$= L_1 \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \end{bmatrix} + L_2\, \vec{x}_t + L_3 \begin{bmatrix} \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \end{bmatrix} + L_4\,. \tag{27}$$

L_2 is constructed as follows:

for $i \in [1, m]\,,\ k \in [1, n]\,,$

$$(L_2)_{ik} = g_{ijr}\,,$$

$$\text{where } j = k,\ r = 0\,, \tag{28}$$

and, L_3 is constructed as follows:

for $i \in [1, m]\,,\ k \in [1, n\,x_{ord}]\,,$

$$(L_3)_{ik} = g_{ijr}\,,$$

$$\text{where } j = ((k-1) \mod n) + 1,\ r = \frac{k-1}{n} + 1\,. \tag{29}$$

The g_{ijr} terms are already packed into $\vec{q}$ from the linear application-response model. The index of a g_{ijr} term in $\vec{q}$ is a simple lookup defined by Equation 8 and Equation 11.

L_1 is constructed as follows:

for $i \in [1, m]$, $k \in [1, m\, y_{ord}]$,

$$(L_1)_{ik} = g'_{ijr}\,,$$
$$\text{where } j = ((k-1) \mod m) + 1,\ r = \frac{k-1}{m} + 1\,. \tag{30}$$

The g'_{ijr} terms are extracted from $\vec{q}$ in the same manner as described above for the g_{ijr} terms.

L_4 is simply a column vector of all the affine constants from Equation 7:

$$L_4 = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}. \tag{31}$$

We need to determine a sequence of inputs $\vec{x}_t$ over some time horizon N (i.e., determine $\vec{x}_{t_0}, \vec{x}_{t_0+1}, \cdots, \vec{x}_{t_0+N-1}$), that would cause the observed application QoS metrics $\vec{y}_t$ to converge to the *programmer-specified QoS objective* $\tilde{\vec{y}}$ (vector of individual $\tilde{y}_i$ objectives from the problem specification in Section 5.4) within N time-steps after $t = t_0$. The trajectory-tracking LQR controller-design procedure requires the system characteristics to be expressed as a linear dynamical system of the following form (Equation 17):

$$\vec{s}_{t+1} = A\,\vec{s}_t + B\,\vec{u}_t\,.$$

Take

$$\vec{u}_t = \vec{x}_t \qquad \text{and} \qquad \vec{s}_t = \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \\ \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \\ 1 \end{bmatrix}. \tag{32}$$

Therefore, we need to define appropriate A and B matrices that satisfy Equation 17 for the choice of $\vec{s}_t$ given above in Equation 32.

Define a special *block-shift* matrix B_s, and a *block-selector* matrix S, that satisfy the following:

$$B_s\ \vec{s}_t = \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}+1} \\ 0 \\ \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}+1} \\ 1 \end{bmatrix}, \text{ and } S\ \vec{x}_t = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vec{x}_t \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}. \tag{33}$$

Therefore,

$$B_s\ \vec{s}_t + S\ \vec{x}_t = \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}+1} \\ \vec{x}_t \\ \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}+1} \\ 1 \end{bmatrix}. \tag{34}$$

Let S_1, S_3 and S_4 be additional *block-selector* matrices that satisfy the following:

$$S_1\ \vec{s}_t = \begin{bmatrix} \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-y_{ord}} \end{bmatrix},\ S_3\ \vec{s}_t = \begin{bmatrix} \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-x_{ord}} \end{bmatrix}, \text{ and } S_4\ \vec{s}_t = \begin{bmatrix} 1 \end{bmatrix}. \tag{35}$$

Therefore, comparing Equation 35 against the definition of $\vec{s}_t$ in Equation 32, we get

$$\begin{bmatrix} S_1 \\ S_3 \\ S_4 \end{bmatrix} \vec{s}_t = \vec{s}_t\,. \tag{36}$$

Now, starting with a time-shifted Equation 32, and substituting using Equation 34, Equation 27, and Equation 35,

$$\vec{s}_{t+1} = \begin{bmatrix} \vec{y}_t \\ \vec{y}_{t-1} \\ \vdots \\ \vec{y}_{t-u'_{\mathtt{max}}+1} \\ \vec{x}_t \\ \vec{x}_{t-1} \\ \vdots \\ \vec{x}_{t-u^{\mathtt{max}}+1} \\ 1 \end{bmatrix} = \begin{bmatrix} \vec{y}_t \\ B_s\vec{s}_t + S\vec{x}_t \end{bmatrix} = \begin{bmatrix} L_1S_1\vec{s}_t + L_2\vec{x}_t + L_3S_3\vec{s}_t + L_4S_4\vec{s}_t \\ B_s\vec{s}_t + S\vec{x}_t \end{bmatrix}$$

$$= \begin{bmatrix} (L_1S_1 + L_3S_3 + L_4S_4)\vec{s}_t + L_2\vec{x}_t \\ B_s\vec{s}_t + S\vec{x}_t \end{bmatrix}.$$

Therefore, we get A and B for Equation 17 as follows.

$$\vec{s}_{t+1} = \underbrace{\begin{bmatrix} L_1S_1 + L_3S_3 + L_4S_4 \\ B_s \end{bmatrix}}_{A} \vec{s}_t \quad + \quad \underbrace{\begin{bmatrix} L_2 \\ S \end{bmatrix}}_{B} \vec{u}_t. \tag{37}$$

Given that S_1, S_3 and S_4 partition the rows of $\vec{s}_t$ into a top, middle and bottom block, respectively, A can be re-written as follows.

$$A = \left[\begin{array}{c|c|c} L_1 & L_3 & L_4 \\ \hline \multicolumn{3}{c}{B_s} \end{array}\right] \tag{38}$$

Recall from Subsection 5.7.4, the trajectory-tracking LQR's cost function and solution does not depend directly on the state, but on a transformed state $C\,\vec{s}_t$. We want the *desired trajectory* $r_t = \tilde{\vec{y}}$, i.e., a constant trajectory. Hence, C must be constructed so that $C\vec{s}_t$ produces the linear estimate of the QoS metrics, $\hat{\vec{y}}_{t-1}$. Note, in general, we *cannot* have $C\vec{s}_t = \hat{\vec{y}}_t$ as the current state $\vec{s}_t$ cannot contain the current input $\vec{x}_t$. Hence, choose $C = [I\,0\,0\,\cdots\,0]$, which satisfies $C\vec{s}_t = \hat{\vec{y}}_{t-1}$.

Note that when $\vec{s}_t$ is used by the LQR controller for *state-feedback* to determine control input $\vec{u}_t$ (as per Equation 25), we need to use the *observed values* for the past QoS metrics $\vec{y}_{t-r}$ in constructing $\vec{s}_t$ as per Equation 32. This ensures that the LQR controller performs *feedback-control* by taking into account the *actual deviations* of the observed trajectory from the desired trajectory (Equation 25). In contrast, the derivation of the linear dynamical model (A, B) from the linear response model (L), given above by Equation 37, required the use of the *predicted values* of the QoS metrics $\hat{\vec{y}}_t$ that were predicted by the linear response model itself (Equation 26).

Now that the dynamical system model is set up, our choices for the matrices A, B, and C, and the desired trajectory $\vec{r}_t$ can be plugged into Equations 21-25 to yield the input sequence that must be applied over the next N frames: $\vec{x}_{t_0}, \vec{x}_{t_0+1}, \cdots, \vec{x}_{t_0+N-1}$. However, we still need to specify the LQR cost matrices Q, Q_f, and R, and the length of the horizon N. Further, the problem specification in Section 5.4 posed an *input-bounds constraint* on the elements of $\vec{x}$ (and hence on each of the $\vec{u}_t$ above): $-N_j \leq x_j \leq N_j$. We have so far not imposed this constraint in any way on the $\vec{u}_t$ sequence. Strictly speaking, if the control input requires $x_j > N_j$ or $x_j < -N_j$, we will simply *clip* $x_j = N_j$ or $x_j = -N_j$, respectively, when applying the input to the application. However, an LQR solution that entirely (or mostly) respects input-bounds constraints is far more likely to be effective at bringing about the intended QoS correction, since the input applied to the application would be faithful both to the range of inputs supported by the application as well as to the linear model used to drive LQR.

5.8 Challenges in the use of LLSE and LQR with Immersive Applications

The previous section demonstrated the use of LLSE and LQR for QoS control in a general setting where the relationship between inputs $\vec{x}$ and outputs $\vec{y}$ needs to be discovered and a regulator constructed. However, with immersive applications, a number of additional problems have to be solved for *i)* LLSE to estimate "good quality" models, *ii)* fully define the *structure* and *cost function* of the LLSE model to be estimated and the LQR regulator to be constructed, and *iii)* improve the robustness of the LQR controller in the face of modeling imprecision and frame-to-frame noise. This section identifies the specific problems. Chapter 6 provides the solutions.

5.8.1 Online Generation of Training Data.

The controller's QoS performance depends on the accuracy of the model estimated by LLSE and on the suitability of the regulator produced by LQR using the estimated model. The accuracy of the estimated model depends on the *quality* of the training data. The quality encompasses several aspects:

- *Is the training data representative of the current application behavior?* The application behavior can change frequently, constrained only by the stable-response-period domain assumption.
- *Does the training data represent the application-response characteristics over the full input range of $\vec{x}$ or over only a narrow sub-range of $\vec{x}$?* The regulator designed with LQR would have the ability to exercise the full input range of $\vec{x}$, which can produce poor control performance if the estimated model did not adequately characterize parts of the input range.

The first challenge is for the controller to ensure that only high-quality training data is used to estimate models. In Section 6.2, we create metrics and estimation schemes that address this challenge.

In our problem definition, the training data is generated entirely online during application execution — the applied $\vec{x}_t$ and observed $\vec{y}_t$ for a sequence of frames t. This requires the controller to simultaneously choose application inputs $\vec{x}_t$ that maximize QoS and also produce high-quality training data, with the two being contradictory requirements. Note that prioritizing the delivery of the highest-possible QoS using the current estimated model has two adverse effects: *i)* the discovery of more accurate models is precluded, and *ii)* the controller is less able to detect changes in application behavior and re-estimate more representative models in a timely manner.

Therefore, the second challenge is for the controller to balance the two contradictory requirements of producing high-quality training data for model estimation and maximizing the QoS delivered by the existing model. In Section 6.4 we define a quantitative criteria for such a balance, and create algorithms and metrics that achieve this balance.

The third challenge is for LLSE to estimate *robust* models from the limited noisy training data — namely, avoid overfitting. In Section 6.2.6 we create an online estimation scheme that determines a suitable value for the LLSE regularization parameter λ to allow robust model estimation.

5.8.2 Improving tolerance of LQR to Approximate Models.

A regulator produced by LQR is expected to perform well when LQR is applied to an accurate model of a system with linear response characteristics. Our problem definition does not require linearity in the system response characteristics, only monotonicity (due to the nature of immersive applications). Further, only a limited amount of training data is available to an online controller, and this data may be noisy given the nature of immersive applications and the likelihood that $\vec{y}$ is impacted by factors additional to the $\vec{x}$ identified by the programmer. The linear model estimated by LLSE is likely to be quite approximate for these reasons, deteriorating the QoS performance possible from an LQR-constructed regulator.

In Section 6.5.2 we apply the adaptive-integral univariate controller from Chapter 4 to the *tracking error* observed in each y_i. This approach enhances the QoS performance delivered by the controller beyond what was possible with LQR alone.

5.8.3 Determination of LQR structuring parameters.

The primary challenge arising from our problem definition is the construction of the input-costs matrix R in a manner that constrains the regulator to respect the input-bounds constraints $-N_j \leq x_j \leq N_j$, and yet have the regulator retain the ability to pick a value for x_j from the full range $[-N_j, N_j]$ that maximizes QoS. To address this challenge, we choose a strategy where LQR is repeatedly performed using the same estimated model to repeatedly construct regulators, but with the entries of R iteratively tuned across the invocations of LQR based on the control inputs $\vec{x}$ produced by the last instance of the regulator. Section 6.5 provides the details.

An important observation — software applications typically do not incur an inherent "cost" when applying large magnitude input values (unlike a physical system, where for example a control input might represent fuel burn rate in a rocket). Therefore, there is no inherent "good value" for R, except what allows high QoS performance. At any rate the only cost involved in our problem definition is the minimization of QoS error. If programmers really need to explicitly constrain input costs within the framework of our problem definition, they can specify a QoS output $y_i = x_j$ with range objective $|y_i - \tilde{y}_i| < \delta_i$ for their application. The range objective would indirectly place a cost to choosing values for x_j that differ from $\tilde{y}_i$, in addition to the usual input-bounds hard-constraint on x_j.

Choice of N. We use $N = 1$ as the controller will repeatedly construct a new regulator using LQR, perhaps multiple times within a single frame t. Using $N = 1$ constructs a regulator that applies the maximal magnitude control inputs necessary to bring about the desired control correction in one time-step. We separately use the R matrix to limit the inputs to the input-bounds constraints.

Choice of Q. The choice of the transformed-state ($C\,\vec{s}_t$) cost matrix Q is dictated by the per-frame instantaneous QoS-performance optimization goal τ_t (Eq 5) as follows:

$$\tau_t = \frac{1}{m} \sum_{i=1}^{m} s_i \frac{|y_{i|t} - \tilde{y}_i|^2}{\delta_i^2} r_i \,.$$

Choosing Q to be a diagonal matrix with the following diagonal terms leads to LQR minimizing the cumulative per-frame performance cost for $t = 1..N$, which Eq 6 defined as $\texttt{MSEQ} = \frac{1}{T}\sum_{t=1}^{T} \tau_t$ (here $T = N$, the LQR horizon):

$$Q_{ii} = \frac{s_i\, r_i}{\delta_i^2} \,. \tag{39}$$

Here the definitions of s_i and r_i come from the extended problem definition in Section 5.5 and are not related to the dynamical system state $\vec{s}_t$ or the desired trajectory $\vec{r}_t$.

Choice of Q_f. We choose the final-state costs $Q_f = Q$ since we would need to keep controlling QoS indefinitely, and not end control after some final state is achieved in N frames.

Determining R. The input-cost matrix R is dynamically tuned over multiple LQR regulator-design steps, as detailed in Section 6.5.

CHAPTER VI

MULTI-VARIATE QOS CONTROL: DESIGN OF THE QOS CONTROLLER

The previous chapter motivated that the QoS control of immersive software applications could be readily modeled as a discrete-time control problem that relied on characterization at runtime of the observed application behavior. We rely on linear least-squares estimation (LLSE) to periodically characterize the application behavior as a linear model. The linear quadratic regulator (LQR) technique from optimal control theory allows the construction of a regulator from the estimated linear model. At each application frame, the regulator adjusts the application control parameters to drive the application QoS metrics towards the desired goal (i.e., feedback control). We proposed an application QoS controller that encompassed the activities of model estimation, regulator construction and feedback control, and the decision logic necessary to orchestrate these activities in a manner suitable for achieving high QoS with immersive software applications.

This chapter details the design of the application QoS controller. In general, the application QoS controller falls under the category of model-identification adaptive control (MIAC), where adaptive control is performed using models estimated at runtime. There are many alternative strategies to craft a controller under MIAC, the specific choice depending on the nature of the system being controlled and the optimization goals for the control problem. Section 6.1 explores the alternative strategies possible and our chosen strategy. Our chosen strategy requires specific technical challenges to be addressed. The subsequent sections of this chapter provide solutions to each of the technical challenges identified as part of our chosen strategy. The technical challenges capture the "decision logic" of the QoS controller, and provide solutions for all the problems not addressed by LLSE and LQR.

6.1 Design Strategies under Model-Identification Adaptive Control

The following questions explore the goals and strategies for crafting a controller under MIAC. We address the questions for our problem of interest, the *domain* of immersive software applications (rather than for a *specific application* with unknown parameters, as is usually the case with adaptive control problems).

Question 1 *Is specific information available for each problem instance the controller is applied to?*

The application QoS controller is provided as a pre-compiled software library. Each immersive application describes a specific QoS optimization problem to the controller using an application programming interface (API). The specific problems will differ on dimensionality, input bounds, and desired ranges for the QoS objectives. Additionally, the application provides a perception-window parameter indicating how quickly the user would perceive a QoS failure. Such specific information allows the controller to optimize for each application. However, the controller is designed based on the domain assumptions about immersive applications in Section 5.6, and there is no customization of the controller code for any given application. The controller API even allows an application to change the QoS problem mid-execution, though doing so briefly disrupts QoS control while the controller re-characterizes the application behavior for the new QoS problem.

Question 2 *When or how often should the model be estimated at runtime?*

This question relates to how quickly is a given estimated model obsoleted by changing application characteristics. Periodic model estimation (i.e., estimate every fixed number of frames) is simplest. However, periodic model estimation requires that a suitable period be known, either from the problem specification or the domain assumptions, or possibly estimated at runtime. *Too frequent* estimation creates the following problems.

- The controller must with greater frequency drive control inputs with the goal of generating sample data suitable for the next model estimation, at the cost of driving inputs for the primary goal of maximizing QoS.

- The controller incurs heavier runtime overheads, which may impact application QoS (particularly, maintaining desired frame-rate).

Too infrequent estimation would fail to track QoS when faced with changing application characteristics. Hence, determining when to re-estimate a model is crucial for achieving good QoS.

While the problem specification and the domain assumptions provide insufficient information to determine a period for model estimation, they do prove sufficient for constructing metrics that track the *prediction accuracy* of the current model. The model's current prediction accuracy is compared against its prediction accuracy when the model was estimated (to identify drift in application behavior) and also compared to a more recently estimated *substitute model*. If the substitute model is found to have a higher prediction accuracy over the most recent frames, it could be for two reasons: *i)* the application behavior has changed since the current model was estimated, and/or *ii)* better quality training data was used in the estimation of the substitute model. The metrics are designed to meet two opposing goals: *i)* determine quickly if either the current model or the substitute model is distinctly superior to the other, and *ii)* allow precise comparison over longer durations of frames when the prediction accuracies of the two models are close. Hence, the controller attempts to filter out poor substitute models quickly, it allows distinctly superior substitute models to be applied quickly, and it prevents model replacement under noisy or ambiguous circumstances.

Model estimation is repeated as soon as new sample data of sufficient quality becomes available after a previous model was discarded. In this manner, model estimation can occur frequently when it is beneficial (e.g., tracking changes in application behavior as they happen) and more slowly when it is not beneficial or actually harmful (e.g., replacing a consistently accurate model during temporary deviations in the application behavior).

Question 3 *How is the regulator updated when a new model is estimated?*

A newly estimated model must become the *active model* before it impacts the regulator. A new model becomes active either when there is no prior model present in the controller or when the new model is a substitute model with better prediction accuracy than the current active model. LQR is used to construct a regulator from the active model. However, the model provides insufficient information about all the parameters needed by LQR. In an offline-design setting, these additional parameters are provided by the human designers based on intuition about their control problem and through trial-and-error in evaluating constructed regulators on test data.

In our online-design setting, the controller estimates these additional parameters by dynamically tuning them based on the observed control properties of the regulator. The controller repeatedly re-designs the regulator from the same active model, performing a *directed search of the parameter space.* The control properties of the constructed regulators are characterized along the search path in the parameter space. The characterization is done simply by applying the regulator to the dynamical-system state for the current frame and evaluating the *suitability of the produced control inputs* against those produced by prior regulators. Such characterization allows iterative refinement over multiple regulator designs, both within the same application frame as well as across frames.

The online-design setting allows the controller to fine-tune the parameters to the specifics of the current operating conditions. In contrast, an offline-designed controller must pick single fixed values for the parameters that provide the best performance trade-offs over a wide range of operating conditions. Section 6.5 describes the additional parameters and the directed search mechanism with low and bounded overhead suitable for an online setting.

Question 4 *Is there model certainty? That is, can the most recent model be considered the best estimate of system behavior?*

MIAC controllers can use different strategies depending on whether model certainty can be assumed. Under model certainty, the most recently estimated model is considered the best predictor of the system behavior and all prior model information can be discarded when re-designing or updating the regulator. Under model uncertainty, noise in the training data or training data that has insufficiently excited a fully representative range of application behavior may allow only a compromised model to be estimated at the current time. But combination with prior modeling information would allow a more accurate model to be determined, even when the application behavior is gradually changing.

The QoS behavior of a typical immersive software application is the emergent behavior over a large number of underlying algorithms, algorithms whose characteristics are often heavily data-dependent. While the emergent behavior is often statistically stable over a sequence of frames, there can be large noise on a frame-to-frame basis. Model estimation via LLSE can filter out additive noise, but very frequently the noise is non-additive in nature due to intermittent application events such as user-interaction events or application functionality that is not executed every frame. Hence, the sample data may intermittently have large noise that cannot be filtered out during model estimation, and we are unable to assume model certainty.

Therefore, we use a hybrid approach for our controller: the new model fully replaces the prior model, but only if metrics indicate it to be superior.

Question 5 *How should the model structure be determined? Is this pre-determined or determined at runtime?*

The structure of the model needs to be fixed before LLSE can be invoked. The structure includes knowledge of input-output impact relationships, model order, and setting a regularization parameter to ensure *robust* LLSE (i.e., avoiding over-fitting to the sample data, where perturbations to the sample data produce non-trivial differences in the resulting models).

Impact Relationships. While many programmers or application experts may have knowledge limiting which input control parameters may impact which QoS objectives for their particular application, we have chosen to ignore this potential knowledge in order to place a lower burden of expertise on the programmers using our controller. In general, the emergent nature of the QoS behavior in immersive applications often makes it difficult for a programmer to accurately identify impact relationships.

We assume that all control inputs can potentially impact any of the QoS objective metrics. Hence, we always estimate a *dense-structure model.* With high-quality training data, we expect LLSE to discover the more precise impact relationships with no additional assistance from the problem specification. Hence, our controller separately ensures that the training data is *probabilistically* of high quality before invoking LLSE.

Model Order. We choose input-order $x_{ord} = 0$ and output-order $y_{ord} = 1$, by default. The effect of the older control inputs ($\vec{x}$'s) can be accounted for by their impact on the older QoS outputs ($\vec{y}$'s). The effect of older $\vec{y}$'s from the past W frames is partially accounted for in $\vec{y}_{t-1}$ due to the use of running averages as the samples saved in $\mathcal{H}$. Since the use of $W > 1$ provides a mechanism to sufficiently incorporate older $\vec{y}$'s, we do not explore $y_{ord} > 1$ in this work. Though, in general, an explicit $y_{ord} > 1$ would allow the model to capture additional detail beyond the running average captured due to $W > 1$.

Regularization. LLSE provides us the least-squared-error-fit model conforming to a required model structure, given some sample data. The sample data often proves insufficient to provide a unique model solution (akin to finding an inverse solution for a non-invertible matrix transform). A linear sub-space of solutions exists, out of which a hyperplane defines an infinite range of equally good solutions (i.e., each solution exhibits the exact same minimal squared-error in fitting the sample data). However, most or all of these models may have very large coefficients, producing very large terms whose difference matches the smaller-magnitude sample data. Such models are fitted to the noise in the sample data. Instead, we prefer models, often from a different hyperplane with a somewhat larger fit-error, whose coefficients produce terms of magnitude close to the sample data. These latter models *average-out the noise* (and the non-linearities) in the sample data and *generalize better* for capturing the application behavior in a linear form.

We apply a standard technique called regularization to LLSE so that LLSE provides a model solution with the above-mentioned desirable characteristics of generalizing better to future, as yet unseen, application sample data. However, regularized LLSE requires a regularization parameter to be provided. This parameter is traditionally determined by offline analysis, which is not feasible under MIAC applied to unknown applications. We develop a light-weight adaptive technique that quickly tunes the regularization parameter until it is appropriate for the current application, and continues fine-tuning as the application characteristics drift. Section 6.2.6 details the adaptive tuning of the regularization parameter.

Question 6 *How does the training data impact the quality of the model estimated?*

We consider the training data to be of high quality if it allows a model with high prediction-accuracy to be estimated. Our underlying assumption is that a more accurate linear model would produce a regulator delivering higher QoS. The training data consists of the control inputs applied and the corresponding QoS outputs observed over a sequence of frames. The following two aspects determine the quality of the training data:

- *Coverage* of the input space.
- How *representative* is the sequence of frames of the current application behavior?

Coverage. Coverage relates to the range exercised for each input variable and the sampling density/distribution over the exercised range. If the training data samples a limited range of an input variable, the estimated model may poorly represent the behavior outside this range. Further, the behavior may perhaps vary considerably over different parts of the input range, even if the behavior response is considered "smooth". In general, dense sampling over the entire input volume would allow an accurate linear model to be estimated, but with no a priori indication of how dense the sampling needs to be (as the system response is unknown). Unfortunately, dense sampling at runtime runs counter to our controller's main objective of driving the input control variables in a manner that best enhances the output QoS.

With immersive applications, we make the assumption of monotonic response. Under this assumption, sampling a large range for each input variable is of primary importance. The sampling density and distribution within the large range is of secondary importance, as this information can be interpolated more accurately when we assume a smooth monotonic response (compared to a more general case where monotonic response is not assumed). Hence, the coverage requirement can be primarily satisfied by sampling the vertices of a *convex polytope* that encloses most of the volume of the input space. Then, any additional samples taken within the polytope contribute to better noise-tolerance in the model estimation and to a more accurate reflection of the non-linearity in the application response. Hence, sampling the vertices of a large polytope serves as the minimal requirement for coverage under the monotonic response assumption. Subsequently, we can parametrically trade-off the overheads of collecting further samples within the polytope against the additional benefits of noise reduction and fitting non-linearities within the polytope.

Unfortunately, estimating the volume of a convex polytope enclosing a given set of points in multi-dimensional space requires a computationally expensive algorithm, leading to potentially large overheads at runtime. Hence, we approximate the polytope with an axis-aligned bounding box (AABB), that simply records the minimum-to-maximum statistical spread of values taken by each input variable. The volume of the AABB is very efficient to calculate when coverage needs to be determined. The AABB would always have a larger volume than the precise convex polytope over the sampled vertices. Therefore, we would require the AABB to cover a larger fraction of the input volume, in order to get the precise (but unknown) polytope to exceed a smaller volume threshold with high probability.

Representativeness. The immediately preceding sequence of frames would be considered the most accurate representatives of the current application behavior. Additionally, we would expect that the longer the frame sequence, the greater the number of sample points available to more densely explore the input space. However, apart from the runtime overheads of storing a long history of sample data, the following factors indicate that a longer frame sequence can become highly detrimental. First, the application behavior may have changed abruptly (as opposed to a gradual shift) at some past frame. Inclusion of frames preceding that change contributes samples that do not reflect the current application behavior. Consequently, a larger number of new frames would have to be collected to dilute the effect of the prior non-representative frames, leading to a larger lead-time before model estimation would produce an accurate model. Second, a longer sequence does not necessarily achieve greater coverage in the sample data. If the controller had confined the control inputs to a narrow range over a frame sequence, but subsequent frames explored a larger portion of the input space, then inclusion of a large number of the confined frames would bias the sample data against the frames exhibiting greater exploration, leading to estimation of a poorer-quality model. In summary, the representativeness of the sample data is enhanced by *i)* detection of *behavior change points* so non-representative data can be discarded, and *ii)* adjusting the duration of the sample data retained so as to capture a representative range of the application's current behavior (in contrast with the samples achieving coverage over the input space).

To ensure estimation of high-quality models with high probability, it is vital for the sample data to have sufficient coverage over the input space and be representative of the current application behavior. We develop metrics that continually and efficiently characterize coverage, identify behavior change-points in the collected sample history data and estimate a duration of frames over which the application behavior becomes stable. The metrics indicate in a precise manner when the data is suitable for high-quality model estimation, when the history size (i.e., the number of preceding frames retained) needs to be increased or decreased, and how the past frames should be weighed against more recent frames during model estimation.

Question 7 *How is the training data generated?*

The previous question explored how the coverage and representativeness aspects of the training data determined the likelihood of estimating a model with high prediction accuracy (i.e., a "good" model). Every frame, the controller produces a control input for the application. On any frame, the controller generates the control input under one of the following three mutually exclusive conditions.

Case 1: No regulator defined.

Case 2: Regulator drives control inputs.

Case 3: Forced exploration of inputs, disregarding available regulator.

Next we describe how the coverage and the representativeness of the training data are impacted in each case. A regulator is not defined until model estimation establishes an active model and LQR is performed on the active model. Subsequently, the controller decision logic may detect that the active model exhibits significantly compromised prediction accuracy on the current application behavior. If so, the controller invalidates the active model even if no suitable substitute model is available to take its place. Model invalidation immediately invalidates the current regulator (because it was constructed from the model), again leaving no regulator defined.

Whenever the regulator is either undefined or forcibly not used (Case 1 or Case 3, respectively), we use an *input explorer algorithm* that at each frame identifies the input variables exhibiting insufficient coverage of their respective ranges. For each variable with insufficient coverage, a value is sampled uniformly at random over a sub-range where coverage was lacking. All the other variables that presently have sufficient coverage hold their values fixed to those from the prior frame.

The input explorer provides the following benefits.

- Within very few frame, provides a high probability that the convex polytope enclosing the sample points achieves non-zero volume (i.e., the sample points cover a non-zero range in every dimension of the input space).

- Within very few frames, provides a high probability that the axis-aligned bounding box (AABB) enclosing the polytope has volume exceeding a desired large fraction of the input space.

- The input dimensions currently exhibiting insufficient coverage increase either their coverage or their sampling density in the given frame.

Impact on coverage. Once a regulator is constructed and is driving application control inputs (Case 2), further exploration of the input space may get inhibited. For example, the regulator may limit a control variable to a narrow range or even a fixed value, if the regulator determines that this range/value delivers best application QoS. In general, the sample data may quickly lose coverage when the control inputs are driven by the regulator.

Therefore, we are faced with an *exploration versus exploitation dilemma* every frame. Should the controller use the current model to control QoS as best possible (Case 2)? Or should the controller explore the application behavior further, creating the opportunity for a new model with potentially far better QoS-control capabilities to be discovered, but preventing the current model's regulator from driving control inputs (Case 3)? The exploration-vs-exploitation dilemma is inherent in any runtime scheme that must choose between using current knowledge and discovering new knowledge. In our controller, exploration directly increases coverage in the training data, while exploitation generally diminishes coverage. The next question explores the criteria used in the controller for balancing exploration versus exploitation.

Impact on representativeness. The controller maintains a finite history of the most recent samples of the application behavior (i.e., the training data). Exploration on a given frame is a waste if the sample data from the given frame gets discarded from the finite training data by the time model estimation is next invoked. Additionally, even when a long history is maintained, the sample data from the given frame may no longer be representative by the time model estimation is performed (as discussed in the previous question).

Hence, one goal during exploration is to retain sufficient representative data and to use it for model estimation before its representativeness is lost. Another goal is to ensure that the forced exploration does not disrupt application QoS to an extent exceeding any subsequent benefits of the exploration to model estimation. The next question discusses the mechanisms employed by the controller to meet these goals and balance exploration versus exploitation.

Question 8 *How do we balance exploration versus exploitation?*

Forced exploration allows the training data to periodically gain sufficient coverage to allow model estimation. If some fixed percentage of application frames are devoted to forced exploration, we want the exploration to occur in patterns that maximize the likelihood of model estimation. Separately, we want to determine what percentage of application frames should be devoted to forced exploration. The following types of considerations factor in here.

1. Is the current model already producing very high application QoS, such that any disruption arising from the search for an even better model is highly counter-productive to the resulting application QoS?

2. If the current model is producing only mediocre QoS, is it likely that another model can significantly outperform the current model? If yes, we want to devote more frames to forced exploration. If not, we want to minimize disruption to salvage whatever QoS the current model is able to provide.

3. Can poor QoS produced by a model trigger a QoS death spiral, where aggressive exploration prevents models from functioning as best as they can, leading to continually more aggressive exploration and ever poorer QoS in subsequent models?

Hence, we break the exploration versus exploitation problem into two sub-problems:

- Determine what patterns of forced exploration would maximize coverage and representativeness benefits if the exploration were limited to a specified fraction of the application frames (the *exploration fraction*).

- Determine what exploration fraction would best balance the estimated benefit of exploration to future QoS against the disruption imposed on current QoS.

Section 6.4 provides the mathematical details. The following two questions cover the above sub-problems.

Question 9 *Sub-problem 1: What patterns of forced exploration maximize benefits when exploration is limited to a specified fraction of the application frames?*

The following mechanisms control the patterns of forced exploration.

- Clustered sequences
- Quantified coverage gap
- Probabilistic structure

Clustered sequences. When the controller forces exploration, it does so for a contiguous sequence of frames. Use of clustered exploration significantly raises the likelihood that the generated training data achieves sufficient coverage to allow model estimation and consists of the most representative sample data. Without exploration in clusters, frames with forced exploration would be scattered over the frame sequence. Scattered exploration increases the likelihood that at any given time the training data contains *some* exploration frames, but may significantly decrease the likelihood that the exploration frames contribute sufficient coverage to allow model estimation (consider the situation where the coverage is consistently almost sufficient, but rarely sufficient). In contrast, clustered exploration *decreases* the likelihood that the training data at any given time contains any exploration frames, but when forced exploration does occur it is more likely to continue for a duration sufficient for the training data to cross the coverage threshold needed for model estimation. Hence, for the same fraction of application frames devoted to forced exploration, clustered exploration is far more likely to exploit the generated samples for model estimation than scattered exploration. Additionally, the clustering maximizes the representativeness of the exploration samples — the coverage threshold is likely exceeded by the end of the clustered exploration sequence, enabling model estimation right at the end of the clustered sequence, before the bounded-length training data likely loses any exploration samples.

Quantified coverage gap. The controller has a metric quantifying the degree to which the current training data falls short of the coverage needed for model estimation. From this coverage-gap metric, the controller estimates the cluster length of the forced exploration that would likely achieve the coverage threshold. Hence, whenever the inputs produced by the regulator are by themselves meeting coverage, forced exploration is *skipped entirely.* More generally, the coverage-gap metric limits the forced exploration to a frequency and cluster length *just sufficient* for achieving the coverage threshold.

Probabilistic structure. Despite having a coverage-gap metric that can deterministically dictate the cluster length for the next forced exploration, we choose to use probabilistic mechanisms for determining whether exploration is triggered on a particular frame and what its cluster length is. Hence, the coverage-gap metric only shapes the parametric probability distributions from which the cluster length and exploration frequency are sampled. A probabilistically sampled exploration structure has certain critical advantages over a deterministic exploration structure. First, an application might have its own patterns of behavior that interact perversely with a deterministic exploration structure of just the right duration and frequency to stymie the working of the controller. For any given application, a probabilistic structure is less likely to repeatedly produce exploration patterns that stymie the controller operation. Second, a probabilistic structure allows very precise control of exploration properties over long sequences of frames. For example, the exploration frequency can be chosen to be arbitrarily close to zero, and over a long sequence of frames very simple probabilistic sampling techniques effectuate the desired frequency. In contrast, a deterministic structure must be produced by a priori fixed pattern generators, requiring very sophisticated schemes to effectuate the desired exploration parameters.

Question 10 *Sub-problem 2: What fraction of application frames devoted to exploration would best balance the likely benefits to future QoS against the disruption to the current QoS?*

The following steps determine a suitable *exploration fraction.*

- Estimation of the QoS potential of the active model

- Estimation of the best achievable QoS by any other model
- Determination of a new exploration fraction from the two estimates

QoS potential of the active model. The achieved QoS is determined both by the capabilities of the active model as well as the exploration fraction. We estimate the QoS potential of the active model by extrapolating from the frames that the regulator was allowed to drive inputs on. Strictly speaking, a scattering of forced exploration frames within a frame sequence impacts the QoS performance of the regulator-driven frames as well. This impact could be due to a non-zero input-order in the underlying application behavior, or when a QoS output variable is set up to average its value over a window of frames. Hence, adjacent forced exploration frames in the frame sequence may have an unpredictable impact on the QoS measurements on the regulator-driven frames. However, there are two mitigating factors that allow us to sufficiently approximate the model's QoS performance as the QoS measured over only the regulator-driven frames. First, the exploration happens in clusters, thereby affecting the QoS of only the regulator-driven frames occurring immediately after the exploration cluster (in contrast to a scattering of exploration frames having a potentially more widespread impact). Second, when the exploration fraction is small, the error in the model's QoS measurement drops. When the exploration fraction is larger, the measurement error can be substantially larger. However, a large exploration fraction occurs only when the model has previously been estimated to have low QoS, allowing large errors only when the model QoS is already estimated to be poor. When the introduced error consistently makes the model QoS appear far better than actual, the exploration fraction would keep dropping, improving the accuracy of subsequent QoS estimation: a built-in feedback mechanism. This leaves us with the situation when the error makes the model QoS appear far worse than it actually is. Such a situation can trigger a QoS "death spiral" for a series of frames until a new active model is estimated. The dropping QoS triggers a large exploration fraction, allowing a new active model to be estimated expeditiously, thereby ending the death spiral quickly: another feedback-based safe-guard even if it sometimes causes the current model to be needlessly replaced.

Best achievable QoS. Having an estimate of how good the QoS can be for the given application allows the controller to determine whether *i)* the active model is far under-performing and aggressive exploration must be attempted to estimate a better model, or *ii)* if the active model is delivering close to the best QoS performance possible on this application and further exploration should be minimized to avoid disrupting whatever QoS the current model is able to provide. A large under-estimation of the best achievable QoS would cause a bad model applied early in the execution of the application to persist. A large over-estimation of the best achievable QoS would expend needless application frames on exploration, preventing a reasonably good active model from delivering on its QoS potential. The best achievable QoS is dependent not only on the application but also on the application data-set. Hence, we cannot rely on an a priori estimate of the best achievable QoS. Instead, the controller initially assumes the best possible QoS as being achievable. Then, as new models continue to become active, the controller continually updates a best-achievable-QoS metric as the *average QoS of the top quantile* of the active models encountered. This scheme has the following advantages: *i)* poor-performing models estimated during noisy or anomalous phases of the application only minimally impact the metric, *ii)* the controller initially allows higher exploration, increasing the likelihood that high-performing models would get discovered, *iii)* if the application behavior characteristics gradually change such that a higher performing model is no longer possible, the accumulation of poorer performing active models will gradually shift the metric down, gradually clamping down on the exploration. The disadvantage of the scheme is that if the application characteristics change suddenly to allow high-performing models after a long sequence of low-performing active models, the scheme would increase the exploration fraction only after multiple good active models have already been encountered. However, note that the scheme's reliance on the top quantile of model QoS clamps down the exploration fraction slowly after encountering a series of poor active models, but can ramp up much faster on encountering relatively fewer good active models.

Determination of the exploration fraction. Two factors are multiplied to determine the next exploration fraction. First, how much exploration can the current active model *tolerate* without significantly impacting the model's QoS performance? Second, what is the *achievability gap* between the estimated best achievable QoS and the estimated QoS of the active model? The tolerance factor is computed as a function of the active model's QoS. The achievability gap factor is computed as the relative difference between the best achievable and the active model's QoS estimates.

Finally, note that we use two different types of metrics to compare the quality of models:

- *Prediction accuracy*: the discrepancy between observed application outputs and the outputs predicted by a model over the recent history of control inputs. This metric is used when comparing the active model against a substitute model.
- *Application QoS*: the application QoS delivered by the regulator constructed from the active model. This metric is used to compare the current active model against past active models to determine the achievability gap.

The prediction accuracy is necessary to make comparisons against substitute models that have not, as yet, been used to drive application control inputs and hence their actual QoS performance is unknown.

6.2 Model Estimation with LLSE

Model estimation relies on the following components.

Sample History $\mathcal{H}$. Application behavior observed over a sequence of frames is used as the training data for LLSE. At any frame t, the control inputs applied and QoS outputs observed over the *most recent* sequence of frames is retained as the history $\mathcal{H}$:

$$\mathcal{H} = \big[\ (\vec{x}_{t-k}, \vec{y}_{t-k}) \mid k \in [1, |\mathcal{H}|]\ \big]. \tag{40}$$

The length of history retained, $|\mathcal{H}|$, is determined adaptively at runtime. The ideal length is

1. long enough to retain enough samples to cover the input space (coverage requirement),
2. long enough samples to capture a stable representation of the application's current behavior (stability aspect of the representativeness requirement),
3. yet short enough that the retained samples only represent the current application behavior, not prior behavior (behavior change-points aspect of representativeness).

The coverage and representativeness requirements are often in conflict in determining $|\mathcal{H}|$. Therefore, for generality, a "forget-rate" parameter γ progressively weighs down older samples: the $k^{\texttt{th}}$ past sample $(\vec{x}_{t-k}, \vec{y}_{t-k})$ has weight γ^{k-1} $(0 < \gamma < 1)$ when used by LLSE. γ is adaptively adjusted by the controller. With the aid of an appropriately chosen γ, $\mathcal{H}$ can be made long enough to achieve coverage, relying on older, less representative samples if necessary. At the same time, the low weightage of the older samples (if retained) in the estimation of model $\mathcal{M}$ helps maintain representativeness. The controller maintains a recommended length setting for $\mathcal{H}$, represented by L_γ (i.e., the controller allows $\mathcal{H}$ to retain up to L_γ samples). We constrain γ and L_γ to always obey the following relationship.

$$\gamma^{L_\gamma} = 0.10 \tag{41}$$

This constraint represents a heuristic that on frame t, only samples $(\vec{x}_{t-k}, \vec{y}_{t-k})$ with weight $\gamma^{k-1} > 0.10$ are retained for LLSE. Any older samples (with $k \geq L_\gamma$) will have weight ≤ 0.10, and are considered to have insufficient impact on LLSE to be worth retaining. The controller manipulates L_γ while γ is updated as a dependent variable. In summary, all the L_γ samples in $\mathcal{H}$ contribute towards coverage, while representativeness is predominantly determined by the newer samples with larger γ^{k-1} weights.

In addition to the training data needing to have an appropriate number of samples, the samples themselves have to be generated in a manner that ensures coverage over the input space. Section 6.2.3 describes metrics that quantify coverage of the training data and an algorithm to generate samples that cover the input space when a new model needs to be estimated but the coverage is not sufficient to allow model estimation.

Active Model $\mathcal{M}$, Regulator $\mathcal{C}$ and Substitute Model $\mathcal{M}'$. The controller applies LQR on an estimated model $\mathcal{M}$ to construct a regulator $\mathcal{C}$. The regulator drives application inputs $\vec{x}$. $\mathcal{M}$ is referred to as the *active model.* The controller also attempts to periodically estimate a *substitute model* $\mathcal{M}'$. The controller attempts to compare the prediction accuracy of $\mathcal{M}$ against $\mathcal{M}'$ using subsequent samples collected in $\mathcal{H}$. Whenever $\mathcal{M}'$ is found unambiguously more accurate than $\mathcal{M}$, the controller replaces $\mathcal{M} \leftarrow \mathcal{M}'$.

$\mathcal{M}$ and $\mathcal{M}'$ are estimated using LLSE on the samples in $\mathcal{H}$ whenever the samples have sufficient coverage and have representativeness to the extent possible. LLSE needs a regularization parameter λ to be chosen to allow robust models to be estimated. Section 6.2.6 covers an adaptive runtime scheme to determine λ suitable for the current application behavior.

Continuous Forced Exploration and Probabilistic Forced Exploration. We need $\mathcal{H}$ to have coverage and representativeness *only* in the following situations.

1. When no active model is defined: $\mathcal{M} = \phi$. This is a critical situation as there is also no regulator $\mathcal{C}$ defined to drive application inputs.

2. Need to estimate a new $\mathcal{M}'$.

3. Need to compare the prediction accuracy of $\mathcal{M}'$ over $\mathcal{M}$.

In the first situation, the controller will extend $\mathcal{H}$ and shape the application inputs $\vec{x}$ to achieve coverage in the minimum number of frames possible. We call this process *continuous forced exploration* (CFE). Representativeness in $\mathcal{H}$ is desired but not required due to the critical need to estimate $\mathcal{M}$ as quickly as possible.

The controller only makes a "best effort" to satisfy the needs identified in the second and third situation. This is because $\mathcal{M}$, and hence $\mathcal{C}$, are already defined and the controller is able to control application QoS. In these two situations, the controller uses a process we call *probabilistic forced exploration* (PFE). PFE *i)* adjusts $|\mathcal{H}|$ based on the coverage and representativeness statistics observed for $\mathcal{H}$, and *ii)* with a calculated probability overrides the inputs $\vec{x}$ produced by $\mathcal{C}$ on certain frames and instead samples $\vec{x}$ from a probability distribution that will boost coverage and representativeness in $\mathcal{H}$. The objective of PFE is to periodically allow $\mathcal{M}'$ estimation and comparison against $\mathcal{M}$ without significantly impacting the QoS performance delivered by $\mathcal{M}$. The adverse impact on QoS arises due to PFE overriding the application inputs from $\mathcal{C}$. Section 6.4 covers PFE. Each occurrence of CFE and PFE spans one or more consecutive frames — referred to as a *CFE cluster* or *PFE cluster* of frames, respectively.

CFE is expected to be a rare occurrence (say, during the initial frames or occasionally when the application behavior changes dramatically within a very short span of frames). The vast majority of the application frames are expected to be either non-exploration (i.e., $\mathcal{C}$ drives the application input $\vec{x}$) or part of a PFE cluster.

Adapting $|\mathcal{H}|$. On any given frame the following metrics determine $|\mathcal{H}|$.

- L_s: the controller's estimate for $|\mathcal{H}|$ that will achieve stability of behavior in the samples in $\mathcal{H}$. The stability aspect of representativeness attempts to collect in $\mathcal{H}$ *all the distinct behaviors* that the application is *currently exhibiting.* Behavior refers to the application's $(\vec{x}, \vec{y})$ input-output response. The controller may not have an estimate for L_s on every frame.

- L_c: the controller's estimate for $|\mathcal{H}|$ that would achieve coverage with high probability *immediately after* a PFE cluster. The intent is to retain sufficient older samples of $\vec{x}$ that in conjunction with the newest samples of $\vec{x}$ from a PFE cluster confer coverage to $\mathcal{H}$.

- L_γ: the desired length for $\mathcal{H}$. $|\mathcal{H}|$ can grow until $|\mathcal{H}| = L_\gamma$. Then, the oldest sample is dropped from $\mathcal{H}$ whenever a new sample is added. However, $|\mathcal{H}|$ is allowed to exceed L_γ during a CFE cluster. L_γ is adjusted as a compromise between the L_s (if defined) and L_c values.

Note that a chosen L_c value is not intended to be sufficient for achieving coverage on every frame t with high probability. Instead, L_c is intended to be sufficient only for achieving coverage soon after a PFE cluster. Immediately after a PFE cluster, $\mathcal{H}$ will also contain samples produced by $\mathcal{C}$ and potentially samples from prior PFEs. L_c is a length recommendation that implicitly encompasses the impact on coverage of the current and prior PFEs and of the inputs produced recently by $\mathcal{C}$. However, the process of estimating L_c is feedback-driven on the coverage statistics of $\mathcal{H}$, hence the process does not need to explicitly model the effect of the PFEs and $\mathcal{C}$ on coverage. *By not requiring L_c to confer coverage on every frame with high probability, we dramatically reduce the adverse impact of PFE on application QoS — far fewer frames have to participate in PFE if coverage is required only occasionally.* The controller dynamically tunes the frequency with which frames achieve coverage by adjusting the probability parameters that PFE relies on.

In contrast, the L_s estimate is chosen to confer stability with high probability on any frame while $|\mathcal{H}| = L_s$, regardless of the specific locations of the PFE clusters within $\mathcal{H}$. The process that estimates L_s also detects behavior change points in $\mathcal{H}$ (the second aspect of representativeness after stability). If frame t_{bcp} in $\mathcal{H}$ is detected as a behavior change point, all samples older than and including t_{bcp} are deleted from $\mathcal{H}$, but L_s and L_γ are not directly impacted. The goal is to prevent model estimation or comparison from using frames that represent outdated application behavior. Essentially, L_s is estimated using the long-term statistics of the application behavior, while behavior change points are detected when the short-term statistics of the current samples in $\mathcal{H}$ exhibit an anomaly compared to the long-term statistics.

Section 6.2.1, Section 6.2.2, Section 6.2.3 and Section 6.2.4 provide details on estimation of L_s, t_{bcp}, L_c and L_γ, respectively.

6.2.1 Quantifying Stability in $\mathcal{H}$

Prediction Error. The prediction accuracy of the active model $\mathcal{M}$ is used to establish changes in application behavior. The *prediction error* e_t (a proxy for prediction accuracy) at frame t is defined as $e_t \triangleq \frac{1}{dim(\vec{y})}||\vec{\hat{y}}_t - \vec{y}_t||^2_{(\vec{s},\vec{\delta})}$, where $\vec{\hat{y}}_t$ is the application output predicted by $\mathcal{M}$ for frame t using the applied inputs and $\vec{y}_t$ is the actual output observed from the application. The subscript $(\vec{s},\vec{\delta})$ indicates that our use of the L2-norm incorporates the user-specified *importance* s_i and *tolerance* δ_i values of each output dimension y_i (Section 5.4), as shown below.

$$||\vec{\hat{y}}_t - \vec{y}_t||^2_{(\vec{s},\vec{\delta})} \triangleq \sum_i s_i \frac{|\hat{y}_{i|t} - y_{i|t}|^2}{\delta_i^2} \tag{42}$$

As with the definition of τ_t (Eq 2), the error in each output is normalized by the corresponding δ_i to allow the errors from the different outputs to be meaningfully combined, and the relative importances s_i allow the controller to react more readily to errors in outputs that are more important to the user.

We use the notation $\vec{\hat{y}}_t = \mathcal{M}(\mathcal{H}, t)$, where it is assumed that $\mathcal{H}$ contains sufficient recent samples (based on the model-order of $\mathcal{M}$) to apply $\mathcal{M}$ at frame t. Hence,

$$e_t \triangleq \frac{1}{dim(\vec{y})}||\mathcal{M}(\mathcal{H}, t) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})}. \tag{43}$$

There might be significant variation in e_t over a sequence of frames, yet that does not necessarily imply that the application behavior has changed. Often, a sequence of somewhat differing behaviors that appear repeatedly and in quick succession may be best recognized by the controller as a single large behavior, for which a single model $\mathcal{M}$ is estimated. We *assume* that if the statistical distribution of e_t over frame-sequences of a suitable length L remains similar to other frame-sequences of length L then the application behavior has not changed. Further, the application behavior is then considered *stable* over lengths L. The controller attempts to estimate the shortest length L over which the behavior is found stable, and if found, represents it as L_s.

Let $\mathcal{T}_{xy}$ represent the history of input-output samples, similar to $\mathcal{H}$, but whose length is manipulated independent of $\mathcal{H}$. Let $\mathcal{T}_e$ represent the $\{e_t\}$ sequence computed over the samples in $\mathcal{T}_{xy}$ using the active model $\mathcal{M}$. Therefore, $|\mathcal{T}_e| = |\mathcal{T}_{xy}| - \max\{x_{order}, y_{order}\}$. $\mathcal{T}_e$ needs to maintain a sufficient number of samples to evaluate stability of any length L that could potentially serve as the stability length L_s.

If $\mathcal{M} = \phi$ on any frame t, $\mathcal{T}_e$ becomes undefined. $\mathcal{T}_e$ must be recalculated whenever model substitution $\mathcal{M} \leftarrow \mathcal{M}'$ occurs or a new model $\mathcal{M}$ is directly estimated.

Kolmogorov-Smirnov Measure of Statistical Dissimilarity. Given a length L as a *candidate length for stability*, the controller uses the Kolmogorov-Smirnov (K-S) distance D $(0 \leq D \leq 1)$ to establish the degree of dissimilarity between two segments of length L extracted from $\mathcal{T}_e$. $D = 0$ implies that the value distributions of the two segments are identical, whereas $D = 1$ implies the maximal dissimilarity between the two value distributions. We use a histogram as a practical means to approximately capture the value distribution of each segment. The histograms are binned over the min-to-max range of e_t seen since the last behavior change point detected. Keeping the range of the histograms narrow in this manner maximizes the sensitivity of K-S in detecting statistical dissimilarity.

We compute D between every two adjacent segments of length L and use a *weighted average* $\bar{D}$ to establish if the length L is statistically *stable*, *unstable* or *highly unstable* in an application-independent manner. We heuristically define classifier $s(L)$, as follows, to query the statistics collected for candidate length L. Note that evaluation of $s(L)$ requires at least two segments of length L to be present in $\mathcal{T}_e$, i.e., need $|\mathcal{T}_e| \geq 2L$.

$$s(L) \triangleq \begin{cases} \text{unknown,} & \text{if } \mathcal{M} = \phi \text{ or } |\mathcal{T}_e| < 2L \\ \text{stable,} & \text{if } \bar{D}(L) \leq 0.10 \\ \text{unstable,} & \text{if } \bar{D}(L) \leq 0.50 \\ \text{highly unstable} & \text{if } 0.50 < \bar{D}(L) \ (\leq 1) \end{cases} \tag{44}$$

Ideally, we would determine the thresholds (0.10 and 0.50) in an adaptive manner suitable for each application. However, we leave as future work the determination of whether application-specific adaptation is necessary, and if so, how to perform it.

Specifically, let $D(L, t_0)$ represent the K-S distance between the following two adjacent segments of $\mathcal{T}_e$: $S([t_0+1, t_0+L]) = \{e_{t'} \in \mathcal{T}_e \mid t_0+1 \leq t' \leq t_0+L\}$ and a similarly defined $S([t_0-L+1, t_0])$. If $\mathcal{M}$ was estimated on frame $t_{\mathcal{M}}$ and the current frame is t, we have $n_L = \left\lfloor \dfrac{t - t_{\mathcal{M}} + 1}{L} \right\rfloor$ complete segments and $n_L - 1$ values for D. In computing $\bar{D}(L)$ we heuristically choose to weigh the oldest available D with 0.10 and the most recent one with 1.0, with the intervening D's given weights in a geometric progression between 0.10 and 1.0.

We retain the most recent D's (all computed using the current $\mathcal{M}$) based on two criteria: *i)* allow at least 20 samples to be retained to confer statistical stability to the mean $\bar{D}$, and *ii)* beyond that discard the oldest samples computed for timesteps older than the current oldest sample in $\mathcal{H}$.

We get the following definition for $\bar{D}$ when $\mathcal{M} \neq \phi$ and $t - t_{\mathcal{M}} + 1 \geq 2L$.

$$\bar{D}(L) \triangleq \frac{\sum_{i=1}^{n_L-1} w^{n_L-i-1}\, D(L,\, t_{\mathcal{M}} + iL - 1)}{\sum_{i=1}^{n_L-1} w^{n_L-i-1}}, \tag{45}$$

where w is s.t. $w^{n_L-2} = 0.10$.

Formally,

$$L_s \triangleq \begin{cases} \text{Shortest } L \text{ s.t. } s(L) = \text{stable} \\ \text{undefined, if no stable } L \text{ is known} \end{cases}. \tag{46}$$

Sketch of Algorithm for Estimating L_s. Recall that W is the perception-window length provided by the programmer for the application (Section 5.4). The variations in application behavior within durations of W frames are considered "noise" as these will not be perceptible to the interactive user. Hence, $L = W$ is the minimum segment length worth considering as a candidate for stability. Further, as an efficiency optimization we restrict all candidates L to be multiples of W.

An efficient algorithm that computes L_s is sketched below.

- Maintain *candidate set* L_s^{set}. On every frame $s(L)$ is updated $\forall L \in L_s^{\text{set}}$. Let $L_s^{\max}$ represent the max L in L_s^{set}. Initially, $L_s^{\text{set}} \leftarrow \{W\}$.
- Allow $|\mathcal{T}_e|$ to grow till $2L_s^{\max}$. After that the oldest sample is dropped when a new one is added to $\mathcal{T}_e$, maintaining $|\mathcal{T}_e| = 2L_s^{\max}$ (until $L_s^{\max}$ changes).
- Binary search to add/remove L's from L_s^{set} on the current frame.
  - If $\nexists L \in L_s^{\text{set}}$ s.t. $s(L) = \text{stable}$ and $|\mathcal{T}_e| = 2L_s^{\max}$, add $2L_s^{\max}$ to L_s^{set}. (Note, $|\mathcal{T}_e| = 2L_s^{\max} \implies s(L_s^{\max}) \neq \text{unknown}$.)
  - If $\exists$ contiguous $L_1, L_2 \in L_s^{\text{set}}$ s.t. $s(L_1) \neq s(L_2) \ \wedge\ s(L_1) \neq \text{unknown} \ \wedge\ s(L_2) \neq \text{unknown}$, add $L = \frac{L_1+L_2}{2}$ rounded to the closest multiple of W to L_s^{set}. Contiguous means that $L_1 < L_2 \ \wedge\ \nexists L' \in L_s^{\text{set}}$ s.t. $L_1 < L' < L_2$. (This step doesn't necessarily add L as L may round to L_1 or L_2).
  - If $\exists$ contiguous $L_1, L_2, L_3 \in L_s^{\text{set}}$ s.t. $s(L_1) = s(L_2) = s(L_3) \neq \text{unknown}$, remove L_2 from L_s^{set}.

The controller can add an external L to L_s^{set} for evaluation, such as when $L_c > L_s$ and it would be desirable not to set $L_\gamma \leftarrow L_c$ if $s(L_c) = \text{highly unstable}$ (a trade-off when the controller must choose between providing full coverage and maintaining representativeness, as the two requirements are often found to be in conflict).

6.2.2 Detection of Behavior Change Points in $\mathcal{H}$

If $L_s \neq \text{undefined}$ (i.e., $s(L_s) = \text{stable}$), can we find t_{bcp} s.t. $D(L_s, t_{\text{bcp}}) > 0.50$? That is, does there exist a segment $[t_{\text{bcp}} - L_s + 1,\ t_{\text{bcp}}]$ that exhibits highly dissimilar behavior against an adjacent segment, while segments of length L_s have *typically* been found to have behavior similar to their adjacent segments? If so, t_{bcp} is a behavior change point and the controller drops samples older than t_{bcp} from $\mathcal{H}$ and also attempts to estimate a new $\mathcal{M}'$ if $t_{\mathcal{M}'} - L_\gamma \leq t_{\text{bcp}}$, i.e., the estimation of $\mathcal{M}'$ may have used some sample data from before t_{bcp} and therefore $\mathcal{M}'$ should no longer be allowed to substitute $\mathcal{M}$. Further, substitute $\mathcal{M} \leftarrow \mathcal{M}'$ if $t_{\mathcal{M}} - L_\gamma \leq t_{\text{bcp}}$ (i.e., the estimation of $\mathcal{M}$ may have relied on outdated samples), and if $\mathcal{M}'$ was estimated only on samples newer than t_{bcp}. Otherwise, the controller sets $\mathcal{M} \leftarrow \phi$ and $\mathcal{M}' \leftarrow \phi$, and CFE is performed.

6.2.3 Quantifying Coverage in $\mathcal{H}$

We would like the training data in $\mathcal{H}$ to span a large fraction of the input space over $\vec{x}$. Under our monotonic-response assumption about the application behavior (Section 5.4), it suffices to sparsely sample the input space as the application response behavior can be approximated by interpolation (implicitly done by LLSE). Therefore, the volume of the polytope enclosing the input samples $\{\vec{x}_t\}$ can be a useful estimate of the coverage of input space by $\mathcal{H}$. However, computing the volume of a polytope has high computational complexity, unsuitable for an online scheme. Therefore, we use the axis-aligned bounding-box (AABB) as an approximation of the polytope.

Coverage Tests. We say that the training data *has coverage* when it passes both the following tests for each input dimension.

- **AABB statistical span test.** Does the *standard deviation* of the x_j input variable exceed a *minimally required fraction* f (say, 50%) of the span of the j^{th} input dimension?

- **Significant swing test.** Does the input variable x_j exhibit at least a *single large swing* exceeding a fraction g of the span of the j^{th} input dimension?

The AABB statistical span test ensures variation over the input space over the bulk of the inputs, therefore preventing any narrow range of input values from dominating during LLSE, but does not ensure that any of the variations are large. The significant swing test ensures that the sample data has some (or at least one) large variation, but does not ensure that the bulk of the sample data exhibits variations. Hence, the two tests are complementary in establishing coverage.

Coverage Metric κ. We define the *coverage metric* κ as the number of input dimensions that satisfy both the above tests over the input samples in $\mathcal{H}$, normalized by the total number of input dimensions.

$$\kappa \triangleq \frac{1}{dim(\vec{x})} \sum_{j=1}^{dim(\vec{x})} I\left\{\sigma_j \geq f\, N_j \;\wedge\; \chi_j \geq g\, N_j\right\}, \tag{47}$$

where, each standard deviation σ_j and implicitly the mean μ_j are computed using samples of input dimension x_j weighed by the forget-rate γ. That is, at time t sample $x_{j|t-k}$ is weighted by γ^{k-1}, just as LLSE would do if the current $\mathcal{H}$ was used for model estimation. χ_j represents the *maximum swing* in x_j seen over the samples in $\mathcal{H}$, defined as follows.

$$\chi_j \triangleq \max_{1 \leq k < l \leq |H|} \left| \gamma^{k-1} x_{j|t-k} - \gamma^{l-1} x_{j|t-l} \right| \tag{48}$$

When a new sample is added to $\mathcal{H}$ or the oldest sample dropped from $\mathcal{H}$, the metrics σ_j, μ_j and χ_j can be updated incrementally (i.e., without traversing $\mathcal{H}$). Appendix A.1 provides the incremental update formulae and shows how suitable f and g are determined. In practice, $f = 0.5$ can be assumed, unless $\gamma < 0.4$ (which is extremely rare). We derive $g = \dfrac{1+\gamma}{2}$.

Estimation of History Length L_c that Achieves Coverage. $\mathcal{H}$ achieves coverage both due to the input samples $\vec{x}$ generated by regulator $\mathcal{C}$ and the samples generated by PFE (CFE is intended to be very occasional, so we don't include it in our reasoning here). The frequency of occurrence of PFE and the cluster length of each PFE are both intentionally probabilistic properties. Hence, the coverage contributed by each successive PFE can differ. In one situation, $\mathcal{H}$ may still contain samples from a prior PFE cluster when the current PFE cluster completes. In another situation, the gap between the current PFE and the prior (i.e., an intervening sequence of non-PFE frames) may be large, causing $\mathcal{H}$ to have already dropped samples from the prior PFE cluster. In the first situation, the latest PFE cluster may confer coverage to $\mathcal{H}$ in conjunction with a prior PFE cluster. In the latter situation, the latest PFE cluster may fail to confer coverage. Hence, the probabilistic nature of PFE makes the L_c value that confers coverage itself a probabilistic entity. Therefore, we attempt to choose an L_c value that will confer coverage with *sufficiently high likelihood*, implicitly conditioned on the current probabilistic characteristics of PFE and the region of input space exercised by $\mathcal{C}$. As a heuristic, the controller estimates an *L_c-recommendation value* at the end of each PFE cluster, maintains a collection of the L_c-recommendations, and determines its L_c from the median statistics of the collection.

Algorithm Sketch for Estimating L_c.

- Maintain a *sorted sequence* of coverage recommendations L_c^{seq}. Each entry is a tuple consisting of a coverage-length sample and the frame it was sampled — $(L,\, t_k)$. The entries are kept sorted by coverage-length. Keep $|L_c^{\text{seq}}| \leq 10$, as that many samples are likely sufficient for filtering out noise, and for sampling a sufficient spread of the PFE and the application's current behavior.

- Initially, $L_c^{\text{seq}} \leftarrow \{(W,\, t)\}$.

- At end of a PFE cluster, sample the coverage κ of $\mathcal{H}$.
  - If $\kappa < 1$, i.e., PFE failed to achieve coverage, set $L \leftarrow |\mathcal{H}| + (1 - \kappa) \times 2\, dim(\vec{x})$.
  - If $\kappa = 1$, i.e., PFE achieved coverage, set $L \leftarrow \max\{|\mathcal{H}| - 2,\, W\}$.
  - Append $(L,\, t)$ to L_c^{seq}. t is the current timestep.

- At any frame t, $L_c \leftarrow$ average of the 50% to 75% median values in L_c^{seq}.

  (This choice creates a high likelihood that L_c will prove sufficient for coverage, while reducing the likelihood of an unnecessarily long L_c, and it does so without explicitly estimating the likelihoods.)

- On detection of t_{bcp}, drop samples from L_c^{seq} that were added before t_{bcp}. If this empties L_c^{seq}, add the current L_c as the best initial guess.

The algorithm uses *dynamic stability* (essentially a form of non-converging feedback control) to arrive at the "correct" L_c. While L_c may converge due to the averaging of the median statistics, the samples L do not. Instead, the L sample values will be increased until the PFE clusters start to achieve coverage and then reduced-increased continually around the values that frequently confer coverage at the end of a PFE cluster. Note that when coverage is proving sufficient over a long sequence of frames, no new entries get added to L_c^{seq}. This can keep an excessively large L_c (if that happens to be the case) from being corrected until coverage drops, PFE occurs and a smaller L_c-recommendation is added to L_c^{seq}. Hence, an excessive value of L_c may persist. In contrast, and crucially, note that when L_c is too small to achieve coverage, samples of increasing value are repeatedly added at the frequency the controller chooses to start PFEs (frequency determined by the metric q — Section 6.4), bringing about a more rapid correction in L_c at a rate of the controller's choosing.

6.2.4 Reconciling Coverage and Representativeness into Recommended History Length L_γ

We have two goals when updating L_γ.

- Reconcile the typically dissimilar recommendations L_s and L_c into a single value in a "safe" manner.
- Create robustness against frequent variations in L_s or L_c.

Frequent changes to L_γ make some of the controller's incremental-update algorithms less cost-effective. They are incremental only while γ is held fixed, requiring expensive re-computations over all the samples in $\mathcal{H}$ otherwise. For this reason, we compute a reconciled value L' from L_s and L_c every frame, update a filtered value L'' from L', and apply $L_\gamma \leftarrow L''$ only when L'' differs non-trivially from the current L_γ. Every frame the controller *i)* computes L', *ii)* then updates $L'' \leftarrow \dfrac{L' + L''}{2}$, and *iii)* then selectively updates $L_\gamma \leftarrow L''$ if $|L_\gamma - L''| \geq W$.

L' computation when L_s is defined. When $L_s \geq L_c$, it is safe to pick $L' \leftarrow L_s$ as any history length longer than L_c should also achieve coverage with a similar or greater probability. When $L_s < L_c$, we have the following situations.

- L_c is estimated as stable. We can safely set $L' \leftarrow L_c$ as stability and coverage are not found to be in conflict.

- L_c has unknown stability. We add L_c rounded to the closest multiple of W to L_s^{set}. If the prior frame's $L'' < L_s$, we conservatively set $L' \leftarrow L_s$. Otherwise, use $L' \leftarrow L''$, which provides some robustness in case L_c frequently takes values of unknown stability — remove L_c as a factor until the stability of its values becomes known.

- L_c is estimated as unstable. We heuristically "split the difference" between achieving coverage with high likelihood against the possibility of retaining non-representative sample data: $L' \leftarrow \dfrac{L_s + L_c}{2}$.

- L_c is estimated as highly unstable. We ignore L_c, thereby compromising the controller's ability to estimate and evaluate $\mathcal{M}'$ with a desired frequency, but *providing safety* against estimating or evaluating $\mathcal{M}'$ on highly non-representative sample data. $L' \leftarrow L_s$.

In the cases described above, we estimate the stability of L_c with the following steps.

1. For $L =$ closest multiple of W of L_c, use $s(L)$ if $L \in L_s^{\text{set}}$.

2. Otherwise, determine if $\exists$ contiguous $L_1, L_2 \in L_s^{\text{set}}$ s.t. $L_1 < L_c < L_2$, s.t. $s(L_1) = s(L_2)$. If so, use $s(L_1)$.

3. Otherwise, stability of L_c is presently unknown.

L' computation when L_s is undefined. When L_s is undefined on the current frame (i.e., $\forall L \in L_s^{\text{set}}, s(L) \neq \text{stable}$), L_c cannot be known to be stable (by definition of L_s).

L_s will be undefined on a frame for one of the following underlying causes.

Cause 1. The application behavior is changing rapidly enough that no L_s can be found. The best course of action is to ignore stability and just aim for coverage: $L' \leftarrow L_c$.

Cause 2. The length of sample data collected for evaluating stability, $|\mathcal{T}_e|$, is not sufficient for finding L_s. This suggests that $L_s > |\mathcal{T}_e|/2$ is a possibility.

Cause 3. There is significant noise in the sample data presently in $\mathcal{T}_e$ and evaluating over fresh data would reveal a stable L_s, perhaps even $L_s < |\mathcal{T}_e|/2$ for the present $|\mathcal{T}_e|$.

Cause 4. $M = \phi \implies \mathcal{T}_e = \phi$. CFE will extend $\mathcal{H}$ until coverage is achieved and $\mathcal{M}$ can be estimated. During each frame of CFE, set $L_\gamma \leftarrow \max(L_\gamma, |\mathcal{H}|)$ and $L'' \leftarrow L_\gamma$.

Cause 4 is detectable by the controller (as $\mathcal{M} = \phi$). However, the controller cannot definitively distinguish between the occurrences of the first three causes. First, on frames that have $\mathcal{M} \neq \phi$ and L_s is undefined, the controller uses $L' \leftarrow L_c$ as the default strategy, which would cover Cause 1 if that was indeed the underlying cause for L_s being undefined. Secondly, recall that the L_s-estimation algorithm (Section 6.2.1) adds $2L_s^{\max}$ to L_s^{set} when no stable L is known and $|\mathcal{T}_e| = 2L_s^{\max}$. In this way the algorithm searches for ever larger L, covering the possibility of Cause 2. Lastly, the controller doesn't need to take any special action in response to the possible occurrence of Cause 3, as continued application execution will naturally explore that possibility when $\mathcal{T}_e$ gets replenished with fresh data.

In summary, the controller *simultaneously responds* to the possible occurrence of Cause 1, Cause 2 and Cause 3, without attempting or needing to discover which of the three has actually occurred.

6.2.5 Invalidation of Active Model on Significant Deviations in Application Behavior

When the application behavior changes suddenly and to a very large degree (manifested as large increases in the prediction error), the controller invalidates the active model $\mathcal{M}$, forcing the immediate start of CFE to estimate a new active model in the shortest number of frames. Without this invalidation mechanism the controller would have had to rely on PFE to first achieve required coverage alongside the normal execution of the regulator $\mathcal{C}$, wait for a replacement model $\mathcal{M}'$ distinct from $\mathcal{M}$ to be estimated and then in due course for the advantage of $\mathcal{M}'$ to be established (potentially requiring an additional PFE cluster to achieve coverage) before $\mathcal{M}$ is finally replaced.

For resilience against frame-to-frame noise, we use the cumulative prediction error over the entire history of $\mathcal{M}$ as the metric for determining if $\mathcal{M}$ should be rejected. The *model tracking error* (MTE) metric at frame t is defined as follows, relying on the forget-rate γ to de-emphasize older prediction errors. Model invalidation is triggered when the MTE significantly exceeds (i.e, $> 10\times$) the model-fit-error of $\mathcal{M}$ over its training data.

$$\texttt{MTE}(t) \;\triangleq\; \frac{1-\gamma}{dim(\vec{y})} \sum_{k=0}^{\infty} \gamma^k \, ||\mathcal{M}(\mathcal{H}^{\infty}, t-k) - \vec{y}_{t-k}||^2_{(\vec{s},\vec{\delta})} \tag{49}$$

$\mathcal{H}^{\infty}$ is the unbounded history of samples retained since application start up, containing samples till at least frame t. The $1-\gamma$ factor normalizes the magnitude of the metric so its value can be meaningfully compared to the model prediction error of any single frame (intuition from geometric series sum: $\sum_{k=0}^{\infty} \gamma^k a = \frac{a}{1-\gamma}$, if a was the "typical" magnitude of the prediction error for any single frame). This normalization also allows the values $\texttt{MTE}(t)$ and $\texttt{MTE}(t')$ to be meaningfully compared for any two frames t and t'.

The MTE metric can be calculated recursively as follows

$$\texttt{MTE}(t) \;\leftarrow\; \frac{1-\gamma}{dim(\vec{y})} ||\mathcal{M}(\mathcal{H}, t) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})} + \gamma \, \texttt{MTE}(t-1). \tag{50}$$

Note that in the recursive calculation we can now use the finite $\mathcal{H}$. This is because $\texttt{MTE}(t–1)$ is pre-computed (pre-computation used the finite $\mathcal{H}$ of the previous frame). Therefore, we use the recursive form to compute MTE in practice.

There are two properties of the MTE metric that will help detect a change in application behavior while being insensitive to short-lived frame-to-frame deviations in behavior.

- The metric rapidly builds up magnitude when large prediction errors are observed over a sequence of frames.
- The impact on MTE of a large prediction error in a single frame is quickly dissipated when the error is not sustained in subsequent frames (robustness against noisy deviations).

6.2.6 Adaptive Determination of LLSE Regularization Parameter λ

The LLSE optimization problem $\vec{v} = A\,\vec{u}$ is ill-posed when A is not full-rank. The regularized problem:

$$\begin{bmatrix} \vec{v} \\ 0 \end{bmatrix} = \begin{bmatrix} A \\ \sqrt{\lambda}\,I \end{bmatrix} \vec{u},$$

becomes well-posed because $\begin{bmatrix} A \\ \sqrt{\lambda}\,I \end{bmatrix}$ is always skinny with full column rank for any $\lambda > 0$.

Given $\vec{v}$, LLSE finds $\vec{u}$ to minimize the cost $||A\vec{u}-\vec{v}||^2 + \lambda||\vec{u}||^2$, essentially finding a *low magnitude* solution $\vec{u}$. A low magnitude LLSE solution is considered to avoid *overfitting* to the noise or anomalies in the training data A and $\vec{v}$. Hence a low magnitude solution is considered to *generalize* better to other non-training data. *Therefore, the goal is to ensure that LLSE attempts to minimize both parts of the cost*: $\eta_\lambda = ||A\vec{u}-\vec{v}||^2$ and $\rho_\lambda = ||\vec{u}||^2$. To meet this goal it is important to choose λ such that both η_λ and $\lambda\rho_\lambda$ parts of the cost have *similar magnitude*, otherwise LLSE will essentially minimize only one or the other.

Literature establishes [72, 73, 74] that plotting $\log \eta_\lambda$ vs $\log \rho_\lambda$ while varying λ produces an L-shaped curve, and that λ should be chosen from the "knee" of the L-curve. Mathematical techniques iteratively re-solve the LLSE problem for different values of λ in order to find the point of *maximum curvature* on the "knee" of the L-curve.

We take a simpler approach more appropriate for an online technique like ours, in order to minimize the number of times the LLSE problem has to be re-solved. Our approach is to scale λ based on the *difference in the order-of-magnitudes of the cost terms* after each LLSE solution, until the LLSE solution has η_λ and $\lambda\rho_\lambda$ within an order-of-magnitude of each other. This termination condition approximates the idea of finding the "knee" of the L-curve, as adjusting λ at this solution point will rapidly exchange the magnitudes of the η_λ and ρ_λ terms. The LLSE problem may be re-solved multiple times within a single application frame t until some *budget* is exhausted. λ is optimized across multiple frames even though the history data, and hence the A and v training data, changes from frame-to-frame.

Specifically, we update λ as follows: if $\eta_\lambda > 10\,\lambda\,\rho_\lambda$ or $\eta_\lambda < \frac{1}{10}\lambda\,\rho_\lambda$, we scale λ by $\sqrt{\frac{\eta_\lambda}{\lambda\,\rho_\lambda}}$, to increase or decrease λ, respectively. Hence, update $\lambda \leftarrow \sqrt{\frac{\eta_\lambda}{\lambda\,\rho_\lambda}}\lambda = \sqrt{\frac{\eta_\lambda\,\lambda}{\rho_\lambda}}$.

The next invocation of LLSE would be expected to produce a correspondingly closer gap between the η_λ and $\lambda\rho_\lambda$. Whenever LLSE produces a pathological zero solution $\vec{u} = 0$ (i.e., $\rho_\lambda = ||\vec{u}||^2 = 0$) or a pathological post-update $\lambda = \infty$, we *dramatically reduce* λ to facilitate the next invocation of LLSE to estimate $\vec{u} \neq 0$ (this is a heuristic because we don't always know whether the pathological solution was produced due to bad training data or the excessive magnitude of λ). The dramatic reduction is $\lambda \leftarrow \sqrt{\lambda}$ if $\lambda > 1$, otherwise $\lambda \leftarrow \lambda^2$. Conversely, $\lambda = 0$ may occur due to limited numerical precision, producing a pathological condition where no regularization is performed. We heuristically correct $\lambda = 0$ by replacing with $\lambda \leftarrow 10^{-6}$, to re-enable regularization and allow λ to be tuned in subsequent estimation steps.

The strategy for determining λ, described above, is effective for the following reasons.

- The typical range of values taken by input and output variables are unlikely to change much over the course of execution of the application. This would be because the input ranges are bounded by the programmer, and the fixed range of $\vec{y}$ domain assumption. Therefore, "good" values for the estimated model coefficients will have a "typical" magnitude, for which a narrow range of λ, once found, will continue to effective.

- Large range of tolerance for λ: we only need to make the two cost components comparable in magnitude.

Therefore, when the application behavior changes gradually over frames, subsequent model estimation automatically adjust λ. It can be generally expected that only with low probability can the application behavior change so rapidly that multiple model estimation steps must be expended to determine a suitable λ again (due to the stable application-response assumption, Section 5.4).

6.3 Model Updates to Track Changing Application Behavior

The active model $\mathcal{M}$ is periodically replaced by a substitute model $\mathcal{M}'$. Once estimated, the performance of $\mathcal{M}'$ is compared against $\mathcal{M}$ to determine if indeed $\mathcal{M}'$ has superior performance compared to $\mathcal{M}$. The following subsections provide details.

6.3.1 Estimation of Substitute Model $\mathcal{M}'$

There are three main reasons to estimate a possible substitute model $\mathcal{M}'$ when an active model $\mathcal{M}$ is already available and driving application inputs.

1. The model $\mathcal{M}$ is *unbalanced.* That is, the current value of the regularization parameter λ does not balance the magnitudes of the LLSE cost components η_λ and $\lambda\rho_\lambda$ (Section 6.2.6).
2. The application response characteristics may have changed since $\mathcal{M}$ was estimated.
3. The training data used to estimate $\mathcal{M}$ was of poorer quality than what is possible, and it may be possible to estimate a better model if attempted again on fresh training data.

The coverage requirement places only a lower bound on the amount of application behavior that must be sampled before estimation is allowed. However, there is no a priori known ideal sampling pattern that would work well with any arbitrary application. Hence, even when the application characteristics remain unchanged over a long sequence of frames, *repeated model estimation over different training data sets increases the likelihood of encountering a model that better approximates the application characteristics.* This could be the very first model estimated (i.e., when $\mathcal{M}$ is initially defined) or a later one. Hence, the controller needs to evaluate each newly estimated $\mathcal{M}'$ against the active $\mathcal{M}$, retaining the better model as $\mathcal{M}$ (essentially, *iterative max-finding*).

When $\mathcal{M}$ is unbalanced, the controller repeatedly re-estimates $\mathcal{M}'$, possibly every frame while the training data maintains coverage, until a balanced $\mathcal{M}'$ is found and applied as $\mathcal{M}$. The repeated invocation of LLSE iteratively converges λ to a value that achieves balance. Our expectation is that the range of magnitudes of the inputs $\vec{x}$ and outputs $\vec{y}$ do not change dramatically over a sequence of frames, as they are reflective of the underlying data-ranges used by the application and on the properties of the given data set (the range-of-$\vec{y}$ domain assumption, Section 5.4). Hence, we expect λ to continue converging to a better value even as the training data differs over each invocation of LLSE.

Once a balanced $\mathcal{M}$ is found, the substitute model is estimated after intervals of *at least* L_γ frames, to allow for completely fresh training data in $\mathcal{H}$. If the training data lacks sufficient coverage to allow estimation of $\mathcal{M}'$, the controller uses *probabilistic forced exploration* (PFE) to enhance the coverage of the training data (details in Section 6.4, here we provide only a gist). The frequency of the forced exploration (represented by θ) is partly a function of the current QoS performance of $\mathcal{M}$: greater frequency when $\mathcal{M}$ delivers low QoS, and very low frequency when $\mathcal{M}$ delivers very high QoS (Section 6.4.3: Eq 63 and Eq 67). *Hence, the coverage requirement and the probabilistic forced exploration mechanism further extend the intervals between estimation of successive substitute models (beyond the minimum of L_γ frames), in a manner that balances the QoS deliverable by the current $\mathcal{M}$ against the potential to find a substitute model $\mathcal{M}'$ capable of significantly enhanced QoS.*

6.3.2 Comparing Performance Potential of $\mathcal{M}'$ Against $\mathcal{M}$

At any frame t, only a single input $\vec{x}_t$ can be applied to the application. Hence, we cannot construct a regulator from $\mathcal{M}'$ and apply to the application to observe if better QoS performance is achieved compared to using the regulator $\mathcal{C}$ constructed from $\mathcal{M}$. Therefore, we construct a metric, $\text{Adv}_{\mathcal{M}',\mathcal{M}}$, that compares the *prediction error* of $\mathcal{M}'$ against $\mathcal{M}$: $||\mathcal{M}'(\mathcal{H},t) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})}$ versus $||\mathcal{M}(\mathcal{H},t) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})}$.

The following considerations are important for the metric.

- *Robustness*: we want to avoid frivolous replacements of $\mathcal{M}$, as there can be a high penalty to application QoS when it is likely that a good $\mathcal{M}$ could be replaced by a poorer $\mathcal{M}'$.

- *Adaptive duration of evaluation*: we want the metric to quickly determine (i.e., within very few frames) if one of $\mathcal{M}$ and $\mathcal{M}'$ is distinctly superior to the other, yet allow a more precise evaluation over a long sequence of frames if the models' prediction accuracies are close.

We ensure robustness in the following ways:

- Compare performance over a sequence of samples to protect against short-lived anomalous behavior.

- Use a different sequence of frames for comparing performance than the ones used to estimate $\mathcal{M}'$ (regression data versus training data), as $\mathcal{M}'$ is by definition a low-error fit to the training data used for its estimation.

- Ensure that the metric does not trigger replacement unless the regression sequence of frames also meet the coverage requirement. In other words, we want the evaluation to take place over the *span* of the input space, not a limited sub-range.

Previously, Eq 43 defined the prediction error of $\mathcal{M}$ at frame t as follows.

$$e_t \;=\; \frac{1}{dim(\vec{y})}||\mathcal{M}\left(\mathcal{H},t\right) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})}$$

We will similarly define the prediction error of $\mathcal{M}'$ as follows.

$$e'_t \;\triangleq\; \frac{1}{dim(\vec{y})}||\mathcal{M}'\left(\mathcal{H},t\right) - \vec{y}_t||^2_{(\vec{s},\vec{\delta})} \tag{51}$$

The following metric captures the *performance advantage* of $\mathcal{M}'$ over $\mathcal{M}$ over the sequence of frames since $\mathcal{M}'$ was estimated $(t_{\mathcal{M}'})$ to the current frame t.

$$\text{Adv}_{\mathcal{M}',\mathcal{M}} \;\triangleq\; \sum_{u=t_{\mathcal{M}'}}^{t} \gamma^{t-t'}\frac{e_u - e'_u}{\max(e_u, e'_u)} \tag{52}$$

The metric can be updated recursively every frame as follows:

$$\text{Adv}_{\mathcal{M}',\mathcal{M}} \;\leftarrow\; \frac{e_t - e'_t}{\max(e_t, e'_t)} + \gamma\,\text{Adv}_{\mathcal{M}',\mathcal{M}}, \tag{53}$$

with $\text{Adv}_{\mathcal{M}',\mathcal{M}} \leftarrow 0$ whenever either $\mathcal{M}$ or $\mathcal{M}'$ is re-defined.

The metric is defined over a sequence of frames. The sequence starts after $\mathcal{M}'$ has been estimated, ensuring the sequence is distinct from the training data for $\mathcal{M}'$ (the *regression sequence*). Further, we maintain a separate coverage metric for the regression sequence (i.e., in contrast with κ for $\mathcal{H}$).

When the regression sequence meets coverage and the metric $\mathrm{Adv}_{\mathcal{M}',\mathcal{M}}$ exceeds a threshold, substitute model $\mathcal{M}'$ replaces $\mathcal{M}$. Estimation of a new $\mathcal{M}'$ can occur at the same or a later frame. The following threshold test is used:

$$\mathrm{Adv}_{\mathcal{M}',\mathcal{M}} > \frac{1-\gamma^{\max(L_\gamma,\,\mathrm{count})}}{1-\gamma} \times 0.10, \tag{54}$$

where, count represents the length of the regression sequence.

The test can be interpreted as checking whether $\mathcal{M}'$'s prediction accuracy consistently exceeds $\mathcal{M}$'s by at least 10% every frame averaged over a sufficiently long sequence of frames (when count $\geq L_\gamma$). Alternatively, over a short sequence of frames (when count $< L_\gamma$), $\mathcal{M}'$ must far surpass $\mathcal{M}$ on at least a few frames without significantly under-performing on others, in order for $\mathcal{M}'$ to quickly replace $\mathcal{M}$.

There are three ways in which the regression sequence (i.e., frames since $\mathcal{M}'$ was estimated) can exceed L_γ frames — either *i)* the current $\mathcal{M}'$ exhibits performance inferior to $\mathcal{M}$, *ii)* the performance improvement is only minor (below the 10% threshold), or *iii)* the regression sequence has not yet achieved coverage. In subsequent frames, if the regression sequence achieves coverage before $\mathcal{H}$ achieves coverage, $\mathcal{M}'$ can be evaluated against $\mathcal{M}$ by the test in Eq 54 and can potentially replace $\mathcal{M}$ if better. Otherwise, the existing $\mathcal{M}'$ is discarded and a new one estimated when $\mathcal{H}$ achieves coverage.

Frequency of $\mathcal{M}'$ estimation. L_γ is a duration sufficiently long for the regression sequence to achieve coverage with high probability if forced exploration were performed (Section 6.4). The last L_γ frames of the regression sequence for the current $\mathcal{M}'$ also become the training data for the next $\mathcal{M}'$ to be estimated. When $\mathcal{M}$ delivers high QoS (rather, QoS close to what the controller estimates is the best possible), the exploration frequency is greatly reduced to not adversely impact the performance potential of $\mathcal{M}$, which in turn increases the interval between successive $\mathcal{M}'$ estimations to become *arbitrarily longer* than L_γ frames. Conversely, when $\mathcal{M}$ exhibits poor QoS (rather, QoS far below the controller's estimate of the best possible), the controller creates a high likelihood that exploration frames will occur with sufficient frequency to achieve coverage in approximately L_γ frames, which allows for the frequent estimation and advantage-evaluation of substitute $\mathcal{M}'$'s.

6.4 QoS Maximization by Balancing Exploration versus Exploitation

Exploration consists of applying randomized control inputs on a sequence of frames to characterize the current application response behavior. Two types of behavior exploration mechanisms are used.

- **Continuous forced exploration** (CFE) is intended to estimate $\mathcal{M}$ in the minimum number of frames possible, when $\mathcal{M}$ is currently undefined. Exploration is performed in every frame until $\mathcal{H}$ achieves coverage.
- **Probabilistic forced exploration** (PFE) is intended to periodically boost coverage so that a candidate replacement model $\mathcal{M}'$ is estimated with a desired frequency, and its prediction accuracy can be compared against $\mathcal{M}$. Exploration is performed in clusters of frames, whose duration and frequency of occurrence is determined through statistics.

When the active model is undefined (i.e., $\mathcal{M} = \phi$), CFE is performed every frame and $\mathcal{H}$ is extended in length until coverage is achieved (allowing $\mathcal{M}$ to be estimated). When $\mathcal{M} \neq \phi$ and $\mathcal{H}$ has coverage, no forced exploration is done (as $\mathcal{M}'$ can be readily estimated using $\mathcal{H}$ if the controller chooses to). However, when $\mathcal{M} \neq \phi$ and $\mathcal{H}$ *lacks coverage*, PFE is performed so that coverage can be periodically enhanced to allow the estimation of substitute model $\mathcal{M}'$ and its prediction accuracy evaluated against $\mathcal{M}$.

The CFE mechanism is a high-priority interruption of the normal operation of the controller, with the goal of estimating a model within a short number of frames. In contrast, the PFE mechanism applies during the normal operation of the controller, typically spanning the vast majority of the application frames.

Only on the frames where PFE could potentially occur (i.e., when $\mathcal{M} \neq \phi$ and $\mathcal{H}$ lacks coverage) does the question of balancing exploration versus exploitation arise. Such frames are referred to as *PFE-capable frames.* The controller uses a parameter $0 < \theta < 1$ to represent the *fraction of exploration against exploitation.* On any PFE-capable frame, the PFE mechanism must determine a probability for starting a PFE cluster (if one is not already underway), and how long the cluster should be if one were started. The PFE mechanism is governed by the following parameters.

- q: probability that a PFE cluster is started on current frame t.
- d_{peak}: shapes the probability distribution from which the cluster length d is sampled (whenever the current frame starts a PFE cluster).
- θ: the percentage of PFE-capable frames on which PFE should occur.

The cluster length d is a random variable sampled from a triangular probability distribution shaped by parameter d_{peak}. d_{peak} is adjusted based on L_{PFE}, where L_{PFE} is an *estimate* of the shortest PFE cluster length that could achieve coverage given the current contents of $\mathcal{H}$. In other words, if the PFE cluster consisted only of maximally variant samples, L_{PFE} is the cluster length that will achieve coverage when these cluster samples are inserted into $\mathcal{H}$, while dropping the oldest samples, if necessary, to respect $|\mathcal{H}| \leq L_\gamma$. Appendix A.2 describes how L_{PFE} is estimated.

Let $E\{d\}$ be the expected value of d. Therefore, $E\{d\}$ is a function of d_{peak}. The following relationship exists between q, $E\{d\}$ and θ.

$$q = \frac{\theta}{E\{d\}(1-\theta) + \theta} \tag{55}$$

Therefore, we can set any two and compute the third. In our methodology, we choose to compute q from θ and $E\{d\}$. θ quantifies the exploration-versus-exploitation balance: $\theta = 0$ *is no exploration, all exploitation;* $\theta = 1$ *is all exploration, no exploitation.*

The following subsections explore the parametric probability distribution from which d is sampled, derive the relationship between the exploration parameters in Eq 55, and elaborate how θ is adjusted based on the QoS performance of the active model $\mathcal{M}$.

6.4.1 Probabilistic Distribution of Cluster Length

A cluster length greater than L_γ is not useful for PFE as it represents going beyond a complete replacement of all samples in $\mathcal{H}$ with exploration samples. For reference, the following sample sequence for input x_j — $\left\{\frac{1}{2}N_j,\, -\frac{1}{2}N_j,\, \frac{1}{2}N_j,\, -\frac{1}{2}N_j,\, \ldots\right\}$ of duration L_γ is just sufficient for achieving coverage (See "Determining Coverage Threshold f" in Appendix A.1). PFE will typically produce samples with larger variance, allowing a duration shorter than L_γ to achieve coverage with high probability. For example, a PFE of duration $\frac{L_\gamma}{2}$ consisting of samples $\{N_j,\, -N_j,\, N_j,\, -N_j,\, \ldots\}$ is sufficient for achieving coverage even if the remaining $\frac{L_\gamma}{2}$ samples in $\mathcal{H}$ are all zero. Additionally, regulator $\mathcal{C}$ may by itself sufficiently explore some of the input dimensions, allowing perhaps even a very short PFE cluster to achieve coverage.

Also consider the possibility of multiple PFE clusters occurring one after another separated only by small gaps, so the multiple clusters fit in $\mathcal{H}$. This scenario is likely if q is high. *Thus, multiple short PFE clusters can collectively achieve coverage.*

The PFE cluster length d is sampled from a triangle-shaped parametric probability distribution. The distribution is parameterized on d_{peak}. The distribution takes integral values in the range $(0,\ d_{\text{high}} + 1)$, where $d_{\text{high}} = \min\{2\, d_{\text{peak}},\, L_\gamma\}$. The distribution has a triangular shape with peak probability at d_{peak} and zero probabilities at 0 and $d_{\text{high}} + 1$. Whenever d_{peak} is updated, the controller numerically computes $E\{d\}$ from the distribution. This can be done efficiently at runtime as an explicit summation over the distribution.

The determination of d_{peak} factors in L_{PFE} — an estimate of the shortest cluster length that could achieve coverage given the current $\mathcal{H}$, and numClusters — the number of PFE clusters to split L_{PFE} exploration frames into.

$$d_{\text{peak}} \leftarrow \frac{L_{\text{PFE}}}{\text{numClusters}} \tag{56}$$

numClusters defines the number of clusters in a *PFE cluster group* — the goal being to achieve coverage after multiple PFE clusters. *The last cluster in a cluster group is always extended indefinitely until coverage is achieved.*

Having multiple clusters is useful in two ways: *i)* helps minimize the *local disruption* to application QoS when L_{PFE} is large by splitting it, and *ii)* when exploration is highly desired (high θ) but the probability of starting a cluster (q) is made trivially small because L_{PFE} is too large, splitting into multiple clusters boosts q (note that $E\{d\}$ grows with $d_{\text{peak}} = L_{\text{PFE}}$

without multiple clusters, eventually making q in Eq 55 trivially small despite $\theta \approx 1$).

To achieve the benefits of multiple clusters described above, we heuristically define numClusters as the smallest positive integer that produces q (via Eq 56 and Eq 55) such that $q \geq \dfrac{\theta}{5}$ for the given θ and L_{PFE} (details in Appendix A.2).

Benefits of PFE. The PFE mechanism described so far has been designed to have the following useful properties.

1. Allows coverage to be achieved in close to L_{PFE} exploration frames, for any specified L_{PFE}.

2. Utilizes randomization to allow exploration with any arbitrary desired frequency θ (something difficult for exploration with a deterministic pattern to achieve).

3. Confers a probabilistic structure to the exploration, to avoid repeatedly triggering perverse interactions with an application (analogous to mitigating unintended resonance in physical systems).

4. Mitigates excessive local disruption to the regular operation of the active model by suitably splitting exploration into multiple clusters.

6.4.2 Relationship between Exploration Parameters

The parameters q, θ and $E\{d\}$ (really, a function of d_{peak}) are related by Eq 55. To derive this relationship, consider a contiguous sequence of frames of length L over which $\mathcal{M} \neq \phi$ and $\mathcal{H}$ does not have coverage. Let the sequence of frames be partitioned into K *epochs*, where each epoch is either an entire PFE cluster or a single frame on which $\mathcal{C}$ drives input. Hence, on the first frame of each epoch, the controller decides to pursue forced exploration with probability q. Let the epochs be indexed by k for $1 \leq k \leq K$. Let indicator function $I_{\text{fe}}(k) = 1$ if the k^{th} epoch is a PFE cluster with cluster length d_k, and $I_{\text{fe}}(k) = 0$ if the epoch is a single frame with inputs driven by $\mathcal{C}$ (with d_k undefined). Then, the following equation holds.

$$L = \sum_{k=1}^{K} \left[I_{\text{fe}}(k)\, d_k \; + \; (1 - I_{\text{fe}}(k))\, 1 \right] \tag{57}$$

$$= K\, q\, E\{d\} \; + \; K\,(1-q) \tag{58}$$

Strictly speaking, $E\{d\}$ in the above equation is the average value of the d_k's observed over the forced exploration epochs. In contrast when we compute $E\{d\}$ as the mean of the triangular distribution parameterized by d_{peak}, we are making a *projection* that if the current coverage gap keeps re-appearing (because of the same application behavior continuing and the same regulator $\mathcal{C}$ operating), we would find that the computed $E\{d\}$ matches the observed $E\{d\}$ because d_k's represent samples from that same distribution.

Note that θ is the fraction of frames in L that undergo forced exploration. That is,

$$\begin{aligned}
\theta &= \frac{\sum_{k=1}^{K} I_{\text{fe}}(k)\, d_k}{L} \\
&= \frac{K\, q\, E\{d\}}{L} \\
&= \frac{q\, E\{d\}}{(1-q) + q\, E\{d\}}.
\end{aligned}$$

Re-factoring terms to extract q, we get Eq 55:

$$q \; = \; \frac{\theta}{E\{d\}(1-\theta) + \theta}.$$

6.4.3 Using Model QoS to Adjust Exploration versus Exploitation

Two considerations drive the selection of θ.

1. Given the current QoS performance of $\mathcal{M}$, how much disruption to the QoS from the forced exploration would be considered *tolerable*?

2. Is it likely that a model with much better QoS performance than $\mathcal{M}$ can be estimated? If yes, we want to boost exploration. If not, we want to minimize the disruption to whatever QoS $\mathcal{M}$ can provide.

The QoS performance of the controller is quantified by two separate metrics: the satisfaction ratio ($\texttt{SR}$) and the mean-squared-error QoS ($\texttt{MSEQ}$). We need θ to optimize for both. The $\texttt{SR}$ metric is the fraction of frames that fully satisfy the QoS requirements (i.e., frames with $\tau \leq 1.0$). The controller is also required to minimize the QoS deviations, which are captured by $\texttt{MSEQ}$ as the average τ over the frames. Note that $\texttt{SR}$ incurs the same penalty for τ slightly greater than 1.0 and τ significantly exceeding 1.0. $\texttt{MSEQ}$ distinguishes between the two cases, but does not distinguish between τ very slightly less than 1.0 and τ very slightly greater than 1.0. Hence, we need θ to optimize for both metrics.

Optimizing for Satisfaction Ratio. $\hat{\texttt{SR}}_{\mathcal{M}}$ is the *estimate* of the $\texttt{SR}$ that $\mathcal{M}$ is capable of delivering, i.e., the $\texttt{SR}$ if the regulator created from $\mathcal{M}$ drove application inputs continuously over a sequence of frames with no forced exploration at all. In practice, the controller can only observe the application behavior with forced exploration frames mixed in with regulator-driven frames. Therefore, we define $\hat{\texttt{SR}}_{\mathcal{M}}$ as follows to attempt to filter out the impact of any forced exploration frames mixed in.

$$\hat{\texttt{SR}}_{\mathcal{M}} \triangleq \frac{\sum_{t'=t_{\mathcal{M}}+W}^{t-1} I\left\{\mathcal{M} \text{ drove inputs in frame } t'\right\}\ I\left\{\tau_{t'} \leq 1\right\}}{\sum_{t'=t_{\mathcal{M}}+W}^{t-1} I\left\{\mathcal{M} \text{ drove inputs in frame } t'\right\}}, \tag{59}$$

with $\hat{\texttt{SR}}_{\mathcal{M}} = 1$ assumed for at least the first W frames after $\mathcal{M}$'s estimation, and until $\mathcal{M}$ gets to drive a frame input. Since the measured objectives are averaged over a moving window of W frames, the first W frames are skipped to drain out the possible impact of prior behavior.

While $\hat{\texttt{SR}}_{\mathcal{M}}$ characterizes $\mathcal{M}$'s performance over the entire sequence of frames for which $\mathcal{M}$ is the active model, we need an additional metric, $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}}$, to characterize the *current* performance of $\mathcal{M}$ over the most recent sequence of frames, particularly so the controller can quickly detect if $\mathcal{M}$ is no longer suitable due to changing application behavior. We choose to compute $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}}$ over the most recent $10 * W$ frames, since this duration is long enough to be perceptible to the user (hence, any observed deviations are of consequence, not noise).

$$\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} \triangleq \frac{\sum_{t'=\max(t_{\mathcal{M}}+W,\, t-10*W)}^{t-1} I\left\{\mathcal{M} \text{ drove inputs in frame } t'\right\} \; I\left\{\tau_{t'} \le 1\right\}}{\sum_{t'=\max(t_{\mathcal{M}}+W,\, t-10*W)}^{t-1} I\left\{\mathcal{M} \text{ drove inputs in frame } t'\right\}}, \tag{60}$$

with $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} = 1$ for at least the initial W frames after a new $\mathcal{M}$ becomes active, and until the first frame driven by $\mathcal{M}$.

Let $\hat{\texttt{SR}}_{\text{Ach}}$ be the estimate of the best SR performance *achievable* on the current execution of the application, based on the SR performances of the past active models. We heuristically define $\hat{\texttt{SR}}_{\text{Ach}}$ as follows.

$$\hat{\texttt{SR}}_{\text{Ach}} \triangleq \mathit{Average}\left\{\begin{array}{c} \text{Top 25\% of all } \hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} s \text{ observed so far,} \\ \text{with each } \hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} \text{ sampled every } 10 * W \text{ model-driven frames,} \\ \text{with an initial } \hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} = 1 \text{ assumed at application startup} \end{array}\right\} \tag{61}$$

Let θ^{tol} be the amount of exploration that is *tolerable* given the current performance $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}}$ of the active model $\mathcal{M}$. We *heuristically* define θ^{tol} as follows.

$$\theta^{\text{tol}} = 0.5 * (1 - \hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}}) \tag{62}$$

Finally, the following combines the tolerance and achievability components of SR into a recommendation for the next θ to apply in the controller.

$$\boxed{\theta \;=\; \theta^{\text{tol}}\, \hat{\texttt{SR}}_{\text{gap}} + 0.01,} \tag{63}$$

where, the SR achievability gap is defined as follows.

$$\hat{\texttt{SR}}_{\text{gap}} \triangleq \begin{cases} \dfrac{\hat{\texttt{SR}}_{\text{Ach}} - \hat{\texttt{SR}}_{\mathcal{M}}}{\hat{\texttt{SR}}_{\text{Ach}}} & \text{, if } +ve \\ 0 & \text{, otherwise.} \end{cases} \tag{64}$$

The 0.01 term allows for a minimal forced exploration even when $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}} = 1$ (and therefore, $\theta_{\texttt{SR}}^{\text{tol}} = 0$), or when $\hat{\texttt{SR}}_{\mathcal{M}} \ge \hat{\texttt{SR}}_{\text{Ach}}$. Notice that the tolerance term uses $\hat{\texttt{SR}}_{\mathcal{M}}^{\text{cur}}$ to be responsive to the current behavior, while the achievability term uses $\hat{\texttt{SR}}_{\mathcal{M}}$ for stability against intermittent noise.

Optimizing for Mean Squared Error QoS. Optimizing for $\texttt{SR}$ also optimizes for $\texttt{MSEQ}$ when $\tau \leq 1.0$ on most frames and frames with $\tau \gg 1.0$ are infrequent. Unfortunately, in the general case, feedback-based θ optimizations that track $\texttt{SR}$ are not sensitive to the extent to which τ exceeds 1.0. Due to this lack of sensitivity, when $\tau \gg 1$ for most frames the θ optimizations for improving $\texttt{SR}$ lose efficacy. Instead, in such a situation, θ ought to be adjusted to reduce τ, which would directly improve $\texttt{MSEQ}$ and would also improve $\texttt{SR}$ if sufficient reduction in τ is possible.

We generalize the technique presented previously for improving $\texttt{SR}$ to now improve $\texttt{MSEQ}$. Improving $\texttt{SR}$ becomes an important special-case of improving $\texttt{MSEQ}$. Consider the *generalized objective* — satisfy $\tau \leq k$ for an arbitrary choice of $k \geq 1.0$. Let $\texttt{SR}(k)$, $\hat{\texttt{SR}}_{\mathcal{M}}(k)$, $\hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}(k)$, and $\hat{\texttt{SR}}_{\text{Ach}}(k)$ represent the corresponding *generalized metrics* characterizing the achievement of the generalized objective $\tau \leq k$. To summarize:

- $\texttt{SR}(k)$ — the QoS performance on objective $\tau \leq k$ over the full execution of the application.
- $\hat{\texttt{SR}}_{\mathcal{M}}(k)$ — the measured performance of $\mathcal{M}$ on objective $\tau \leq k$, since $\mathcal{M}$ became the active model.
- $\hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}(k)$ — the measured performance of $\mathcal{M}$ on objective $\tau \leq k$, over the most recent $10 * W$ frames.
- $\hat{\texttt{SR}}_{\text{Ach}}(k)$ — the estimated performance achievable for objective $\tau \leq k$, based on the observed performance of past active models.

First, note that any model $\mathcal{M}$ will produce arbitrarily good $\texttt{SR}(k)\ \forall\, k \geq k_{\text{tr}}$, for a suitably large k_{tr} (since $\texttt{SR}(k)$ is an increasing function of k, eventually becoming $= 1$). Second, note that a model $\mathcal{M}$ that delivers good performance at $\texttt{SR}(k_{\text{tr}})$ has no guarantee to provide either good or bad $\texttt{SR}(k)$ for $1 \leq k < k_{\text{tr}}$. *Our strategy is to reduce $\texttt{MSEQ}$ indirectly, by finding the k_{tr} that balances the undesirability of widening the objective to $\tau \leq k_{tr}$ against the improved $\texttt{SR}(k_{tr})$ that becomes possible.* We refer to k_{tr} as the *performance tracking level*, i.e., the choice of k at which the controller will track $\texttt{SR}(k)$ to optimize θ.

For tractability, we discretize the allowed values of k to $\{1.0, 1.50, (1.50)^2, \ldots, (1.50)^{l_{\max}}\}$, referred to as the *levels of* k. $k = (1.50)^l$ is referred to as the l^{th} level. $l_{\max} = \left\lceil \frac{\log \tau_{\max}}{\log 1.50} \right\rceil$, where $\tau_{\max} =$ maximum τ encountered for a frame so far. Every frame the level that locates the largest improvement in failure rate is chosen as k_{tr}. The largest improvement is indicative of *traction*, i.e., the τ's for a large fraction of the application frames fall between k_{tr} and $1.50 \times k_{\text{tr}}$. Therefore, adjustments to θ based on feedback of observed $\texttt{SR}(k_{\text{tr}})$ will likely produce an improvement in subsequent $\texttt{SR}(k_{\text{tr}})$, improving $\texttt{MSEQ}$ at $k_{\texttt{tr}}$.

$$\boxed{\begin{aligned} l_{\text{tr}} &\leftarrow \underset{0 \le l < l_{\max}}{\text{argmax}} \frac{\hat{\texttt{SR}}_{\text{Ach}}((1.50)^{l+1}) - \hat{\texttt{SR}}_{\text{Ach}}((1.50)^{l})}{l+1} && (65) \\ k_{\text{tr}} &\leftarrow (1.50)^{l_{\text{tr}}} && (66) \end{aligned}}$$

We use $\hat{\texttt{SR}}_{\text{Ach}}(k)$ to determine k_{tr} as that is the current estimate of the best performance possible by any model for objective $\tau \le k$. The $l+1$ denominator creates a preference for choosing a smaller k_{tr}.

Initially at application startup when no knowledge about application behavior is available, we assume $\hat{\texttt{SR}}_{\text{Ach}}(k) = 1.0\ \forall k$. Eq 66 will produce $k_{\text{tr}} = 1$, which will cause the controller to initially optimize θ based on $\texttt{SR}$. After encountering more frames the various $\hat{\texttt{SR}}_{\text{Ach}}(k)$ metrics can be expected to take values closer to the ground truth, and the controller may gradually pick a larger k_{tr}. Later, if the application behavior changes to allow estimation of models $\mathcal{M}$ that achieve a tighter objective, the $\hat{\texttt{SR}}_{\text{Ach}}(k)$ metrics for a smaller k will gradually improve, leading to subsequent reduction in k_{tr} to a tighter feasible level k.

On any given frame, θ is determined as follows (a generalization of Eq 63).

$$\boxed{\theta = \theta^{\text{tol}}(k_{\text{tr}})\ \hat{\texttt{SR}}_{\text{gap}}(k_{\text{tr}}) + 0.01,} \tag{67}$$

where,

$$\theta^{\text{tol}}(k) = 0.5 * (1 - \hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}(k)),$$

$$\hat{\texttt{SR}}_{\text{gap}}(k) = \begin{cases} \dfrac{\hat{\texttt{SR}}_{\text{Ach}}(k) - \hat{\texttt{SR}}_{\mathcal{M}}(k)}{\hat{\texttt{SR}}_{\text{Ach}}(k)} & \text{, if } +ve \\ 0 & \text{, otherwise,} \end{cases}$$

The above derivations are valid under an implicit assumption, which we explore here. Let $\hat{\tau}_{\text{exp}}$ represent the average τ over the frames that underwent exploration. Let $\hat{\tau}_{\mathcal{M}}$ represent the average τ on the frames that $\mathcal{M}$ drove application inputs. We have so far assumed $k_{\text{tr}} < \hat{\tau}_{\text{exp}}$, i.e., *we assume that the exploration frames will always deteriorate the QoS performance that an estimated model can achieve for the application.* This assumption may not be true under certain pathological conditions, such as under the following situations.

- Due to noise in the training data, a bad $\mathcal{M}$ is determined and applied on a sequence of frames.
- Due to noise, the regulator design heuristics determine a poor $\mathcal{C}$ on some frames.
- The application does not satisfy the domain requirements. Here, the QoS performance of an estimated $\mathcal{M}$ can be worse on average than having randomly generated inputs.

The scope of our work precludes dealing with the last situation. In the other situations arising out of intermittent noise, the use of long-term metrics filters out the effects of noise — $\hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}(k)$ averages over $10 * W$ frames, $\hat{\texttt{SR}}_{\mathcal{M}}(k)$ over frames since $\mathcal{M}$ was estimated, and $\hat{\texttt{SR}}_{\text{Ach}}(k)$ over all the active models estimated so far. *In general, the satisfaction of the domain assumptions implies that with high probability $\hat{\tau}_{\mathcal{M}} < \hat{\tau}_{exp}$.*

6.5 Regulator Construction with LQR

LQR is used to construct a regulator $\mathcal{C}$ from the active model $\mathcal{M}$. Most of the LQR design parameters are determined from the structure of model $\mathcal{M}$, the actual $\mathcal{M}$ estimated, and the controller problem specification. These include the state-space representation $\vec{s}_t$, the matrices describing the linear dynamical system, the target trajectory $\vec{r}_t$, the state-cost matrix Q, and the final state-cost matrix $Q_f = Q$. However, the horizon N and the input-costs matrix R remain to be determined. In the traditional offline-design setting of LQR applied to an a priori fixed system model, N and R can be tuned by the human designer based on intuition and trial-and-error until a regulator with desirable performance characteristics is arrived at. Typically, N can be set very large if a steady-state regulator is desired (a fixed K and K^v feedback-control matrix applied at every time-step), or a shorter

N can produce a sequence of K_k and K_k^v feedback-control matrices, $0 \leq k < N$, applied over N time steps. R is typically chosen such that the control inputs in $\vec{u}_t$ tend to maintain magnitudes that the system is capable of accepting and operating efficiently under (say, a system may have hard limits on control inputs, and low magnitude control inputs may help maintain a low fuel-burn rate).

Given the online setting of our controller, the likelihood that the application characteristics (and hence the estimated model) can change frequently, and the lack of detailed a priori knowledge about the application characteristics, we can rely only on the observed performance of the applied regulator to determine if N and R are suitable. For generality, we choose to potentially re-design regulator $\mathcal{C}$ at every frame t, even as the active model $\mathcal{M}$ stays unchanged for multiple frames. That is, we choose to fix $N = 1$. We construct a directed search algorithm that adjusts the coefficients of R by iteratively constructing $\mathcal{C}$ using LQR, potentially multiple times per frame. The algorithm examines the properties of the input $\vec{u}_t$ produced by the current $\mathcal{C}$ and the variation in the input from that produced by the previous design iteration of $\mathcal{C}$, to determine how best to adjust R towards producing the next design iteration of $\mathcal{C}$.

In the typical offline setting, the regulator solution produced by LQR converges quickly. However, in our modified online setting, we continue to adjust R over each successive iteration of LQR applied to the previous solution (essentially, we repeatedly solve with $N = 1$, with the new iterative solution replacing the previous one, rather than generating a solution sequence of length $N > 1$).

6.5.1 Adaptive Correction to Input-Costs Matrix R

Based on system model $\mathcal{M}$ and the observed current state $\vec{s}_t$ the constructed regulator $\mathcal{C}$ determines the input $\vec{x}_t$ to apply at time-step t. However, one or more dimensions x_j of the input vector may exceed bounds, i.e., $|x_j| > N_j$. The inputs are clipped to bounds (and rounded to closest integer) before being applied to the application. However, an LQR regulator $\mathcal{C}$ that projects state trajectory using inputs quite different from those actually applied to the application will have a compromised ability to control application state. As an illustration, consider two LQR regulators, $\mathcal{C}_1$ and $\mathcal{C}_2$, constructed from $\mathcal{M}$. $\mathcal{C}_1$ projects that a large $|x_j|$ will best adjust state $\vec{s}_t$, only to have x_j clipped. In contrast, a large input-cost R_{jj} forces $\mathcal{C}_2$ to pick a low magnitude x_j, while also exercising other inputs x_i to adjust state since x_j by itself is of insufficient magnitude. $\mathcal{C}_1$ expects to see a large correction in $\vec{s}_t$ towards the desired trajectory, but this doesn't happen as the application only sees a clipped input. On the other hand, $\mathcal{C}_2$ only expects to see a modest improvement in state $\vec{s}_t$, and the actual effect is close to what $\mathcal{C}_2$ projected, allowing $\mathcal{C}_2$ to make continued adjustments over multiple time-steps along a trajectory it can project with some accuracy.

For any given $\mathcal{C}$ constructed however optimally, state $\vec{s}_t$ can be constructed so that $\mathcal{C}$ produces next input with a dimension x_j that exceeds its bounds. Hence, rather than attempt to determine a best *fixed* input-costs matrix R, we need to *dynamically adapt* R based on actual observed states and produced inputs.

Goals. We seek to achieve the following goals when we adapt R:

1. Ideally, the input $\vec{x}_t$ should be determined based solely on $\mathcal{M}$ and $\vec{s}_t$. That is, the entries of R should be sufficiently small that the designed regulator $\mathcal{C}$ produces inputs that are projected to absolutely minimize state tracking error, disregarding the magnitude of the inputs.

2. Except, when some input dimension exceeds bounds ($|x_j| > N_j$), the R_{jj} entry should be increased in magnitude to force a re-designed regulator $\mathcal{C}$ to just borderline exceed bounds: maintain $|x_j| \geq N_j$ while minimizing $|x_j| - N_j$. This allows $\mathcal{C}$ to produce saturating inputs (i.e., $|x_j| = N_j$ after clipping) when the situation demands, while still retaining the ability to effectively project system state (like $\mathcal{C}_2$, and unlike $\mathcal{C}_1$ in the earlier example).

At time-step t, the current state $\vec{s}_t$ is already fixed. The projected states $\hat{\vec{s}}_{t+1}$, $\hat{\vec{s}}_{t+2}$, ... and the next inputs used $\vec{x}_t$, $\vec{x}_{t+1}$, ... will depend on the regulator $\mathcal{C}$. Hence, we have the *opportunity to evaluate alternative designs* for $\mathcal{C}$ at time-step t until the projected inputs and projected states appear most suitable (e.g., a regulator that closely tracks the desired state trajectory, while producing inputs with the least violation of the input-bounds constraints).

However, to keep the regulator design process tractable and have low overhead at run-time, we will restrict ourselves to *iteratively refine* the design of $\mathcal{C}$ within a given time-step, and terminate the iterative process as soon as a design satisfies a *sufficiency criterion*. In particular, we restrict ourselves to refining $\mathcal{C}$ by only adjusting the entries of the diagonal input-costs matrix R.

Strategy. We attempt to classify the *condition* of x_j as one of the following:

1. **Under-constrained** – R_{jj} is sufficiently small that the designed regulator $\mathcal{C}$ essentially disregards the magnitude of x_j and finds the best inputs based solely on state $\vec{s}_t$ and model $\mathcal{M}$.

   If $|x_j| \leq N_j$, this is a highly desirable condition, which we refer to as **desirably under-constrained**. When $|x_j|$ significantly exceeds N_j, the condition instead becomes **problematically under-constrained** and must be corrected.

   The under-constrained condition is tested by examining whether changes to R_{jj} *fail to produce* any significant changes to $|x_j|$ after regulator refinement. However, care must be taken to differentiate against the situation when $|x_j|$ fails to change significantly because R_{jj} is already so large that the trajectory-tracking-error component of the LQR cost is essentially ignored compared to the input-magnitude component of the LQR cost. This latter situation will also exhibit $x_j \approx 0$ due to the dominance of the input cost in LQR, and an additional test will allow us to definitively distinguish the over-constrained condition from under-constrained (see over-constrained condition below).

   When desirably under-constrained, we would like to stop further adjustments to R_{jj} as a suitable value has already been determined. When problematically under-constrained, we would like to increase R_{jj} until the condition dissipates.

2. **Over-constrained** – R_{jj} is large enough that $\mathcal{C}$ is constrained to produce a lower-magnitude input x_j, even though a higher-magnitude input would be better suited to respond to current state $\vec{s}_t$.

   This condition is detected *either* when

- $|x_j| \leq N_j$ and adjusting R_{jj} produces corresponding adjustments in x_j after regulator refinement, or
- $x_j \approx 0$, adjusting R_{jj} produces no significant changes in x_j after regulator refinement, and $(0.5)^2 * R_{jj} \gg (\vec{\tilde{y}}_t - \vec{\tilde{y}})^T \, Q \, (\vec{\tilde{y}}_t - \vec{\tilde{y}})$. Here, $\vec{\tilde{y}}_t$ is the projected next output when input $\vec{x}_t$, produced by $\mathcal{C}$, is applied to model $\mathcal{M}$.

The above comparison test, $(0.5)^2 * R_{jj} \gg (\vec{\tilde{y}}_t - \vec{\tilde{y}})^T \, Q \, (\vec{\tilde{y}}_t - \vec{\tilde{y}})$, is determined by examining the LQR cost function J in Eq 20, which for each time-step consists of a trajectory-tracking-error component and an input-cost component. Let $TT(u_j) = (C\vec{s}_{t+1|u_j} - \vec{r}_t)^T \, Q \, (C\vec{s}_{t+1|u_j} - \vec{r}_t) = (\vec{\tilde{y}}_{t|u_j} - \vec{\tilde{y}})^T \, Q \, (\vec{\tilde{y}}_{t|u_j} - \vec{\tilde{y}})$ represent the trajectory-tracking-error component of J due to choosing input $\vec{x}_t = \mathcal{C}(\vec{s}_t)$ but with the j^{th} input dimension over-ridden: $x_j = u_j$. Also, let $IC(u_j) = (u_j)^2 \, R_{jj}$ capture the input-cost component of applying the over-ridden $\vec{x}_t$, but only for the j^{th} input dimension. *Then, the comparison test above can be expressed as* $IC(u_j = 0.5) \gg TT(x_j)$, where x_j is the original, non-over-ridden value of the j^{th} input dimension. Our goal is to diagnose if an excessive magnitude R_{jj} is forcing $\mathcal{C}$ to produce $x_j \approx 0$ when a larger magnitude x_j^{best} would have tracked the trajectory better, i.e., $TT(x_j) \geq TT(x_j^{best})$. $u_j = 0.5$ is chosen as the *least magnitude value that would be rounded to a non-zero integer* when applied to the application.

Let's consider the situation where $x_j \approx 0$ has been detected over two consecutive refinement steps. To determine if R_{jj} is so large that the input cost *significantly dominates* the trajectory-tracking-error cost (thereby being the cause of $x_j \approx 0$), we consider the following mutually exclusive and collectively exhaustive cases:

(a) $IC(u_j = 0.5) \gg TT(x_j)$. Consider two sub-cases:

- Assume $|x_j^{best}| \geq 0.5$. $\therefore IC(x_j^{best}) \geq IC(u_j = 0.5) \gg TT(x_j) \geq TT(x_j^{best})$ $\Rightarrow IC(x_j^{best}) \gg TT(x_j^{best})$. R_{jj} over-constrains and precludes application of input x_j^{best}, $\therefore$ reduce R_{jj}.
- Assume $|x_j^{best}| < 0.5$. Reducing R_{jj} is not harmful as the rounded integral

input applied to the application will remain 0. Also, R_{jj} will not indefinitely keep getting reduced with each refinement step. Reduction will stop eventually when $IC(u_j = 0.5) \not\gg TT(x_j)$.

The controller does not have visibility into which of the two sub-cases may be occurring. However, reducing R_{jj} in the current refinement step is either immediately helpful or at least helps the controller diagnose the situation better over subsequent iterations while not being harmful in the current refinement step.

(b) $IC(u_j = 0.5) \sim TT(x_j)$. Because $IC(u_j = 0.5) \geq IC(x_j)$, one of the following must be true:

- $IC(x_j) \sim TT(x_j)$. $\therefore x_j^p \not\approx x_j^r$, where x_j^p and x_j^r are the values of x_j produced over consecutive refinement steps of R_{jj} where R_{jj} was scaled by a non-trivial scale-factor. Therefore, at least one of the following must be true: $x_j^p \not\approx 0$ or $x_j^r \not\approx 0$, contradicting the primary assumption that x_j stays ≈ 0 over subsequent refinement steps.
- $IC(x_j) \ll TT(x_j)$. This represents a *desirably under-constrained* condition described previously and no adjustment to R_{jj} is needed.

(c) $IC(u_j = 0.5) \ll TT(x_j)$. $\therefore IC(x_j) \ll TT(x_j)$, again desirably under-constrained, requiring no further adjustment to R_{jj}.

Therefore, the test $IC(u_j = 0.5) \gg TT(x_j)$, conducted when x_j stays ≈ 0 over consecutive refinement steps, precisely indicates whether R_{jj} is having an over-constraining or under-constraining effect. Note that while the entities x_j^{best}, $IC(x_j^{best})$ and $TT(x_j^{best})$ are used to theoretically justify the test, the test never needs their actual values to be known.

3. **Barely under-constrained** – without the constraining effect of R_{jj} during regulator design, x_j would exceed bounds.

   This condition is detected when $|x_j|$ is close to N_j, and small adjustments to R_{jj} cause $|x_j|$ to swing from $\leq N_j$ to $> N_j$ and vice versa after regulator refinement. When barely under-constrained, we would like to make *ever finer* adjustments to R_{jj}

to draw $|x_j|$ closer to N_j. The motivation is to ensure that x_j gets maximally applied when demanded by state $\vec{s}_t$, yet $\mathcal{C}$ is forced to possibly exercise other inputs x_i to produce additional needed correction to state.

4. **No inference** – when none of the above conditions can be reliably detected.

   This condition occurs, in particular, when the adjustments to R_{jj} produce unexpected changes to x_j. For example, x_j increases in magnitude despite an increase in magnitude to R_{jj}. Such a situation will commonly occur when adjustments to R_{ii} of other inputs x_i produce a larger effect on x_j than the adjustment of R_{jj}.

Iterative Refinement Process. At time-step t, let $\mathcal{C}^k$ identify the sequence of LQR regulators, each one refined from the previous using a single LQR design step. $\mathcal{C}^0$, $\mathcal{C}^1$, $\mathcal{C}^2$, …, represent the successive LQR design iterations until a regulator meeting sufficiency requirements is arrived upon (or the regulator design budget is exhausted for the current time-step).

Usually, $\mathcal{C}^0$ would represent the final LQR regulator produced in the previous time-step $t-1$, serving as the initial design for time-step t. However, whenever a new model $\mathcal{M}$ is estimated, the last regulator design is invalidated. $\mathcal{C}^0$ is determined by the first LQR design step from the new model. However, the R from the last regulator is still used in the first design step, as a best initial guess, with the understanding that subsequent refinement steps will correct R as needed. R is arbitrarily initialized to the identity matrix the very first time LQR is invoked in an application's execution.

The refinement process is applied independently for each input dimension x_j. The k^{th} step takes the following information as input:

- The *previous* regulator $\mathcal{C}^{k-1}$, designed using input-cost R_{jj}^{k-1}, and producing input value x_j^{k-1}.

- The resulting *refined* regulator $\mathcal{C}^k$, designed using input-cost R_{jj}^k, and producing input value x_j^k.

- Pre-conditions: flag f_j^{bu} indicating whether x_j^{k-1} was considered *barely under-constrained*, and flag f_j^{term} signaling whether step $k-1$ had indicated a termination of the refinement process for input x_j.

The k^{th} refinement step produces the following information:

- A value for R_{jj}^{k+1}, produced either by *refining* R_{jj}^k, by *holding* R_{jj}^k, or by *reverting* to the older R_{jj}^{k-1} if the previous refinement step $k-1$ produced neither an improved value in R_{jj}^k nor actionable information for further refinement.
- Post-conditions: flag f_j^{bu} indicating if x_j^k is considered barely under-constrained, and flag f_j^{term} indicating whether the refinement process can be terminated for input x_j. $f_j^{\text{term}} = \text{true}$ also indicates that either a *hold* or a *revert* action was taken at step k.

Initially, we only know x_j^0, R_{jj}^0, $\mathcal{C}^0$ and flag f_j^{bu} from time-step $t-1$, but no refinement action has been taken on R_{jj}. Tables 6.1–6.4 detail the tests to determine the condition of x_j (the conditions were described previously) and the refinement action taken. Table 6.1 describes the initial step $k=0$. For the next step $k=1$, Table 6.2 is followed when the previous time-step determined x_j *not* to be barely under-constrained (i.e., when $f_j^{\text{bu}} = \text{false}$). Otherwise, Table 6.3 is followed when $f_j^{\text{bu}} = \text{true}$. Subsequent steps have to additionally consider whether the refinement for x_j was terminated in a prior step. If terminated (indicated by pre-condition $f_j^{\text{term}} = \text{true}$), Table 6.4 describes whether to continue holding R_{jj} at a fixed value or to restart refinement. If not terminated (pre-condition $f_j^{\text{term}} = \text{false}$), Table 6.2 is followed under the pre-condition $f_j^{\text{bu}} = \text{false}$, and Table 6.3 is followed under the pre-condition $f_j^{\text{bu}} = \text{true}$.

Different input dimensions x_i and x_j may need to refine over a differing number of iterations. There is only a single regulator $\mathcal{C}^k$ produced at step k, not a separate one for each input dimension. In turn, $\mathcal{C}^k$ produces a vector $\vec{x}_t^{\,k}$ that refines all input dimensions simultaneously. Hence, a dimension x_j may terminate its refinement earlier (i.e., R_{jj} is held constant for subsequent iterations) compared to another dimension x_i, which may continue to refine for additional steps. Further, continued refinement iterations for x_i may change the conditions under which x_j had previously terminated. For example, if x_j had terminated with x_j bounded, further refinement of $\mathcal{C}$ due to x_i may cause $|x_j| > N_j$. The

Table 6.1: LQR input-costs: Initial Refinement Step

← $\vec{s}_{t-1}$	$\vec{s}_t$ →	
R^M_{jj}	$= R^0_{jj}$	$R^1_{jj} =?$
$x^M_{j\|t-1}$	$x^0_{j\|t}$	
Case 1: $f^{bu}_j =$ true (was barely under-constrained)	retain c_j	
	(a) $\|x^0_{j\|t}\| > N_j$	$\Rightarrow R_{jj} \uparrow$
	(b) $\|x^0_{j\|t}\| \leq N_j$	$\Rightarrow R_{jj} \downarrow$
Case 2: $f^{bu}_j =$ false (was *not* barely under-constrained)		
	(a) $\|x^0_{j\|t}\| > N_j$	$\Rightarrow R_{jj} \uparrow$ (Problematically under-constrained)
	(b) $\|x^0_{j\|t}\| \leq N_j$	$\Rightarrow R_{jj} \downarrow$ (default action, to generate data)

refinement of x_j would need to *restart* until it again satisfies a termination criterion for one of the conditions. Overall, the need to refine any one dimension x_i triggers another step of refinement for $\mathcal{C}$, until either the runtime budget for regulator refinement is exhausted for the current time-step t, or all the inputs dimensions indicate termination of their refinement in the same step k.

Table 6.2: LQR input-costs: Continued Refinement Step, when f_j^{term} = false and f_j^{bu} = false

	R_{jj}^{k-1} x_j^{k-1}	R_{jj}^{k} x_j^{k}	$R_{jj}^{k+1} =?$
Invariant:	$f_j^{\text{term}} = \text{false} \quad \Rightarrow$	$R_{jj}^{k-1} \neq R_{jj}^{k}$	
Case 3:	$\lvert x_j^{k-1}\rvert \leq N_j$	$\lvert x_j^{k}\rvert \leq N_j$	
(a)	$R_{jj} \updownarrows \not\Rightarrow \lvert x_j\rvert \updownarrows$ and $(x_j \approx 0 \wedge IC(0.5) \gg TT(x_j))$		$\Rightarrow R_{jj} \downarrow$. (No impact, over-constrained)
(b)	$R_{jj} \updownarrows \not\Rightarrow \lvert x_j\rvert \updownarrows$ and $(x_j \not\approx 0 \vee IC(0.5) \not\gg TT(x_j))$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{term}} \leftarrow$ true. (No impact, desirably under-constrained)
(c)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}. (No inference, other factors dominate)
(d)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow R_{jj} \downarrow$. (over-constrained)
Case 4:	$\lvert x_j^{k-1}\rvert \leq N_j$	$\lvert x_j^{k}\rvert > N_j$	
(a)	$R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow f_j^{\text{bu}} \leftarrow \text{true},\ c_j \mathrel{+}= 1,\ R_{jj} \uparrow$. (Initiate boundary tuning)
(b)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow$ Revert R_{jj}. (No inference, other factors dominate)
Case 5:	$\lvert x_j^{k-1}\rvert > N_j$	$\lvert x_j^{k}\rvert \leq N_j$	
(a)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow f_j^{\text{bu}} \leftarrow \text{true},\ c_j \mathrel{+}= 1,\ R_{jj} \downarrow$. (Initiate boundary tuning)
(b)	$R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}. (No inference, other factors dominate)
Case 6:	$\lvert x_j^{k-1}\rvert > N_j$	$\lvert x_j^{k}\rvert > N_j$	
(a)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow R_{jj} \uparrow$. (Problematically under-constrained)
(b)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}. (No inference, other factors dominate)

Table 6.3: LQR input-costs: Boundary Tuning Step, when $f_j^{\text{term}} = \text{false}$ and $f_j^{\text{bu}} = \text{true}$

	R_{jj}^{k-1}	R_{jj}^{k}	$R_{jj}^{k+1} =?$
	x_j^{k-1}	x_j^{k}	
Invariant:	$f_j^{\text{term}} = \text{false} \quad \Rightarrow \quad R_{jj}^{k-1} \neq R_{jj}^{k}$		
Case 7:	$\lvert x_j^{k-1}\rvert \leq N_j$	$\lvert x_j^{k}\rvert \leq N_j$	
(a)	$R_{jj} \uparrow\downarrow \not\Rightarrow \lvert x_j\rvert \uparrow\downarrow$ and $(x_j \approx 0 \wedge IC(0.5) \gg TT(x_j))$		$\Rightarrow R_{jj} \downarrow$. (No impact, over-constrained)
(b)	$R_{jj} \uparrow\downarrow \not\Rightarrow \lvert x_j\rvert \uparrow\downarrow$ and $(x_j \not\approx 0 \vee IC(0.5) \not\gg TT(x_j))$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{bu}} \leftarrow$ false, $f_j^{\text{term}} \leftarrow$ true. (No impact, desirably under-constrained)
(c)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{bu}} \leftarrow$ false. (No inference, other factors dominate)
(d)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow R_{jj} \downarrow$. (Continue till boundary crossing)
Case 8:	$\lvert x_j^{k-1}\rvert \leq N_j$	$\lvert x_j^{k}\rvert > N_j$	
(a)	$R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow c_j \mathrel{+}= 1,\ R_{jj} \uparrow$. (Boundary crossed, reverse with finer step)
(b)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{bu}} \leftarrow$ false. (No inference, other factors dominate)
Case 9:	$\lvert x_j^{k-1}\rvert > N_j$	$\lvert x_j^{k}\rvert \leq N_j$	
(a)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow c_j \mathrel{+}= 1,\ R_{jj} \downarrow$. (Boundary crossed, reverse with finer step)
(b)	$R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{bu}} \leftarrow$ false. (No inference, other factors dominate)
Case 10:	$\lvert x_j^{k-1}\rvert > N_j$	$\lvert x_j^{k}\rvert > N_j$	
(a)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \downarrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \uparrow$		$\Rightarrow R_{jj} \uparrow$. (Continue till boundary crossing)
(b)	$R_{jj} \uparrow \Rightarrow \lvert x_j\rvert \uparrow$ or, $R_{jj} \downarrow \Rightarrow \lvert x_j\rvert \downarrow$		$\Rightarrow$ Revert R_{jj}, $f_j^{\text{bu}} \leftarrow$ false. (No inference, other factors dominate)

Table 6.4: LQR input-costs: Hold Step, when $f_j^{\text{term}} = \text{true}$

$\vec{s}_t \ \rightarrow$

R_{jj}^0	R_{jj}^1	$\dots$	R_{jj}^{k-1}	R_{jj}^k	$R_{jj}^{k+1} = ?$
x_j^0	x_j^1	$\dots$	x_j^{k-1}	x_j^k	

Invariant:	$f_j^{\text{term}} = \text{true} \quad \Rightarrow$	$f_j^{\text{bu}} = \text{false}$
Case 11:	$R_{jj}^{k-1} \neq R_{jj}^k$	$\Rightarrow R_{jj}^{k+1} \leftarrow R_{jj}^k$ (Previous step had revert action, start hold)
Case 12:	$R_{jj}^{k-1} = R_{jj}^k$	
(a)	$\not\Rightarrow \ \lvert x_j \rvert \uparrow\downarrow$	$\Rightarrow R_{jj}^{k+1} \leftarrow R_{jj}^k$ (Continue hold)
(b)	$\Rightarrow \ \lvert x_j \rvert \uparrow\downarrow$	$\Rightarrow R_{jj} \downarrow,\ f_j^{\text{term}} \leftarrow \text{false}$ (Change is due to other factors, restart refinement to possibly re-correct x_j)

6.5.2 Introducing Adaptive-Integral Control into LQR to Compensate for Model Approximation

The LQR regulator design strategy provides provably optimal control inputs (with optimality defined as minimizing cost function J) when the linear dynamical model of the system is an accurate reflection of true system dynamics. However, we need to drive LQR using an estimated dynamical model. The model must be estimated using very limited history data, since immersive applications are expected to exhibit rapidly time-varying characteristics. Further, the underlying behavior is possibly highly non-linear, though monotonicity of the behavior has been guaranteed by the programmer. Model estimation via LLSE can at most produce a good linear approximation of the underlying behavior. In particular, it is crucial that model estimation gets the signs of the dominant x_j-y_i relationships correct, otherwise applying a linear regulator like LQR will force a bad control input to become a progressively worse control input at each subsequent time step.

Here we make the assertion that so long as the signs of the dominant coefficients are estimated correctly, a modified version of LQR that we propose can tolerate significant errors in the magnitudes of the estimated coefficients. Our modification to LQR introduces adaptive-integral control to each of the output variables to account for the observed discrepancy between the linear model $\mathcal{M}_{\mathtt{LLSE}}$ estimated for the application and the true application response model $\mathcal{M}_{\mathtt{App}}$.

The adaptive control takes the following form: instead of updating state $\vec{s}_t$ with the observed output $\vec{y}_{t-1}$ as per Eq 32, we now update $\vec{s}_t$ with a *scaled version* of the output, $\vec{y}^{\,\mathtt{sc}}_{t-1}$.

The scaled output is determined as follows:

$$\vec{y}^{\,\mathtt{sc}}_{t-1} = \vec{\tilde{y}} + \vec{\beta} \circ \varepsilon \vec{y}_{t-1}, \tag{68}$$

where, $\vec{\tilde{y}} = [\tilde{y}_1, \tilde{y}_2, \cdots, \tilde{y}_m]^{\mathtt{T}}$ are the objectives for the corresponding outputs $y_1, \cdots y_m$. The *output error* observed at time $t-1$ is denoted by $\varepsilon \vec{y}_{t-1} = \vec{y}_{t-1} - \vec{\tilde{y}}$. $\vec{\beta}$ captures the individual *scaling factors* for the observed error in each y_i, with the notation $\vec{c} = \vec{a} \circ \vec{b}$ representing the *Hadamard Schur product*: $c_i = a_i.b_i$, i.e., an element-wise multiplication. Hence, the adaptive policy consists of determining appropriate values for the β_i's based on the observed tracking error for the y_i's. Note that $\beta_i = 1.0$ leaves unmodified the LQR state update.

The dynamic scaling of the tracking error with β_i allows LQR to drive $\varepsilon y_{i|t}$ to zero, when LQR by itself is unable to do so. An instance of the univariate controller from Chapter 4 is used for determining each β_i in each time-step t. For the univariate controller to be applicable in this setting, we have to demonstrate that each tracking error $\varepsilon y_{i|t-1}$ varies monotonically with the changes applied by the regulator to every control input x_j in the next time step t. Appendix A.3 demonstrates that this property indeed holds.

6.6 Bounding Runtime Overhead

The user specifies a per-frame budget b as a limit on the controller runtime overhead. The controller has a *budget objective* to keep its *average* per-frame overhead less than or equal to the time duration b.

The controller maintains a *debt* metric that *accumulates* by how much the past frames exceeded the budget b. The debt decreases after a frame when the controller has smaller overhead in that frame than b. Every frame a *usable budget* b_{usable} is computed as the difference between b and the accumulated debt.

b_{usable} limits the invocation of replacement model estimation (that is, LLSE for estimating $\mathcal{M}'$) and regulator re-design (that is, iterative refinement of an existing $\mathcal{C}$ using LQR). If $b_{\text{usable}} < 0$, only the work absolutely critical to the controller's QoS performance will be invoked. The critical work consists of model estimation if $\mathcal{M} = \phi$ and the initial design of the regulator $\mathcal{C}$ from a new $\mathcal{M}$ (that is, when $\mathcal{C} = \phi$).

The non-critical work of estimating the alternative model $\mathcal{M}'$ and of iterative regulator refinement is done only if $b_{\text{usable}} \geq 0$. The controller queries the system time before running each iteration of non-critical work in a frame, and executes that iteration of work only if the b_{usable} budget has not already been exhausted. *Basic work* involving collection of metrics and state update is not limited by the budget.

When b is greater than the time required for the per-frame basic work, the controller achieves its objective of keeping the average per-frame overhead $\leq b$. Note that the overhead may exceed b on the individual frames that must do the critical work, or when non-critical work gets started before the deadline set by b_{usable} but only completes after the deadline. When $b_{\text{usable}} < 0$, the controller will suppress all non-critical work for a series of frames until $b_{\text{usable}} \geq 0$ again holds, thereby achieving the budget objective.

CHAPTER VII

MULTI-VARIATE QOS CONTROL: EXPERIMENTAL EVALUATION

We apply the multi-variate controller to full applications available as open-source. The applications are representative of computer vision and video encoding. Each application is highly compute intensive and has a requirement to maintain a smooth frame-rate.

The applications, as made available, use fixed values for parameters in their key algorithms. The QoS behavior exhibited by each application changes over a wide range as the application is re-executed with different values for the algorithm parameters. We apply the QoS controller to these applications to dynamically tune the values of the algorithm parameters after every frame. We evaluate the QoS performance of an application for every possible fixed setting of the algorithm parameters ("fixed cases") and with the controller ("controller case"). Each case (the use of the controller, or of a particular fixed setting) is evaluated over a range of target values for the QoS objective (e.g., a range of desired frame-times). For each case, we compute the average QoS over the objective range. We present summarized results that compare the average QoS of the controller case against the fixed case with the best average QoS ("best fixed case"). We also present detailed results for each case showing the QoS obtained for each value of the QoS objective.

The experimental evaluation seeks to establish the following.

1. Our controller can significantly improve the frame-time QoS when one or more control parameters are tuned by the controller.

2. Allowing the controller to simultaneously tune multiple parameters (nX) typically produces better QoS than tuning only one parameter ($1X$). While expected, the results establish that the controller can explore a much larger higher-dimensional input space with low overhead and still find "good" parameter values with high probability.

3. We construct application-specific "accuracy of results" metrics, and demonstrate that the controller can simultaneously optimize for frame-time and accuracy QoS (nX–$2Y$).

4. Finally, with application case studies, we illustrate the process of applying the controller to applications. In particular, we illustrate how programmers may verify if their application sufficiently satisfies the required domain assumptions, and how programmers may easily adapt an application implemented with fixed parameter settings and without explicit QoS metrics for dynamic tuning with the controller.

7.1 Applications

mpeg2enc. This is an MPEG2 video encoder from the MediaBench II video benchmark suite [70]. The motion-estimation algorithm is known for being the most compute-intensive part of this application. The *search-window-size* parameters determine how much effort the algorithm expends in trying to find a matching macroblock for the current macroblock being encoded. In general, a larger search window typically allows for less lossy compression by making it more likely that a matching macroblock will be found, while incurring a correspondingly higher frame-time. We use the frame time as the application QoS metric y and the search-window-size parameters as the control inputs $\vec{x}$ used by the controller to tune y. The $1X$ evaluation uses a single search-window-size for the horizontal and vertical searches in both the forward (next frame) and backward (previous frame) search directions on any given frame. The $2X$ evaluation uses two control parameters to allow independent setting of the horizontal and vertical search-window-sizes. The $4X$ evaluation uses four control parameters to independently set the horizontal and vertical search-window sizes in the forward and backward directions.

ferns. This is a computer vision application [75]. Ferns uses keypoint-detection techniques to detect and track the presence of a reference planar pattern in a video in a 3D translation-, scaling- and rotation-invariant manner. For example, we use a flat picture of a mousepad as the reference planar pattern and the test video consists of a person moving and rotating the mousepad in 3D space in a room. We use our controller to maintain a desired frame-time y while varying the *number of iterations* executed by a classification algorithm in the program (represented by x). Once the target pattern is detected in a frame, ferns is capable of tracking the pattern in subsequent frames at a lower computational cost. However, for our purpose of measuring the impact of the controller on the ferns detection algorithm, we disable tracking and perform detection every frame. We evaluate two QoS objectives — the frame-time and an "accuracy of results" metric defined by us. The accuracy metric estimates how accurate is the number of keypoints detected by ferns in a frame. As a heuristic, we designate every fourth frame as a "reference frame" for which x is set to the maximum number of iterations. The measured number of keypoints detected is treated as a reference to compute accuracy for the next three frames. We consider two factors in defining a ferns-specific accuracy metric — *i)* ferns, as written, treats 10 keypoints as the detection threshold for the target pattern, and *ii)* the detection algorithm employs randomization and exhibits large frame-to-frame variations in the number of keypoints detected, though the detection pattern is clear when viewed over multiple frames. Given these factors, we define the accuracy metric as follows: $\text{accuracy} = 1 - \frac{|\text{ref-keypoints} - \text{detected-keypoints}|}{\max\{\text{ref-keypoints, detected-keypoints}\}}$, and ref-keypoints is kept at least 5 to avoid a possible divide-by-zero error, as there is a possibility of detecting no keypoints in a frame due to the classification algorithm's inherent randomness. Therefore, our definition of the accuracy metric may incorrectly penalize or help the accuracy of subsequent frames whenever a reference frame has a large error in ref-keypoints. However, the accuracy metric is still useful as an approximate estimator of accuracy, and helps us illustrate how the accuracy of computer vision applications may be tuned dynamically. We will evaluate $1X$ against $1Y$, with y being frame-time and then accuracy, and finally $1X$ against $2Y$ (the joint QoS optimization of frame-time and accuracy).

rtftr. This is a real-time face-detection and tracking application [76]. The application detects the presence of one or more faces in every frame of the video and can track them across video frames. The detection algorithm extracts rectangular blocks of different sizes from the video frame, to explore the possible occurrence of a face at any scale in the frame. We apply the controller to control detection frame-time y by varying two parameters that control the granularity at which block extraction is performed — a *scalefactor* parameter (*sf*) that determines the next scaled-down box size to try, and a *minrect* parameter (*mr*) that determines the smallest block size to consider. Use of a larger scalefactor speeds up the detection algorithm, as fewer scales are considered from the largest till minrect. However, use of a larger scalefactor increases the risk of missing the occurrence of a face that could only be detected at a particular block size. A smaller minrect allows correspondingly smaller faces to be detected, but increases compute time as the number of smaller blocks in an image grows exponentially. At every frame, the application produces a count of the number of faces detected. As a heuristic, we treat every fourth frame as a reference frame, where the corresponding mr or sf or both parameters take the most compute-intensive values and the resulting number of faces detected is stored as the reference for the next three frames. We define an rtftr-specific accuracy metric, accuracy $= 1 - \frac{|\text{detected-face-count} - \text{ref-face-count}|}{\max\{\text{detected-face-count, ref-face-count}\}}$. We will evaluate the use of control parameters mr and sf individually and then together against one and then two QoS objectives — frame-time and accuracy ($1X-1Y$, $2X-1Y$ and $2X-2Y$ evaluations).

x264. We use the x264 benchmark from the PARSEC 2.1 benchmark suite [77]. x264 implements the video encoder from the H.264/AVC standard [78]. The performance of x264 scales with number of cores. Additionally, x264 utilizes a large number of encoding parameters and choice of algorithms for various stages that are intended to scale the encoding characteristics of the encoder based on a static configuration. Out of these we choose the following control parameters for dynamic tuning — *i)* the number of cores (which impacts frame-time but not fidelity), *ii)* the choice of sub-pixel interpolation algorithm (e.g., fullpel, SAD mode decision, SATD mode decision, etc.), and *iii)* motion estimation search window size. The choice of subpel algorithm and the motion estimation search window parameters impact both the frame-time and the fidelity. The chosen control parameters satisfy the monotonicity assumption against the explicit QoS goal of frame-time and the implicit

QoS goal of fidelity. The x264 benchmark as a whole satisfies the domain assumptions for immersive applications.

bodytrack. We use the bodytrack benchmark from the PARSEC 2.1 benchmark suite [77]. bodytrack tracks the pose of a user in a video (e.g., recognize that a hand is raised and determine its position). bodytrack uses particle filters [79, 80] for pose estimation. bodytrack's performance scales with the number of cores. Additionally, we demonstrate that the number of particles can be dynamically varied to tradeoff pose-detection accuracy versus achieving a desired frame-time.

7.2 Experimental Setup

Table 7.1 summarizes the scalable algorithms, the control parameters and the QoS objectives used for each application. $\delta = 20\%$ is used for the QoS objectives.

Table 7.1: Application scalable algorithms, tunable parameters and QoS objectives.

Benchmark	Scalable Algorithms	X	$Y(\delta = 20\%)$
mpeg2enc	Motion estimation search size: - **H**orizontal / **V**ertical - **F**orward / **B**ackward	**1X** = H / V / F / B **2X** = H F / B, V F / B **4X** = H F, H B, V F, V B	**fr** = frame-time
ferns	Scale and rotation invariant object detection using keypoints	**kp** : max number of iterations for detecting keypoints	**fr** = frame-time **ac** = accuracy
rtftr	Generate candidate rectangles that may frame faces - minimum rectangle size - scale up for next rectangle size	**mr** : min rectangle size **sf** : scale factor	**fr** = frame-time **ac** = accuracy
x264	Parallel implementation, multiple algorithms to choose from for fast vs accurate sub-pixel interpolation, motion estimation search size	**nc** : num cores **sp** : sub-pel algo choice **me** : motion est search size	**fr** = frame-time
bodytrack	Parallel implementation, particle-filter based pose search	**nc** : num cores **np** : num particles	**fr** = frame-time

7.3 Results

7.3.1 Frame-time QoS

Figure 7.1 compares the average frame-time `SR` (measure of QoS) of the controller against the best fixed case for each of the applications. Each application is evaluated on multiple data-sets which differ in content and/or factors such as video resolution. `mpeg2enc` is evaluated on the standard `dolbycanyon` and `dolbycity` video sequences, with a resolution of 320×240 for the former, and 320×240 and 640×480 for the latter. Similarly, `rtftr` and `ferns` use their corresponding test video sequences at two different resolutions.

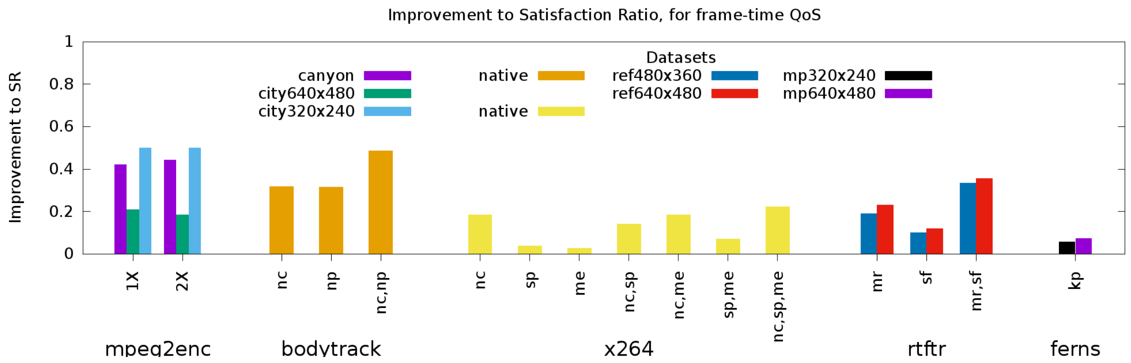


Figure 7.1: Comparison of average frame-time QoS between controller case and best fixed case: across benchmarks, data sets and choice of control parameters.

Each application was executed over a range of frame-time objectives $\tilde{y} \pm \delta$, where $\tilde{y}$ is varied over a range. An example objective would be to keep frame-time in the window $0.02 \pm 20\%$ seconds. The range of frame-time objectives $\tilde{y}$ simulates the execution of the application on a variety of platforms with differing processing capabilities, as well as a genuine requirement for changing the target frame-time if the application has varied use-cases. Figure 7.1 shows several benchmarks and data sets where the controller delivers *average* `SR` improvements in the range of ≈ 0.2 to 0.5, a significant improvement in QoS compared to keeping $\vec{x}$ fixed at the best possible value. We do not show results for `mpeg2enc` with $4X$ as that evaluation would require an inordinate number of runs of the application corresponding to the 8^4 possible fixed cases, requiring weeks of machine time.

7.3.2 Accuracy QoS

Figure 7.2 summarizes the adverse impact on the accuracy due to the use of the controller. We have defined the accuracy metrics for `ferns` and `rtftr` with respect to setting the corresponding control parameters at their most compute-intensive settings. Therefore, the controller will always do worse compared to the most-compute intensive fixed case. The best we can hope for is that the controller will rapidly determine that the control parameters must be set to their most compute-intensive values in order to maximize QoS. For `ferns` we see a minimal adverse impact. For `rtftr` we observe a minimal adverse impact when only one control parameter (either `mr` or `sr`) is exercised by the controller. When both control parameters are tuned, there is a larger adverse impact to `SR` in the range of -0.10 to -0.20. With both control parameters, the accuracy metric falls very sharply if either control parameter deviates from its most compute-intensive setting. Despite the fact that the $x-y$ response for `rtftr` satisfies the monotonicity requirement, the large non-linearity in the $x-y$ response implies that the controller essentially needs to sample the input space at or near the most compute intensive settings in order to drive the inputs to those values. Otherwise, the estimated model of the $x-y$ response will fail to sufficiently incorporate the disproportionate impact of the most compute-intensive settings on accuracy. The single parameter cases have a 1 in 5 chance of randomly sampling the most compute-intensive setting, compared to only a 1 in 5^2 chance when both parameters are used, causing the larger deterioration in accuracy `SR` when both parameter are used. However, the adaptive scaling of the feedback error to the LQR regulator (Section 6.5.2) helps mitigate the full adverse impact of the response non-linearity.

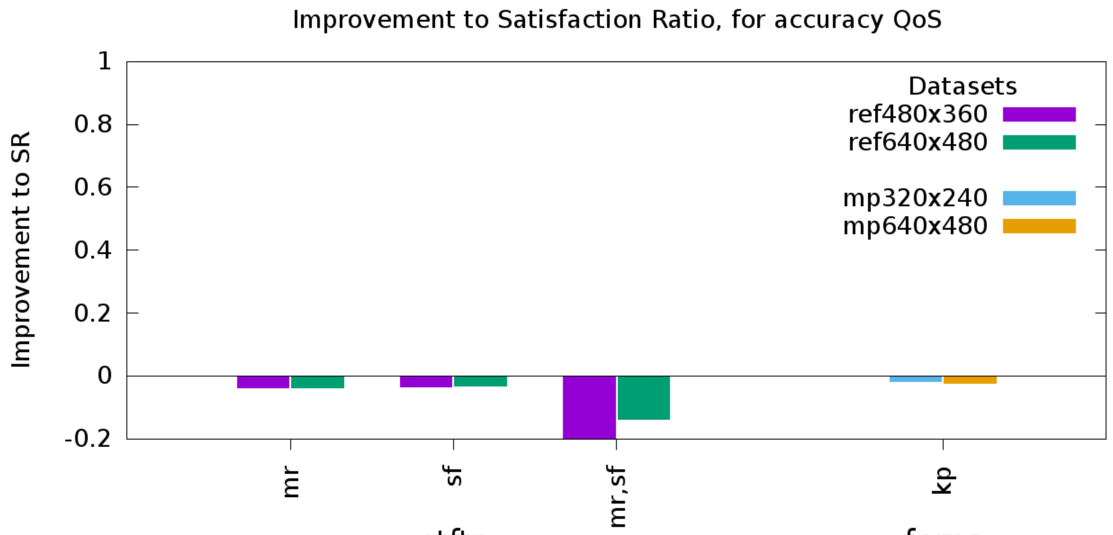


Figure 7.2: Comparison of average accuracy QoS between controller case and best fixed case: across benchmarks, data sets and choice of control parameters.

7.3.3 Joint Accuracy and Frame-time QoS

Figure 7.3 summarizes the benefit to the joint frame-rate and accuracy `SR` with the use of the controller. The frame-rate and accuracy aspects are typically in opposition — the accuracy is maximized when the most compute-intensive settings are chosen for the control parameters, while the frame-time may require less compute-intensive setting of the control parameters. The controller will balance the frame-rate and accuracy aspects when tuning the application to the joint QoS objective. We observe modest `SR` improvements for `ferns`, with 0.05 to 0.15 improvements in `SR` for `rtftr`. As expected, the improvements are not as large as when the controller optimized only for frame-time QoS (Figure 7.1). When optimizing only for frame-time, no fixed setting would work well across the range of frame-time objectives, giving the controller a huge advantage. However, a single fixed setting by our definition of the accuracy metric always produces the best accuracy regardless of the frame-time objective, limiting the improvements in the joint accuracy and frame-time QoS.

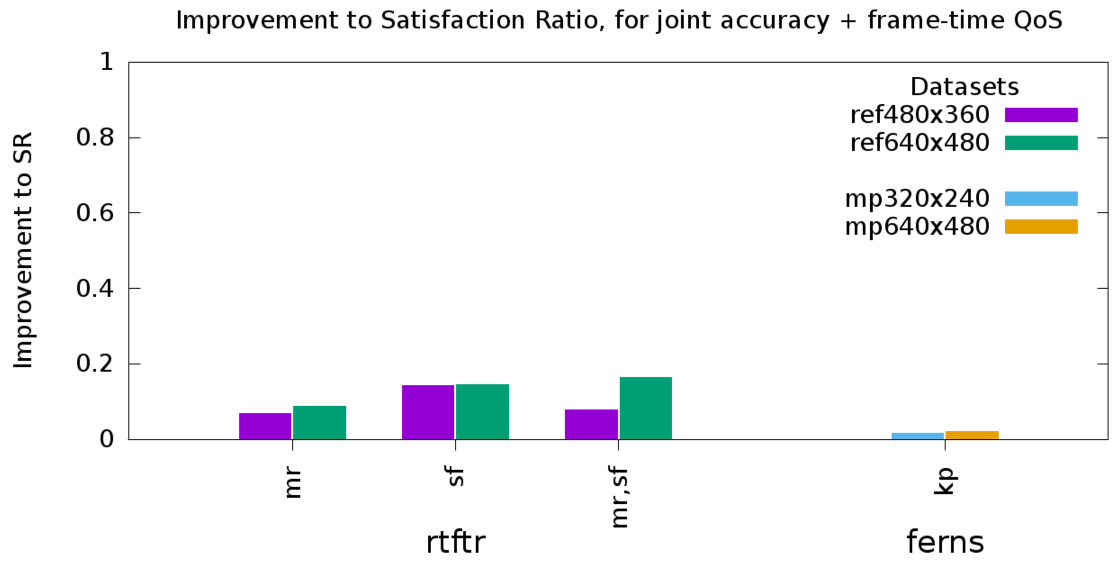


Figure 7.3: Comparison of average joint accuracy and frame-time QoS between controller case and best fixed case: across benchmarks, data sets and choice of control parameters.

7.3.4 Benefit of Multiple Control Parameters

Figure 7.4 and Figure 7.5 contrast the `SR` when the controller tunes multiple control parameters (nX) compared to a single control parameter ($1X$). `ferns` is not shown as we use only a single control parameter in that application.

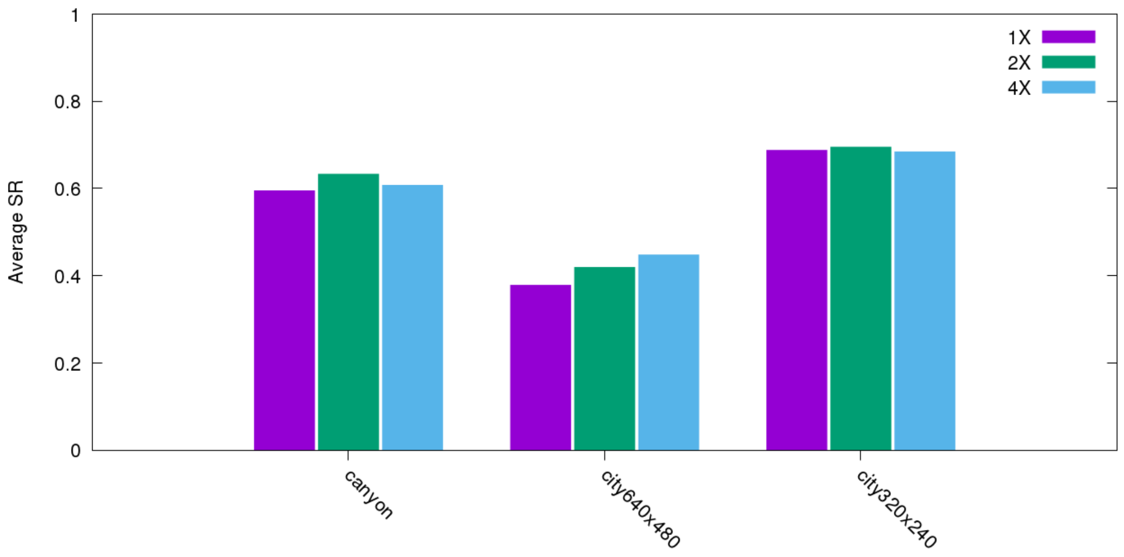


Figure 7.4: `mpeg2enc`: Impact of tuning multiple control parameters on frame-time QoS.

Figure 7.4 shows a modest improvement in the frame-time `SR` for `mpeg2enc` when the horizontal and vertical search-window-sizes are tuned independently as two separate parameters ($2X$), and when the vertical and horizontal search-window-size parameters for the forward and backward motion-prediction directions are also tuned independently ($4X$). One data set, `dolbycity` at 320×240 resolution, shows an insignificant reduction in `SR`, essentially indicating that the `SR` is unchanged in going from $1X$ to $4X$. While the results do not show large improvements for $2X$ and $4X$ over $1X$, they do demonstrate that the controller overhead stays limited in exploring the much larger $2X$ and $4X$ input-spaces (of size 8^2 and 8^4, respectively) compared to exploring the $1X$ input-space (consisting of 8 values). High controller overhead in terms of either more frames wasted on the PFE/CFE mechanisms (Section 6.2) due to the higher-dimensional input space, or by contributing to the frame-encoding time, would have manifested as a deterioration in `SR`.

Figure 7.5 shows the evaluation of `rtftr` for three objectives — a QoS objective of frame-time (`fr`), a QoS objective of accuracy (`ac`) and a combined QoS objective giving equal weight to frame-time and accuracy (`ac`, `fr`). For `fr` we see large improvements when control parameters `mr` and `sr` are tuned together compared to tuning only one of them. We see a significant reduction in `SR` in going to multiple parameters for the `ac` objective, for reasons explained in Section 7.3.2. For the joint QoS objective, combining the control parameters gives similar, somewhat worse, and somewhat improved QoS compared to the use of a single control parameter — a mixed result, overall suggesting no harm with the use of both parameters for the joint objective.

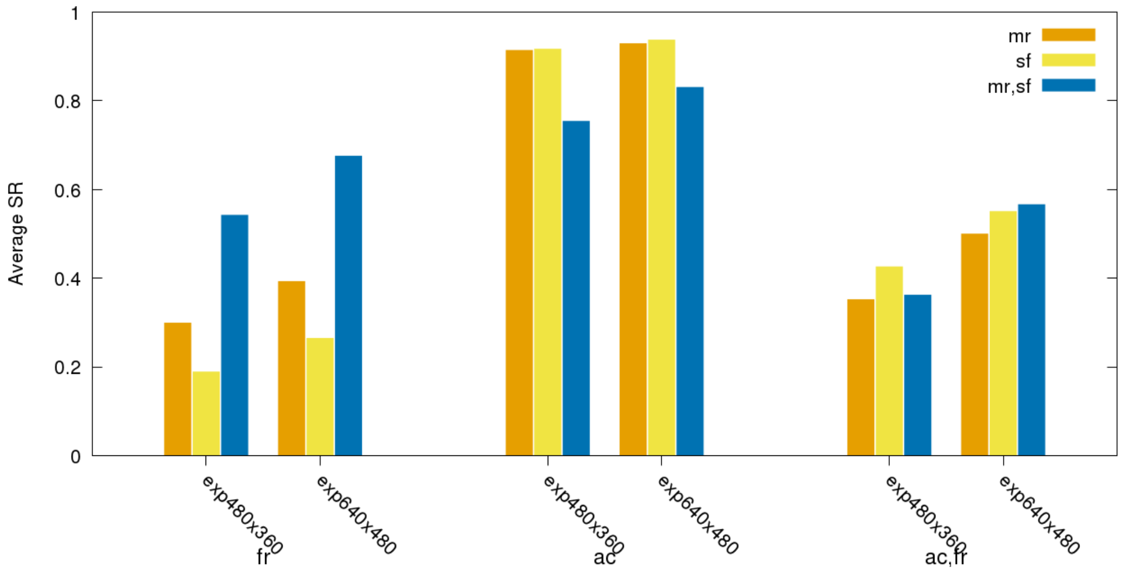


Figure 7.5: `rtftr`: Impact of tuning multiple control parameters (`mr` and `sf` individually and together). Evaluation performed for frame-time QoS (`fr`), accuracy QoS (`ac`) and the joint QoS (`ac`, `fr`).

7.4 `mpeg2enc` Detailed Evaluation

The search window size is determined by the following parameters.

- Forward horizontal size.
- Backward horizontal size.
- Forward vertical size.
- Backward vertical size.

All experiments for mpeg2enc are run on an Intel Q6600 CPU @2.40GHz desktop computer running Ubuntu Linux.

We run three sets of experiments. The first set uses a single parameter x to set all four of the search window parameters to equal values ($1X$ set). The second set uses x_1 to set identical values for the two horizontal sizes, and x_2 for the two vertical sizes ($2X$ set). The third set uses x_1, x_2, x_3 and x_4 to vary each of the search window sizes independently ($4X$ set). In general, it is expected that the use of multiple independent parameters will create more opportunity for the controller to find fine-tuned solutions to meet the application QoS (the desired frame-rate) than with just one parameter, producing a higher `SR` (satisfaction ratio). Our experimental results in Section 7.4 demonstrate that this is indeed the case.

Due to the I-P-B-B-P-B-B GoP pattern used by mpeg2enc, we use sliding window $W = 7$. Table 7.2 relates the range of control input values generated by the controller (-4 to 4) against the actual search window sizes used inside the application (the last value is repeated as by convention the controller needs an odd number of settings: integers $-N$ to N, for some N).

Next, we use the $1X$ set to verify that monotonicity indeed holds between search-window sizes and the encoding frame-time y. Figure 7.6 shows the spread of frame-time for increasing values of x on two test video sequences.

Figure 7.7 shows the `SR` produced for the $1X$ set on the two test video sequences. The x-axis shows increasing frame-time objectives $\tilde{y}$. We use a 20% tolerance for y, i.e., $\delta = 0.20\,\tilde{y}$. Two different test video sequences are used, *dolbycanyon* with 320×240 resolution and

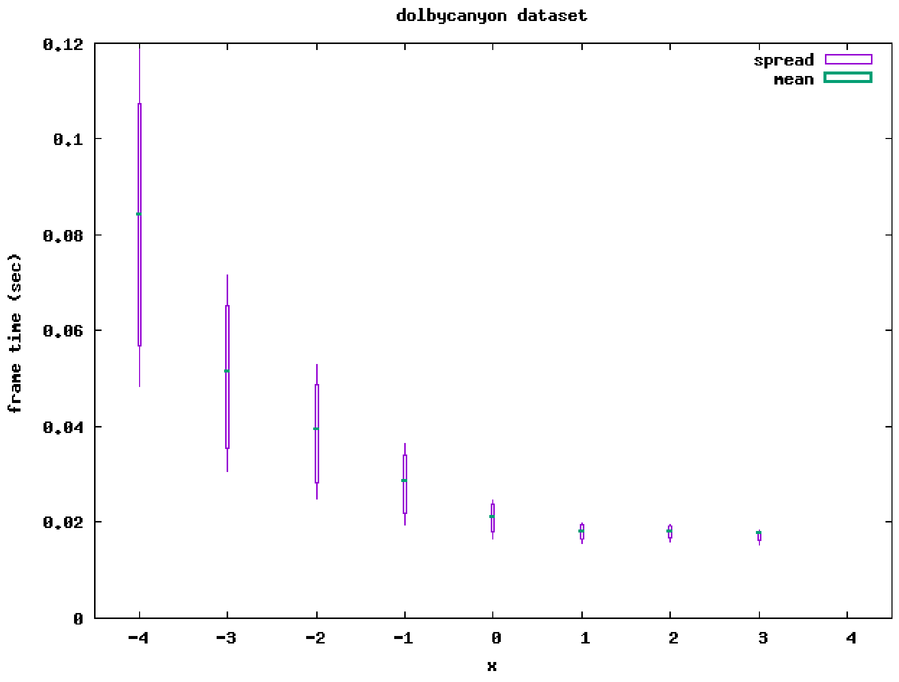
Table 7.2: Mapping of control values to search window sizes for mpeg2enc

x	search window size
-4	30
-3	20
-2	15
-1	10
0	5
1	2
2	1
3	0
4	0

*dolbycity*640 with 640×480 resolution, each consisting of approximately 1000 frames. The QoS performance of each application execution is measured as the `SR`. `SR` is compared between application executions that keep x fixed versus application executions that use the controller to dynamically tune x. The controller overhead is limited to 5% of the frame-time objective, i.e., budget $= 0.05\,\tilde{y}$. Notice that the controller executions significantly outperform the fixed cases at almost every $\tilde{y}$ (frame-time objective). Where a given fixed x execution outperforms the controller execution, it only does so marginally (*dolbycity*640 dataset graph), and only for a single $\tilde{y}$. For example, fixed $x = 2$ outperforms the controller at $\tilde{y} = 0.08 secs$, but significantly underperforms against the controller everywhere else. *This demonstrates that the controller is able to greatly enhance the QoS across a wide range of operating conditions (different frame-rates, or conversely execution on platforms with widely differing compute capabilities).*

Similarly, Figure 7.8 shows the `SR` achieved for the $2X$ set on the two test video sequences, comparing the QoS performance of the controller against the fixed x_1, x_2 cases. There are several characteristics worth noticing. First, note that x_1, x_2 combination fixed cases can outperform the x fixed cases from the $1X$ set. Second, notice that for the *dolbycanyon* dataset, the controller noticeably underperforms the best fixed cases for $\tilde{y} = 0.02$. Examining the output log of the application execution identifies the reason — the controller readily exhausts the $5\% \times 0.02$secs runtime overheads budget and is unable to perform LLSE and LQR on most frames when they are needed. In fact, looking at the output logs for $\tilde{y} = 0.02, 0.04$ on even the $1X$ set also reveals that the budget is exhausted on most frames, with the controller barely getting sufficient opportunity to perform LLSE and LQR to deliver high `SR`. With $\tilde{y} = 0.02$secs on the $2X$ set, the controller gets too constrained. The current implementation of the controller in C++ has not been optimized. We expect an opportunity for $> 10\times$ reduction in overheads once such optimization is attempted. Third, comparing the *dolbycity*640 dataset across the $1X$ and $2X$ sets shows mixed gains in QoS — $\tilde{y} = 0.08$secs performs better in the $2X$ set, taking advantage of the finer tuning possible with two control variables instead of one, as expected; for $\tilde{y} = 0.12$secs we get slightly poorer QoS performance with $2X$, due to the controller expending comparatively more frames in exploration and fewer in the exploitation of a good model compared to the $1X$ case.

Finally, Figure 7.9 shows the `SR` achieved for the $4X$ set on the two test video sequences. Due to the very large number of combinations of fixed cases x_1, x_2, x_3, x_4, we skip generating results for the fixed cases. Instead, we merely compare the results against the $2X$ and $1X$ sets. In particular, notice that the $4X$ set noticeably outperforms both the $2X$ and $1X$ sets on the *dolbycity*640 dataset — the gains from tuning four control inputs clearly outweigh the overheads of exploring and modeling application behavior in higher dimensions.



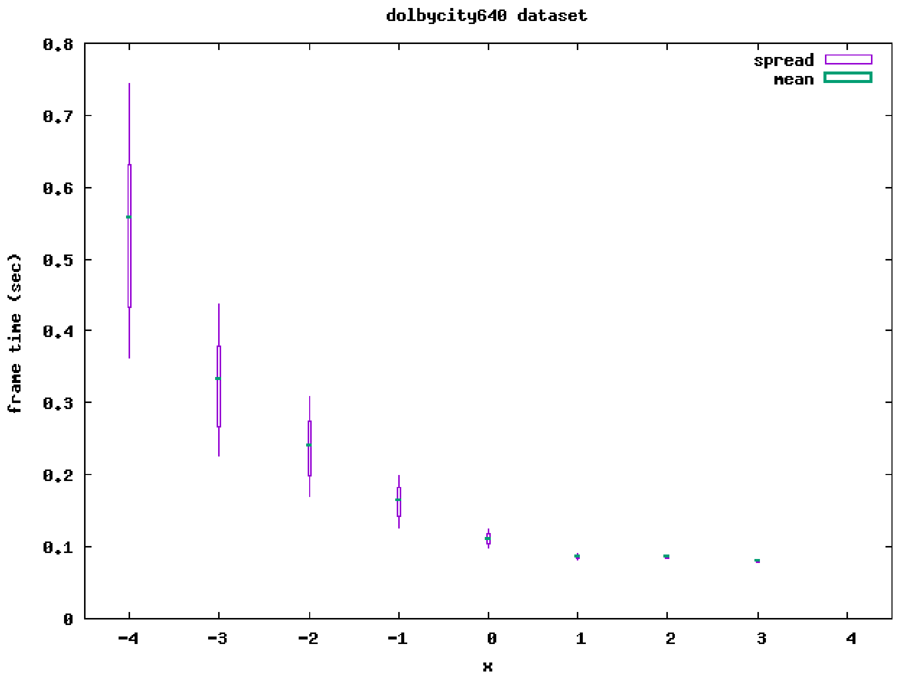
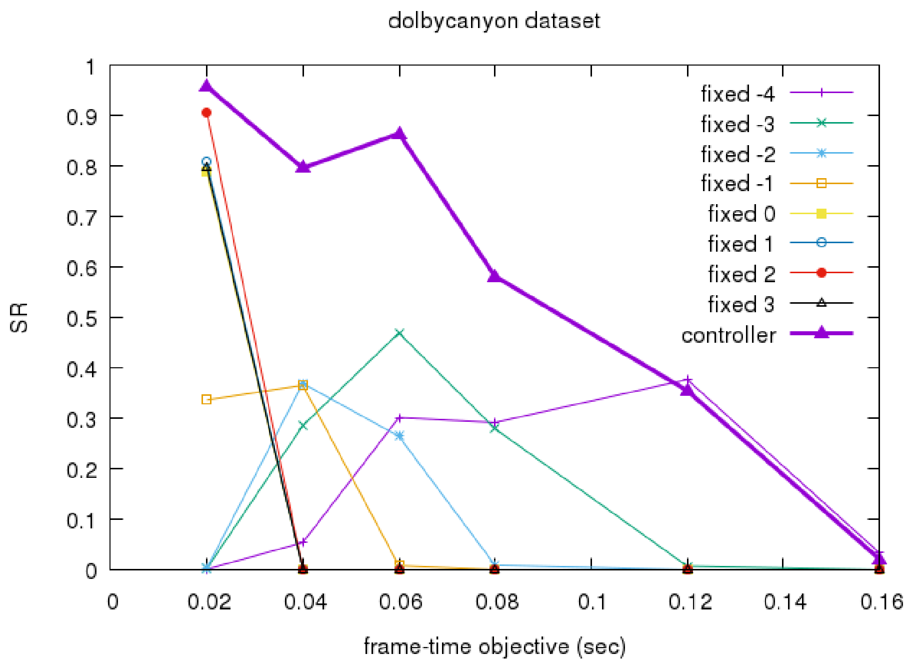


Figure 7.6: mpeg2enc: monotonic response between x and frame-time



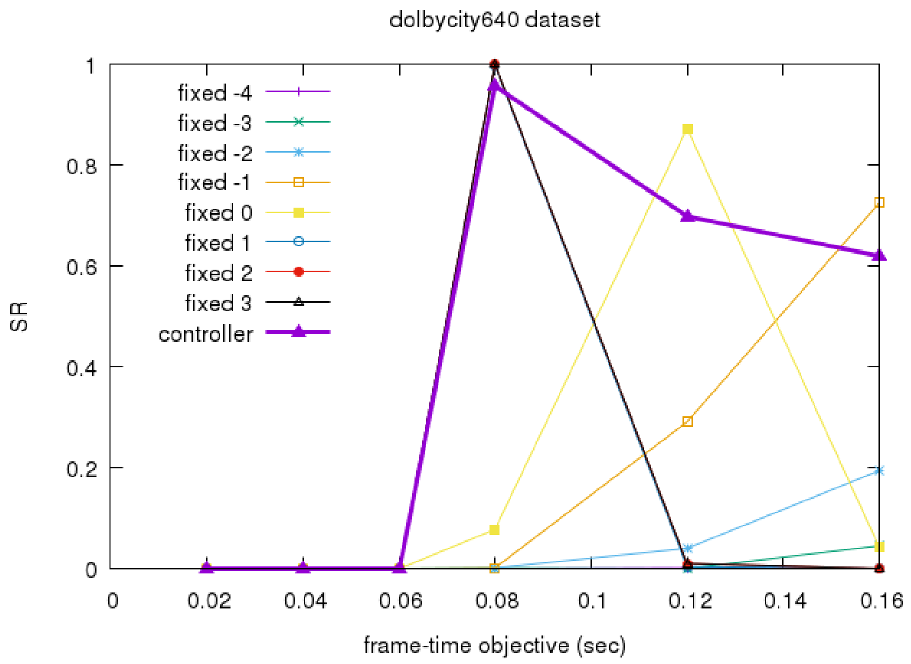
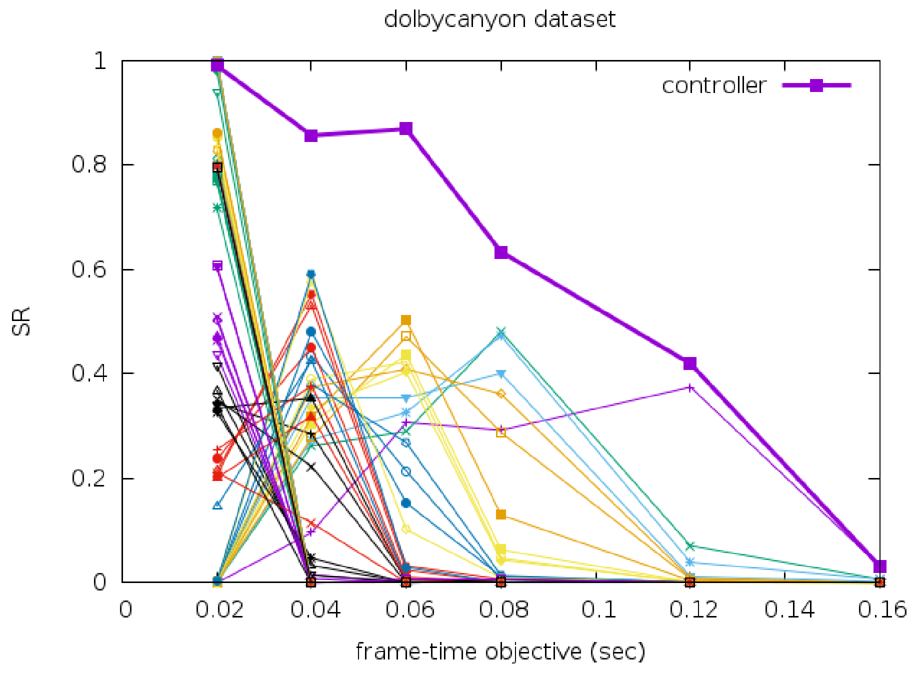


Figure 7.7: mpeg2enc $1X$: comparison of QoS performance for fixed x vs controller across varying frame-time objectives



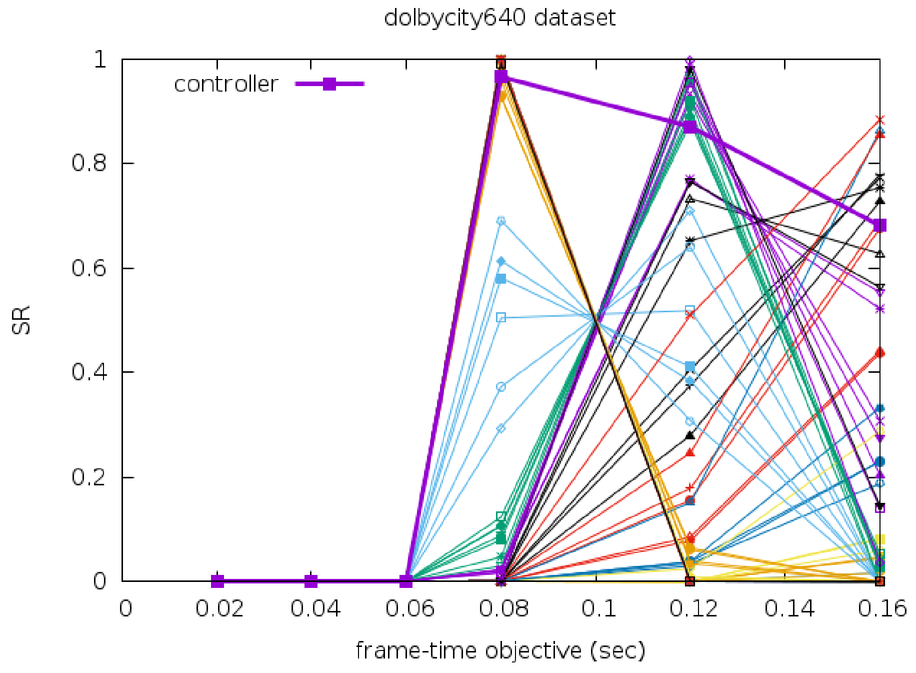
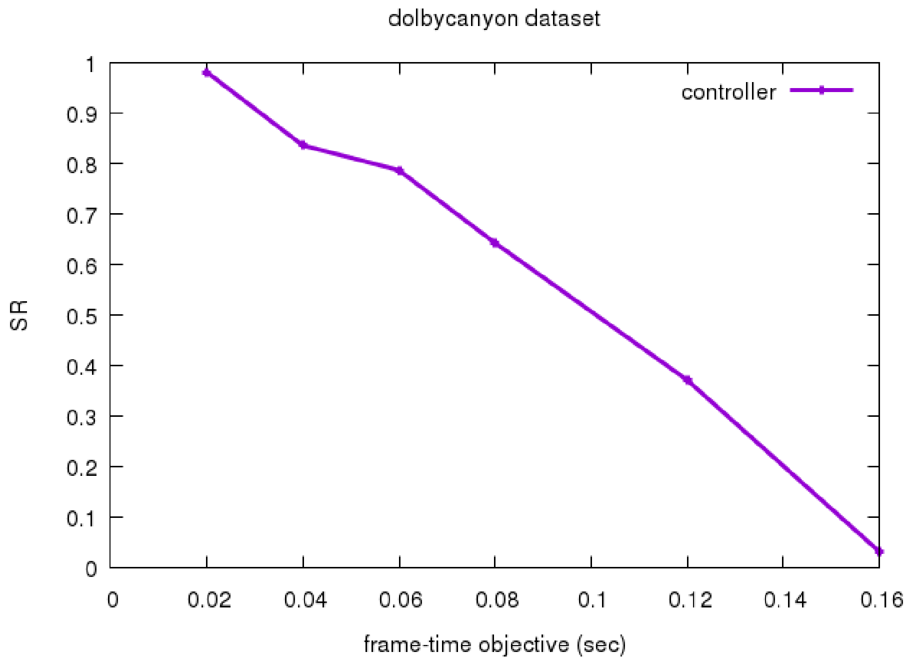


Figure 7.8: mpeg2enc $2X$: comparison of QoS performance for fixed x_1, x_2 vs controller across varying frame-time objectives



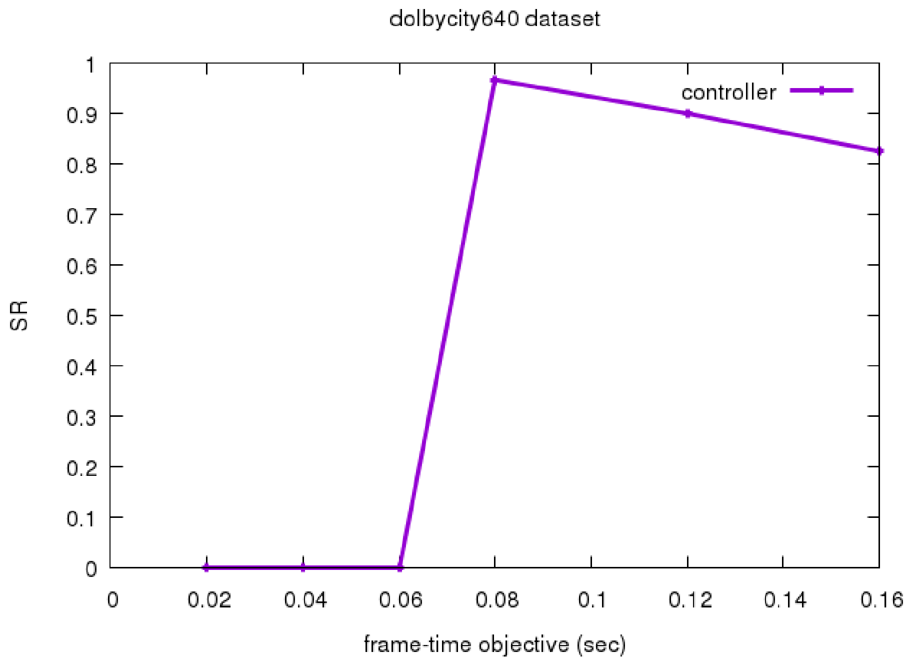


Figure 7.9: mpeg2enc $4X$: QoS performance of controller across varying frame-time objectives

7.5 `ferns` Detailed Evaluation

All experiments for ferns are run within an Ubuntu Linux virtual machine running on an Intel i7-3630QM CPU @2.40GHz running Ubuntu Linux.

We initially tried the following mapping, using $N = 6$, between the values of the control input x to the number-of-iterations used inside a ferns detector algorithm (Table 7.3).

Table 7.3: Mapping of control values to number-of-iterations for ferns

x	num iterations
-6	30
-5	25
-4	20
-3	18
-2	15
-1	13
0	10
1	8
2	5
3	4
4	2
5	1
6	1

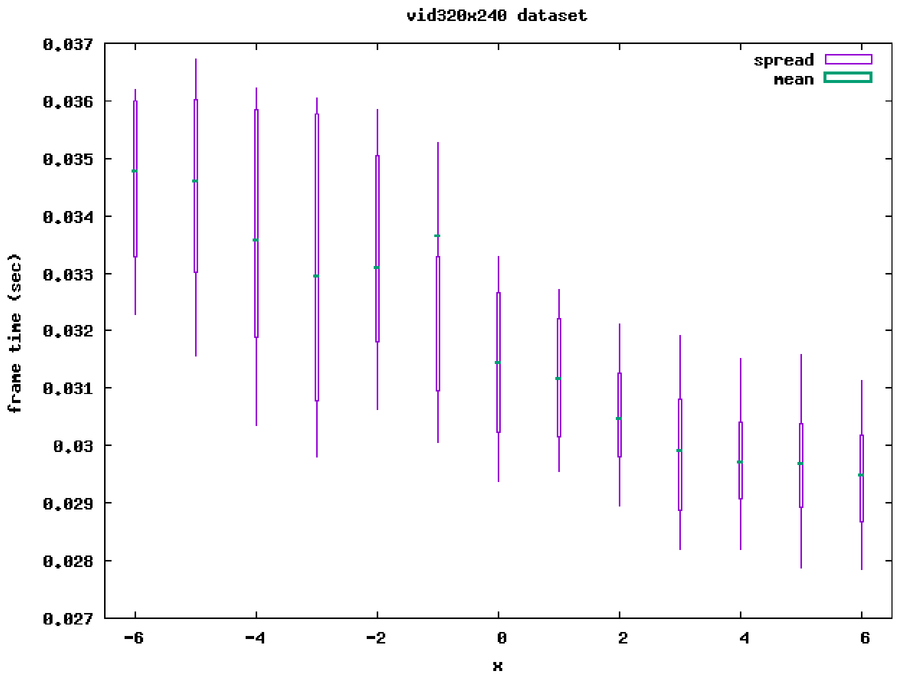
On two different test videos, Figure 7.10 shows that monotonicity between x and y (the detection frame-time) does not hold over several of the sub-ranges. Consequently, our controller exhibited erratic and frequently poor `SR` performance when the mapping in Table 7.3 was used. The two datasets are of the same test video scaled to different resolutions — $vid320 \times 240$ and $vid640 \times 480$. The two datasets potentially trigger different execution characteristics in the ferns algorithm, and definitely exhibit very different frame-times.

We trimmed Table 7.3 ($N = 6$) to Table 7.4 ($N = 2$) by dropping values that appeared to hinder monotonicity, and we finally observed distinctive x-y monotonicity, as shown in Figure 7.11. The spreads still overlap somewhat, but there is a much higher likelihood of monotonicity on any given frame of the application over every part of the input range. Note that domain experts and application programmers would have to establish for themselves if their application will show monotonic behavior on any intended use-case, and place the necessary restrictions inside their application to establish monotonicity before using our controller. With ferns we illustrate the process of establishing monotonicity, but do not make any claims about the additional use-cases over which this monotonicity will continue to hold.

Table 7.4: A better mapping of control values to number-of-iterations for ferns ($N = 2$)

x	num iterations
-2	30
-1	20
0	15
1	8
2	2

Using the mapping in Table 7.4, we get the `SR` performance shown in Figure 7.12 over increasing frame-time objectives $\tilde{y}$ with a 20% window. We use $W = 1$ since each frame independently performs detection and the results of the detection are often used for higher-level computer vision analysis. Figure 7.12 provides details on the frame-time QoS for ferns summarized in Figure 7.1. Figure 7.13 and Figure 7.14 provide similar details for the accuracy QoS and the combined QoS, respectively.



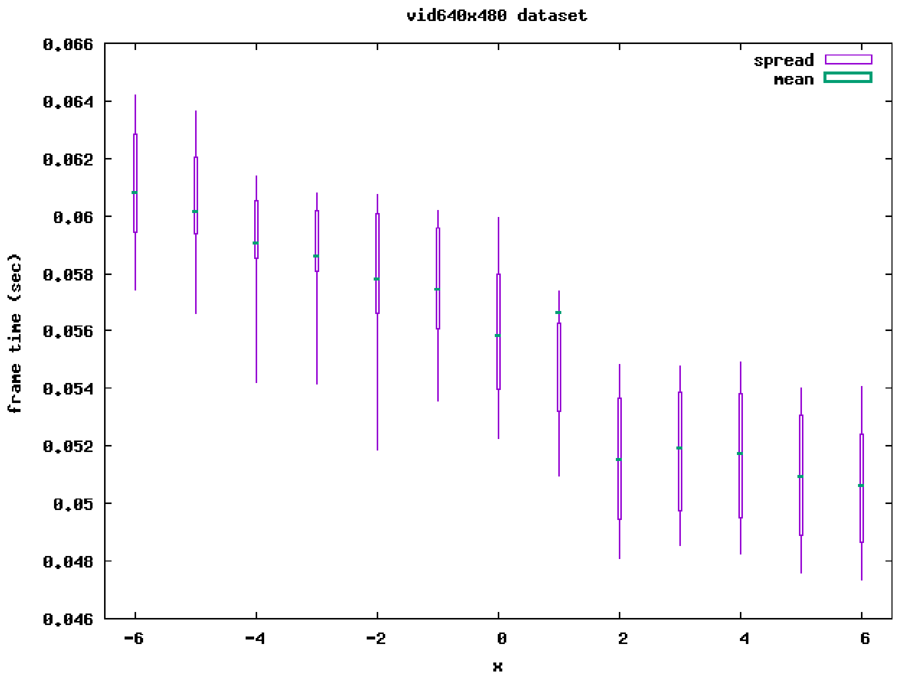
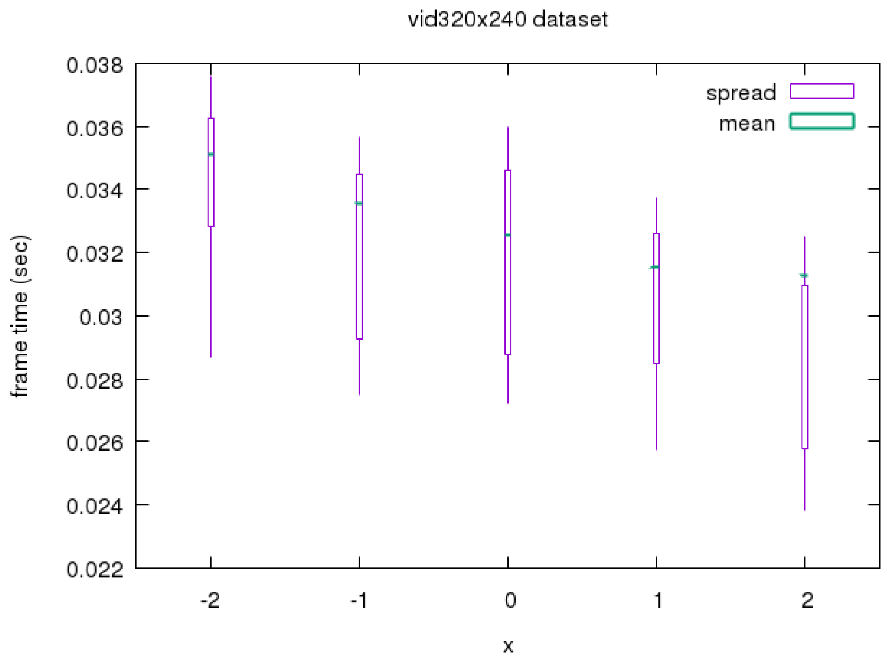


Figure 7.10: ferns $N = 6$: poor monotonic response between x and frame-time



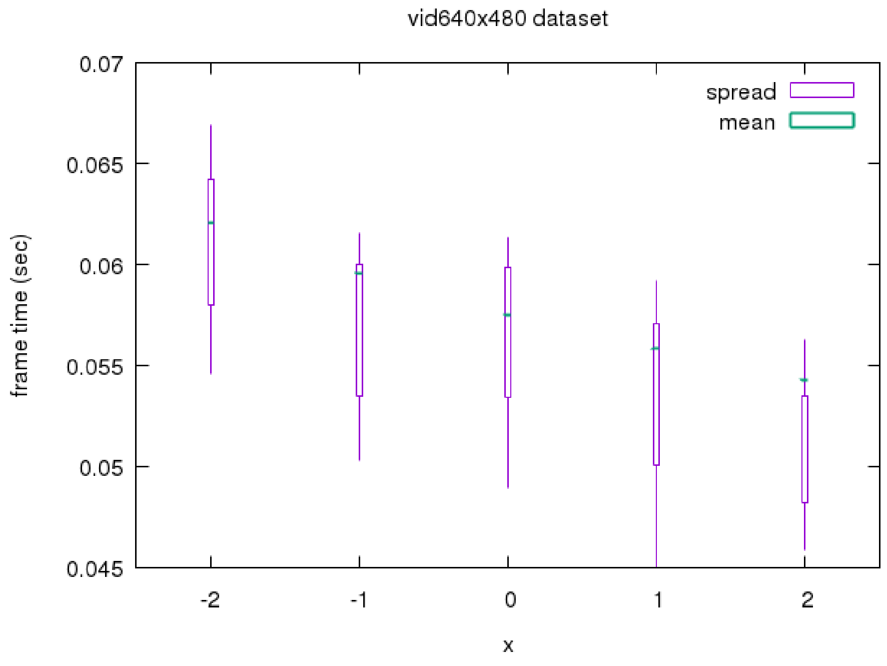
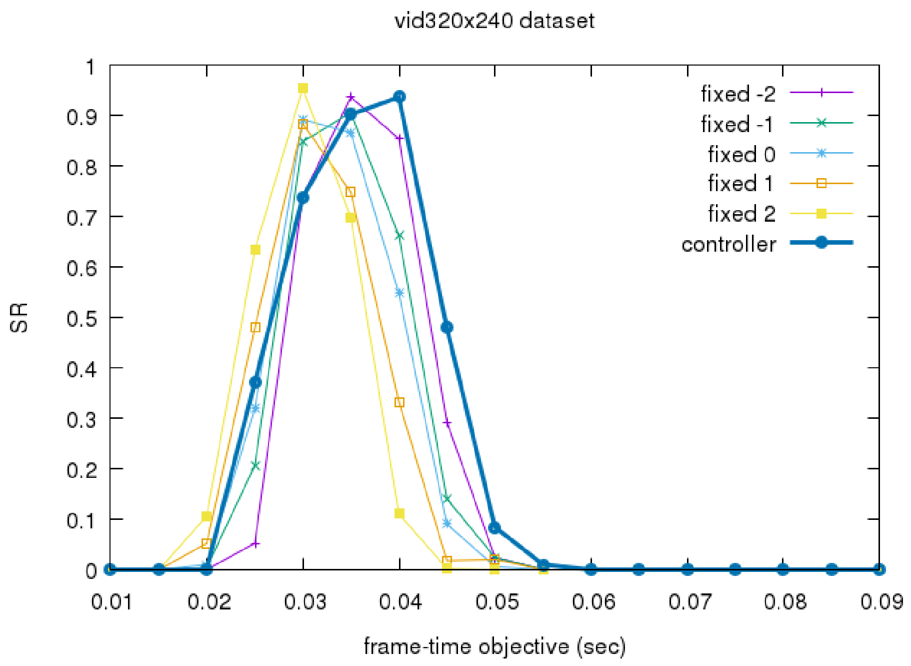


Figure 7.11: ferns $N = 2$: improved monotonic response between x and frame-time



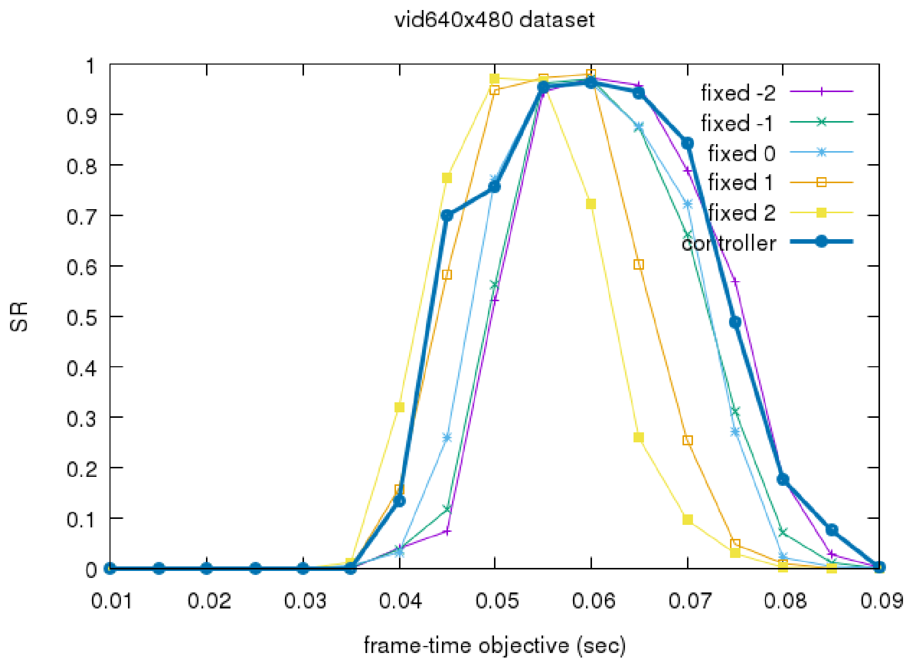
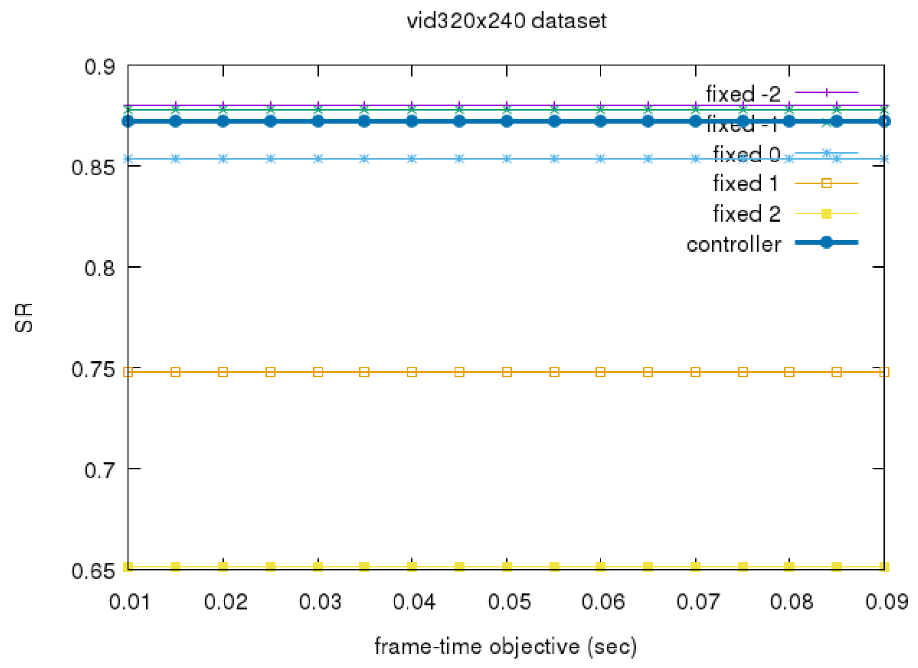


Figure 7.12: ferns: comparison of frame-time QoS for fixed x vs controller across varying frame-time objectives



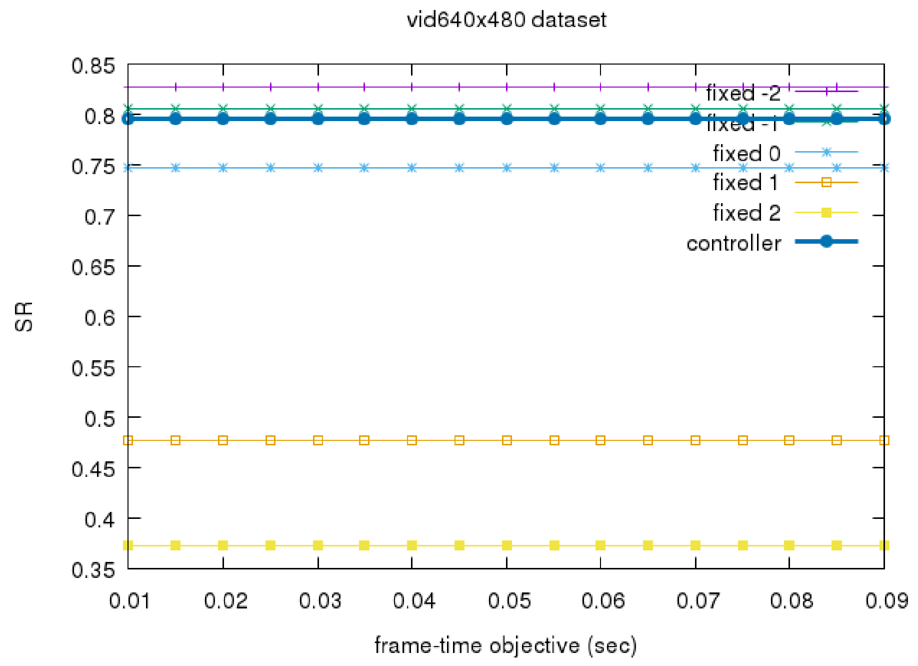
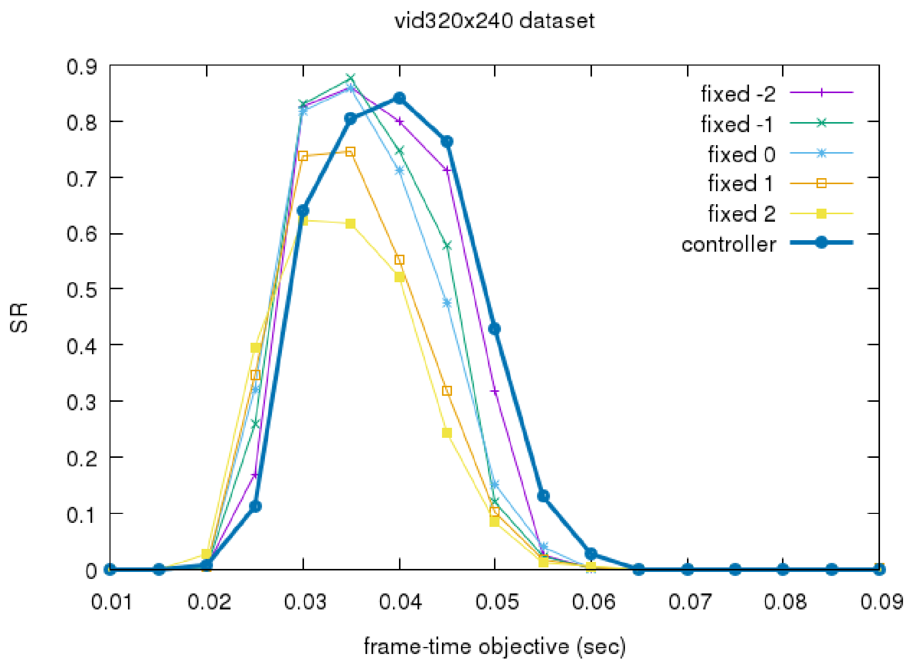


Figure 7.13: ferns: comparison of the accuracy QoS for fixed x vs controller across varying frame-time objectives



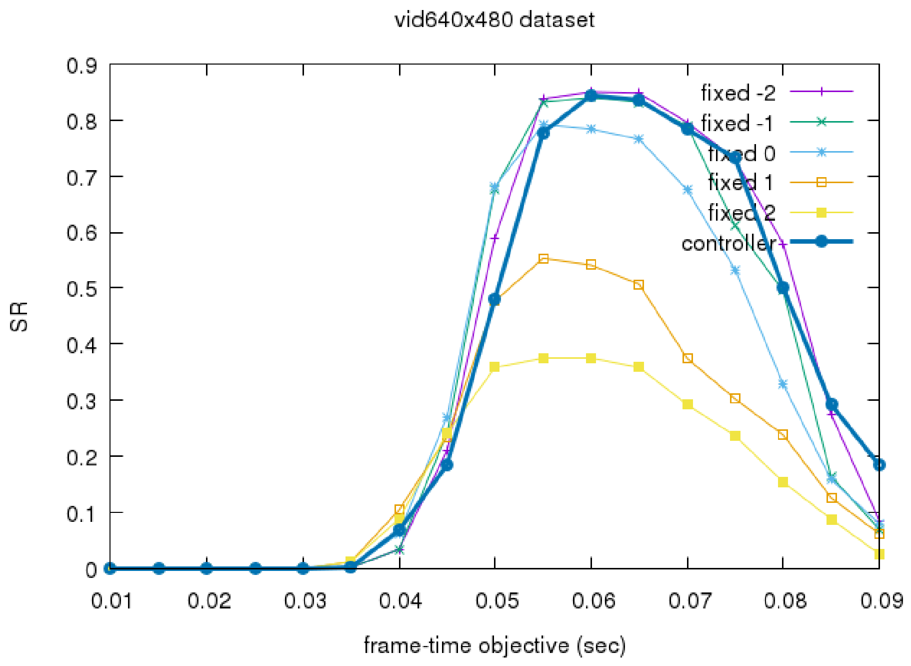


Figure 7.14: ferns: comparison of the combined QoS for fixed x vs controller across varying frame-time objectives

7.6 `rtftr` Detailed Evaluation

All experiments for rtftr are run within an Ubuntu Linux virtual machine running on an Intel i7-3630QM CPU @2.40GHz running Ubuntu Linux.

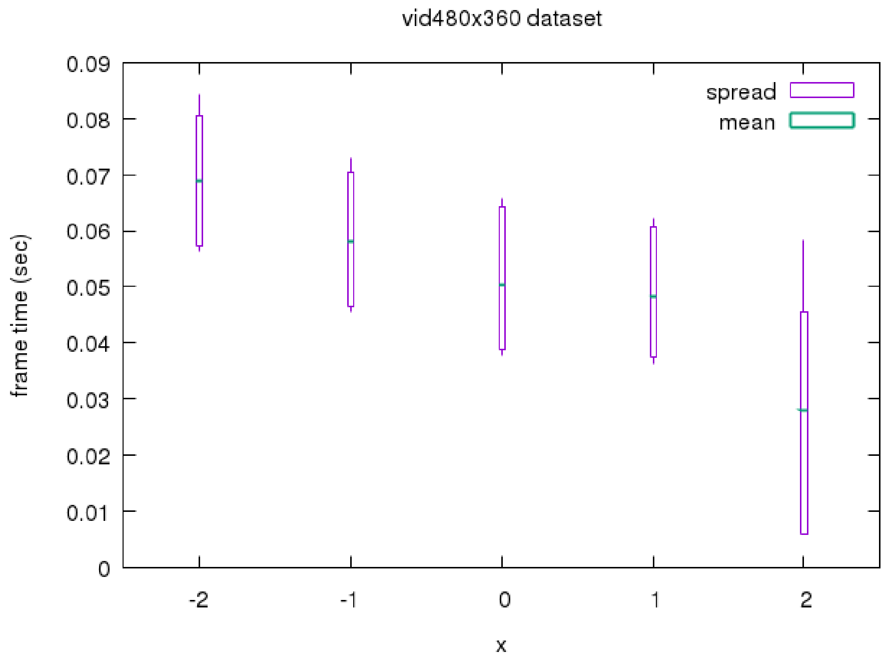
Similar to ferns we had to trim our initial set of mappings of x_1 to scalefactor and x_2 to minrect in order to get better x_1-y and x_2-y monotonicity. Table 7.5 shows the final mappings used. Figure 7.15 shows the resulting *mostly* monotonic response characteristics for scalefactor over two datasets. Similarly, Figure 7.16 shows the monotonic response characteristics for minrect over the two datasets. The two datasets consist of the same test video scaled to two different resolutions — $vid480 \times 360$ and $vid640 \times 480$.

Table 7.5: Mapping of control values to scalefactor and minrect for rtftr

x_1	scalefactor	x_2	minrect
-2	1.1	-2	20
-1	1.2	-1	30
0	1.3	0	50
1	1.6	1	100
2	2.1	2	360

Figure 7.17 shows the frame-time `SR` performance of the controller against fixed cases with scalefactor alone, and Figure 7.18 with minrect alone. Figure 7.19 shows the frame-time `SR` performance when scalefactor and minrect are simultaneously tuned by the controller. We especially see the cumulative benefits of tuning both parameters for larger values of $\tilde{y}$ — while the individual tuning of scalefactor and minrect could not deliver high `SR`, the joint tuning does.

Similarly, Figures 7.20-7.22 show the combined accuracy and frame-time QoS performance. Figures 7.17-7.22 provide details on the results previously summarized in Figures 7.1-7.3.



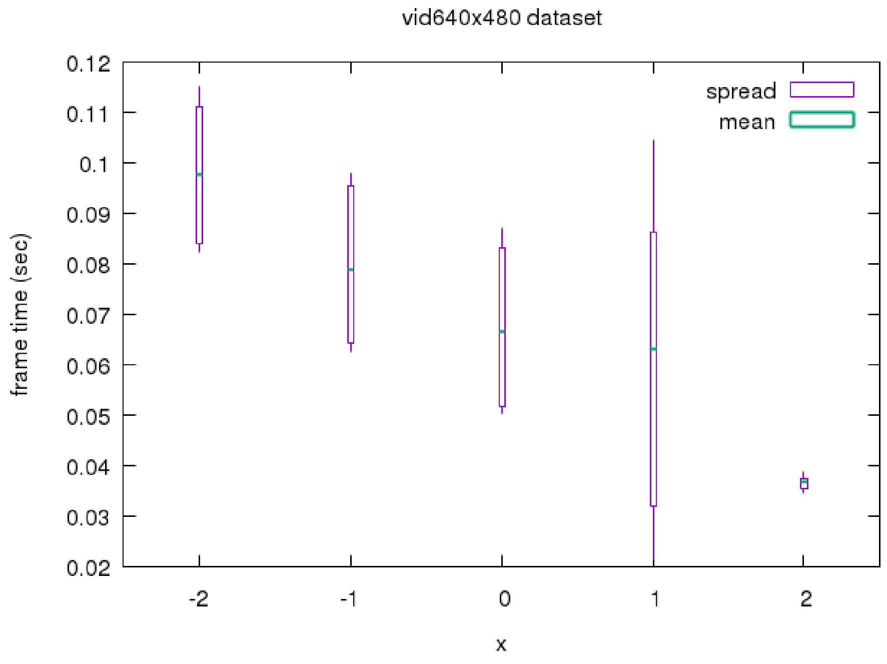
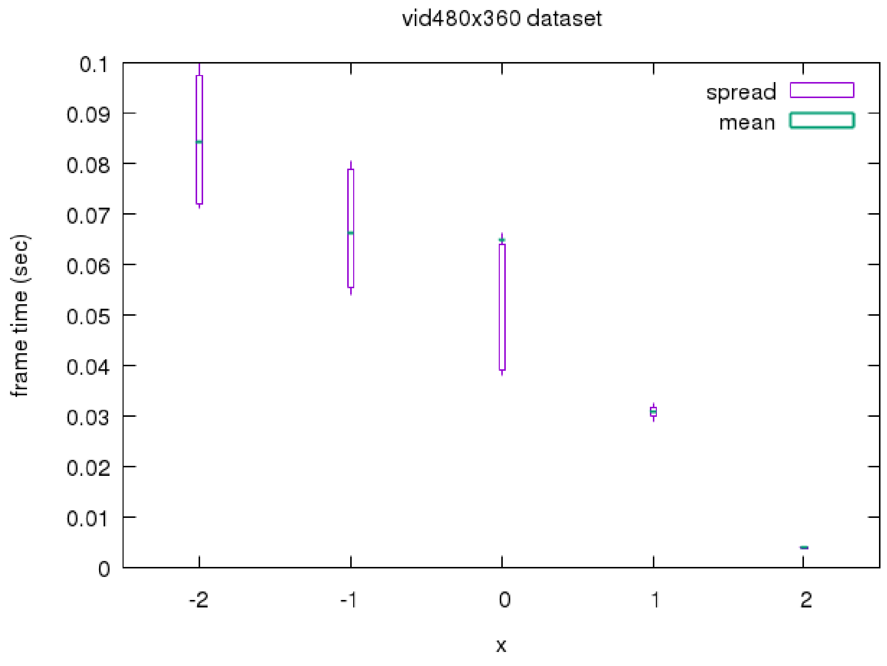


Figure 7.15: rtftr: mostly monotonic response between x_1 (scalefactor) and frame-time



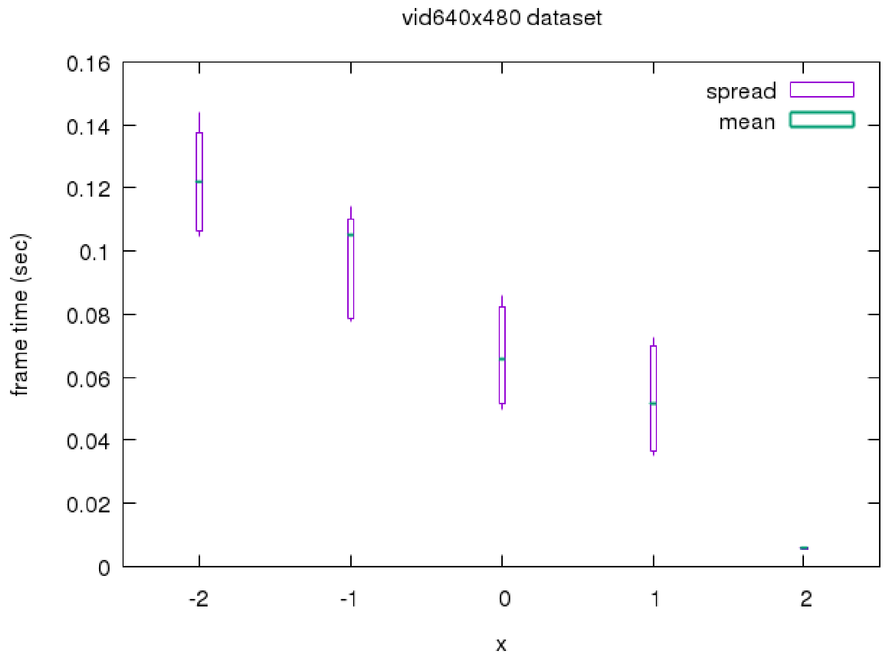
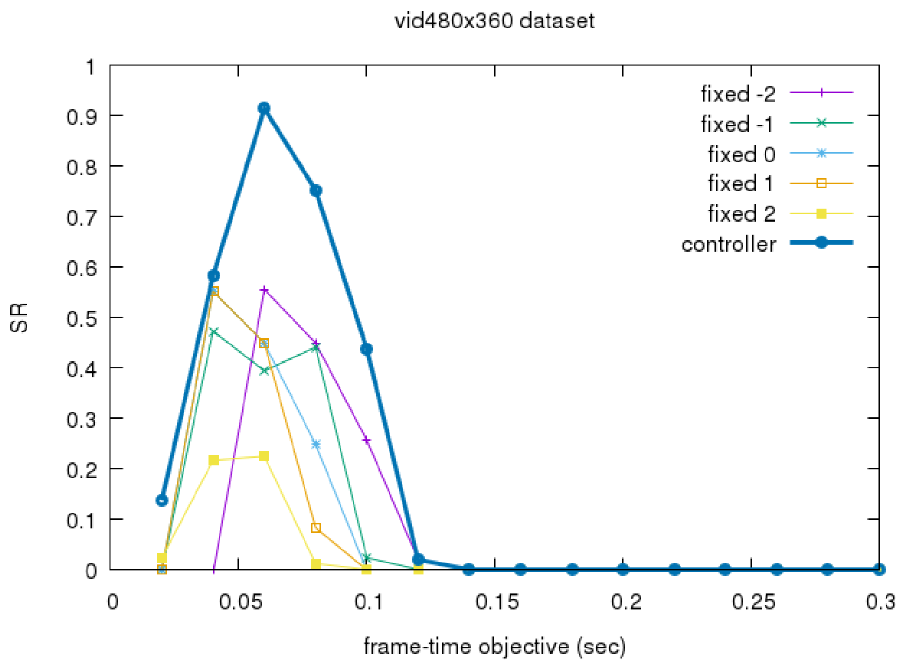


Figure 7.16: rtftr: mostly monotonic response between x_2 (minrect) and frame-time



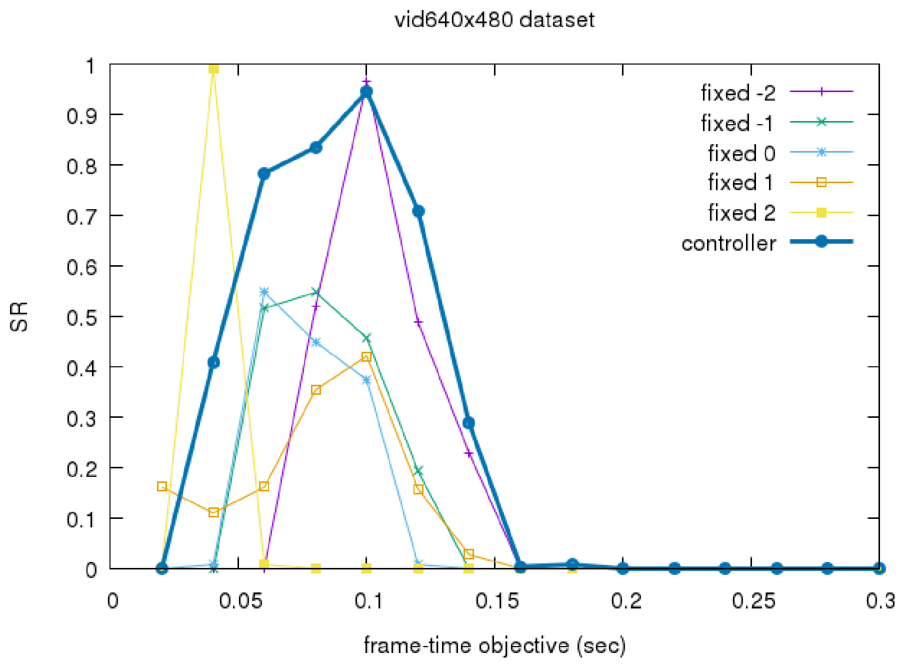
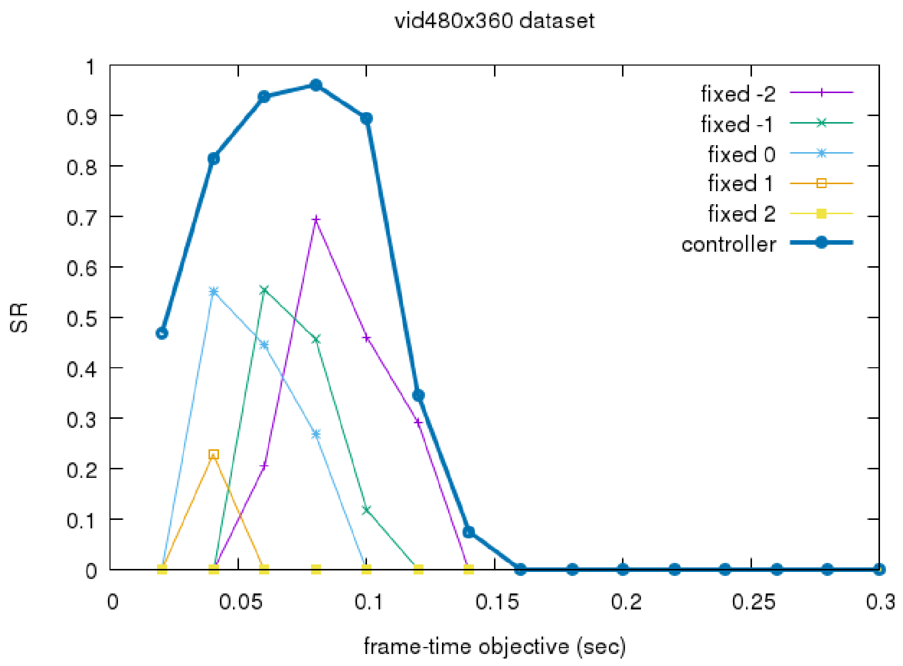


Figure 7.17: rtftr: comparison of frame-time QoS of fixed x_1 (scalefactor) vs controller across varying frame-time objectives



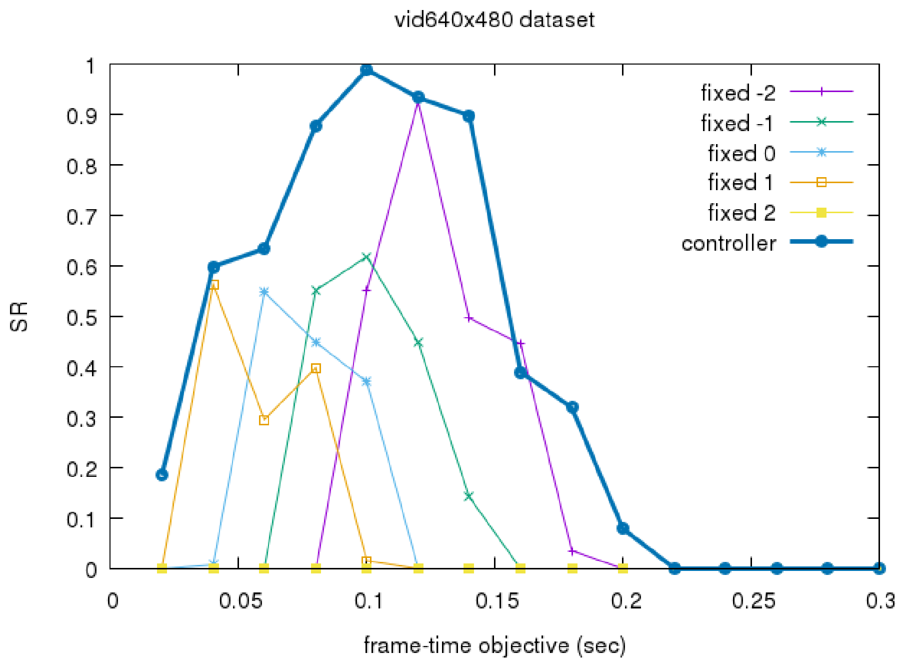
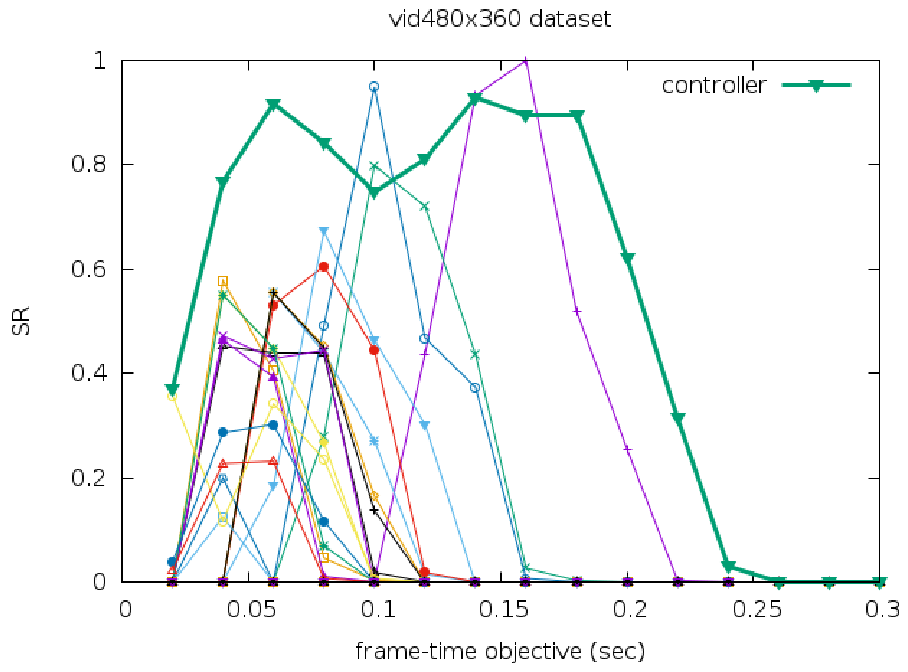


Figure 7.18: rtftr: comparison of frame-time QoS of fixed x_2 (minrect) vs controller across varying frame-time objectives



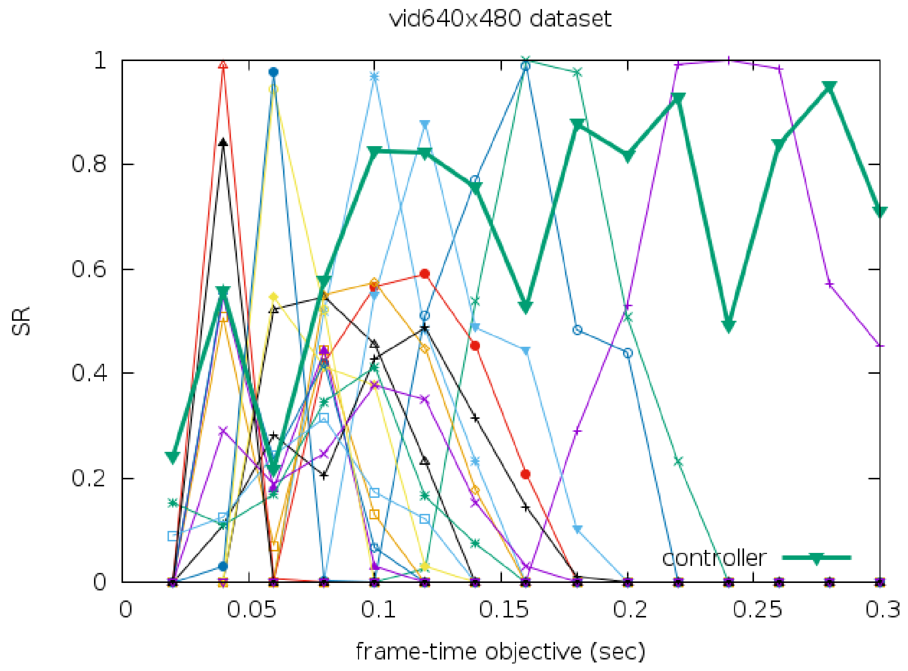
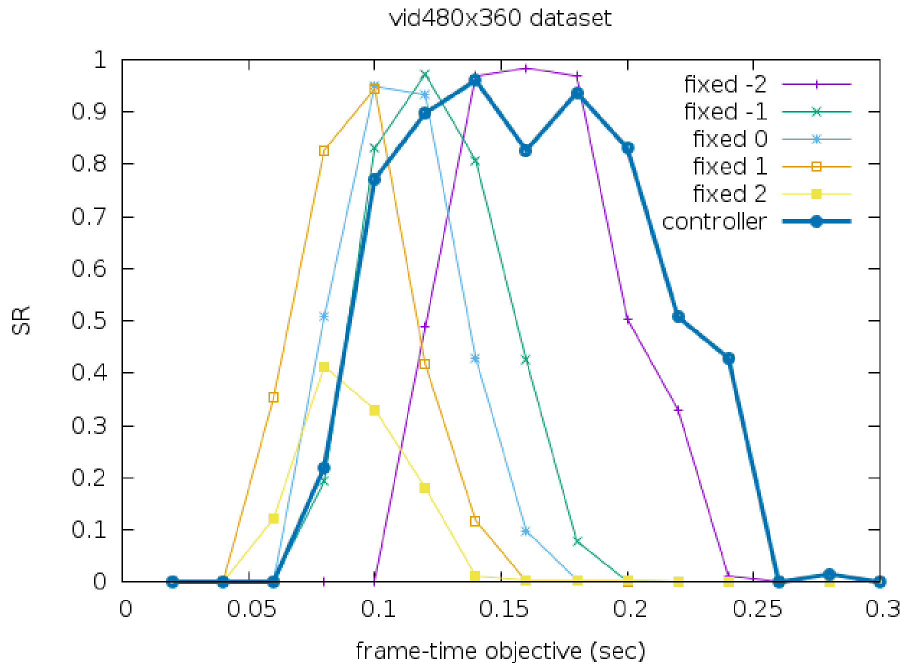


Figure 7.19: rtftr: comparison of frame-time QoS of fixed x_1, x_2 (scalefactor, minrect) vs controller across varying frame-time objectives



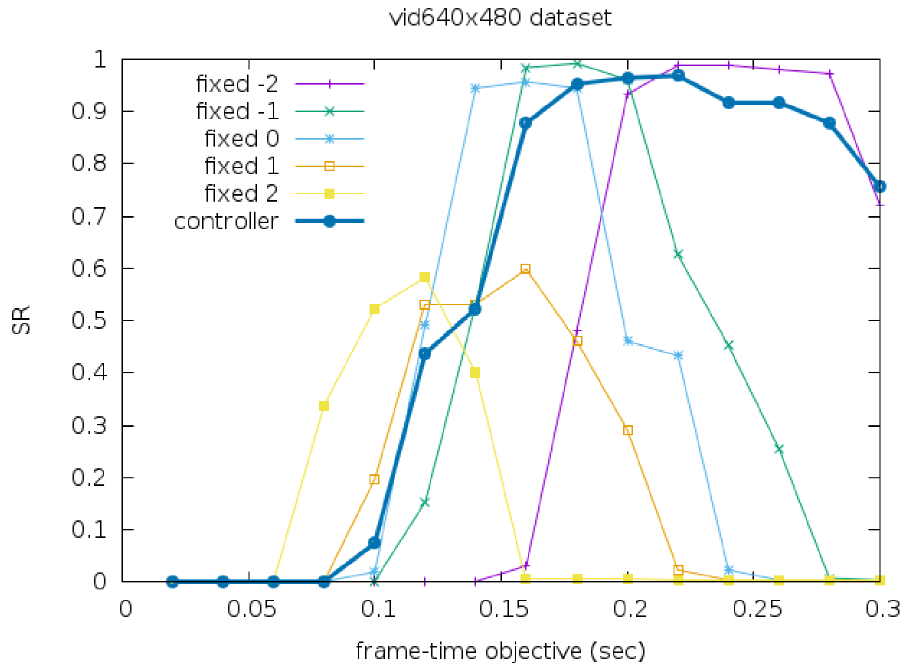
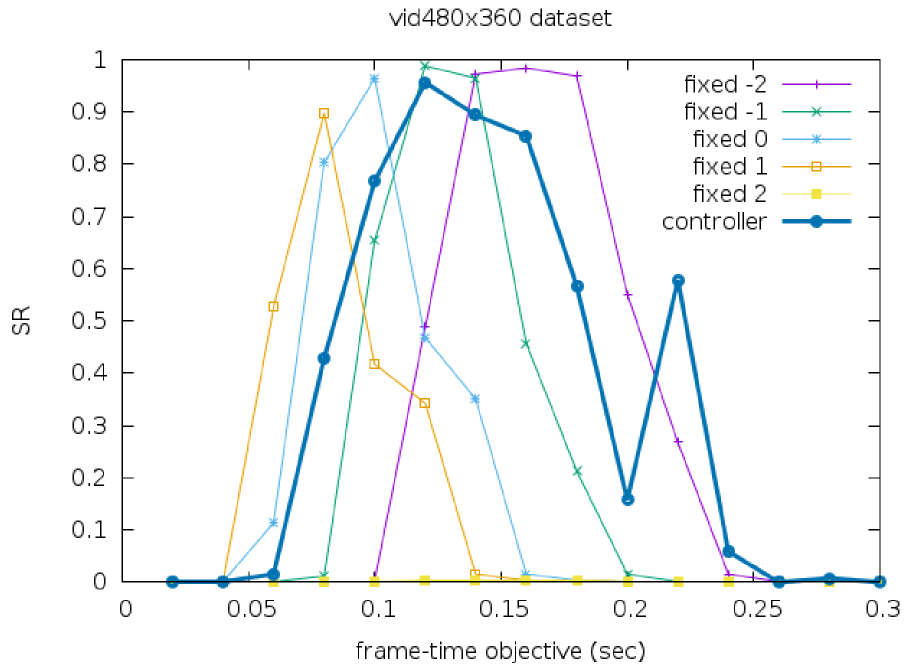


Figure 7.20: rtftr: comparison of combined accuracy and frame-time QoS of fixed x_1 (scalefactor) vs controller across varying frame-time objectives



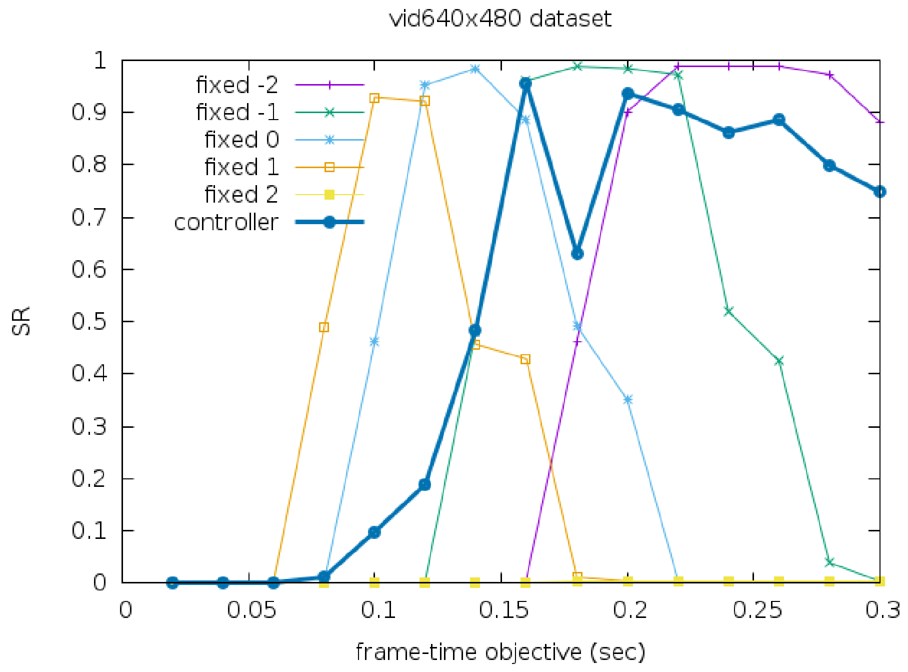
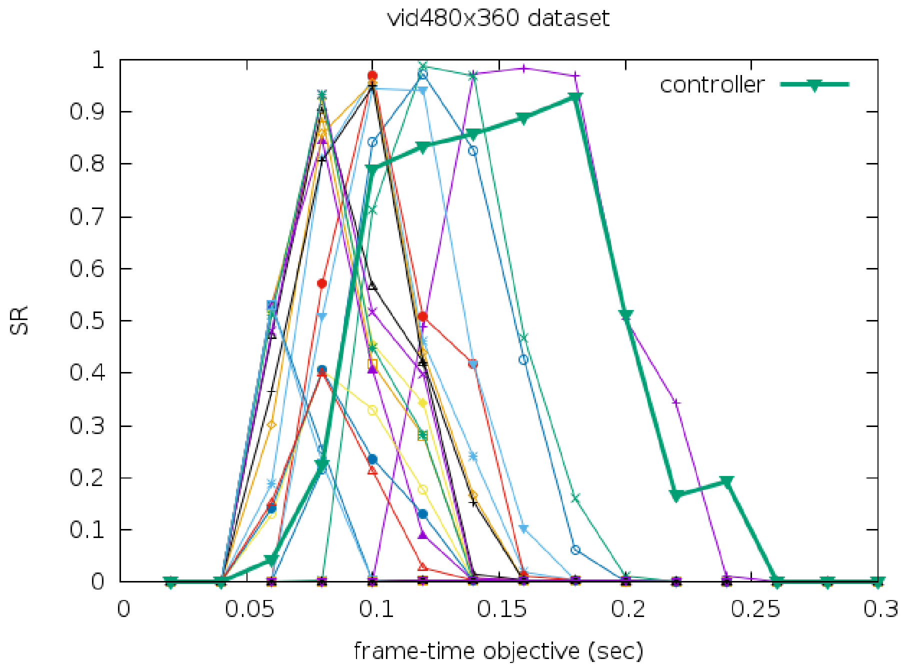


Figure 7.21: rtftr: comparison of combined accuracy and frame-time QoS of fixed x_2 (minrect) vs controller across varying frame-time objectives



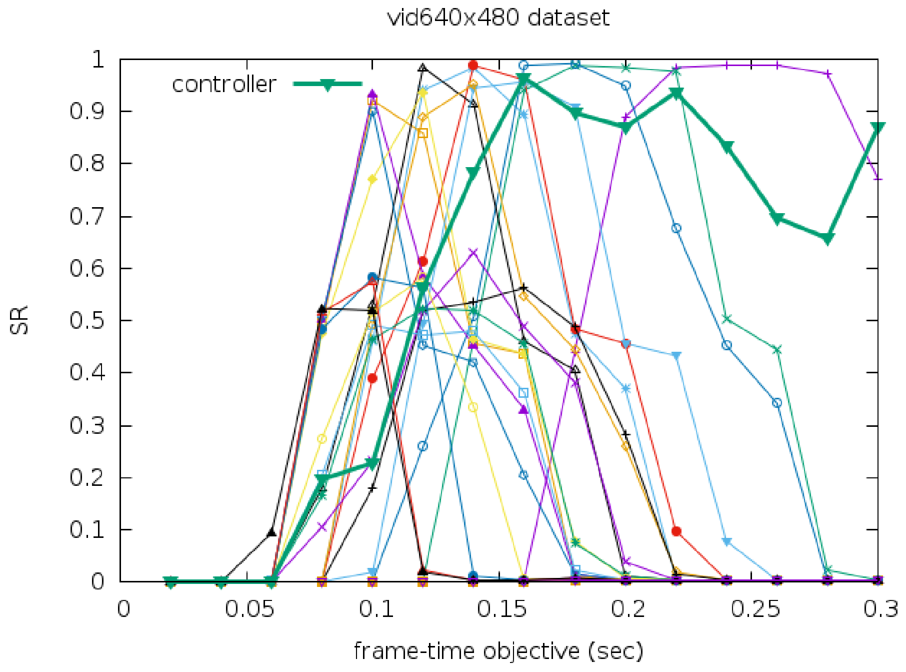


Figure 7.22: rtftr: comparison of combined accuracy and frame-time QoS of fixed x_1, x_2 (scalefactor, minrect) vs controller across varying frame-time objectives

7.7 `x264` Detailed Evaluation

All experiments for x264 are run on an Intel Q6600 CPU @2.40GHz desktop computer running Ubuntu Linux.

Table 7.6, Table 7.7 and Table 7.8 show the mapping of the control parameters for selecting number of cores **nc**, selection of sub-pixel interpolation mode **sp**, and the motion estimation search window size **me**. The sub-pixel mode choices select three out of a possible ten modes, to both *maximize the range* on frame-time and fidelity covered and to *create sufficient separation* in the impact on frame-time and fidelity between adjacent values, based on experimental evaluation on datasets. Similarly, we sample a large range of possible motion estimation search window sizes into three values for the **me** control parameter. Ideally, we would choose a denser set of samples for **me**, but we restricted to three widely separated samples to limit the time for running experiments over all the possible combinations of the three control parameters (**nc**, **sp**, and **me**) when evaluating the fixed cases.

Table 7.6: Number of cores for x264 ($N = 1$)

nc	Number of cores
-1	1
0	2
1	4

Table 7.7: Choice of sub-pixel interpolation mode for x264 ($N = 1$)

sp	Sub-pixel interpolation mode
-1	SAD mode decision, one qpel iteration (setting 1)
0	SATD mode decision, more qpel (setting 4)
1	RD mode decision for all frames (setting 7)

Table 7.8: Motion estimation search window size for x264 ($N = 1$)

me	Motion estimation search window size
-1	4
0	16
1	40

Figure 7.1 summarizes the improvement to frame-time QoS achieved when the **nc**, **sp**, **me** control parameters are exercised individually and in all combinations. The results show moderate to large improvements in `SR` by the use of the controller in each combination

compared to the best performing fixed case for the corresponding combination. Additionally, we can infer that the controller is able to deliver greater improvements in `SR` when two parameters are exercised together instead of one, with the best improvement when all three parameters are exercised together.

7.8 `bodytrack` Detailed Evaluation

All experiments for bodytrack are run on an Intel Q6600 CPU @2.40GHz desktop computer running Ubuntu Linux.

Table 7.9 and Table 7.10 show the mapping of the control parameters for selecting number of cores **nc**, and the number of particles used **np**. As with other benchmarks, the application is written to use fixed values for these parameters. We modify the application so the controller may dynamically tune them every frame. The reference data-set was set up to be used with 4000 particles in the unmodified application.

Table 7.9: Number of cores for bodytrack ($N = 1$)

nc	Number of cores
-1	1
0	2
1	4

Table 7.10: Number of particles for bodytrack ($N = 2$)

np	Number of particles
-2	2000
-1	3000
0	4000
1	5000
2	6000

Figure 7.1 summarizes the improvement to frame-time QoS achieved when the **nc** and **np** control parameters are exercised individually and together. The results show large improvements in `SR` by the use of the controller in each combination compared to the best performing fixed case for the corresponding combination. Additionally, we can infer that the controller is able to deliver significantly greater improvements in `SR` when the two parameters are exercised together instead of individually.

CHAPTER VIII

CALL-CONTEXT VARIANCE ANALYSIS

In this work [81, 82] we use segments of call chains to distinguish the behavior of highly variant functions under their calling contexts. Multiple individual *full* call chains get summarized as a set of very few and very short call-chain *segments* that *uniquely predict the behavior* of the variant function associated with them. By extracting *simple, short signatures* from a large collection of behavior, we attempt to extract a more *general pattern of behavior* that would be representative of the behavior of the application on future, as yet unseen, data sets. In this manner, any program tuning or algorithmic optimizations carried out by the programmer after viewing the variance-analysis results produced for a profiling data set, would also similarly tune or optimize the behavior of the application for any future regression data sets.

As an important use-case in its own right, and as an illustration of the use of the variance-characterization results for other kinds of program-tuning goals, in this work we study an application's variant behavior with an eye towards tuning the application's *soft-real-time behavior*, i.e., detecting and characterizing behavior that is likely to impact user-responsiveness or frame-rate in an interactive application.

We pose the following research questions in order to drive the design of our profile-analysis framework:

Question 1 Component discovery *Can the recurrent behavior identified during the profiling of function calls be used to identify the individual components of an application's soft-real-time functionality?*

Question 2 Structure discovery *Can the identified recurrent behavior be used to reconstruct the soft-real-time structure of the application? The structure would consist of the components of soft-real-time functionality identified in the application.*

Question 3 Context-sensitivity discovery *Can the context-dependent variability in the behavior of the soft-real-time components be detected? The behavior of a component may differ significantly depending on what context it is invoked under.*

Question 4 Generality *How reliably can behavior discovered during profiling be expected to generalize to future runs of the application on arbitrary inputs?*

8.1 Contributions

We make the following specific contributions in this work.

- We describe a tractable approach for succinctly capturing the behavior of *millions-to-billions* of profile events in terms of *tens* of soft-real-time components. The discovered components are the functions that introduce significant variability to the application's real-time behavior, and hence are the most important ones to be brought to the attention of a programmer interested in improving soft-real-time behavior.

- We demonstrate that function-call-chain segments capture the context sensitivity of a component's soft-real-time behavior. We motivate how varying the length of the call-chain segments gives them a varying ability to differentiate between the multiple contexts of invocation of a component. We provide algorithms that choose the correct segment lengths that produce highly succinct profile results that differentiate between only those contexts where behavior is significantly different.

- We illustrate the use of specific statistical theory for constructing algorithms that find *patterns* of behavior (dominant components and their corresponding invocation contexts). As a consequence of the probabilistic guarantees provided by the statistical theory, the produced patterns generalize well for describing the behavior of the application executing on arbitrary input data.

We validate the correctness of the identified components by profiling well-known multimedia applications. Extensive prior research exists about the soft-real-time behavior of these applications. The components reported by our profiling methodology match closely with those described in prior research as the main causes of soft-real-time variance in these applications.

Among the questions listed above, only the structure discovery question is not satisfactorily answered by our current methodology (leading us to create the Dominant Variance Analysis technique, described in Chapter 9). Although the discovered components and their call-chain contexts do allow the programmer to infer the structure, this inference is not sufficiently precise and may not work in all circumstances.

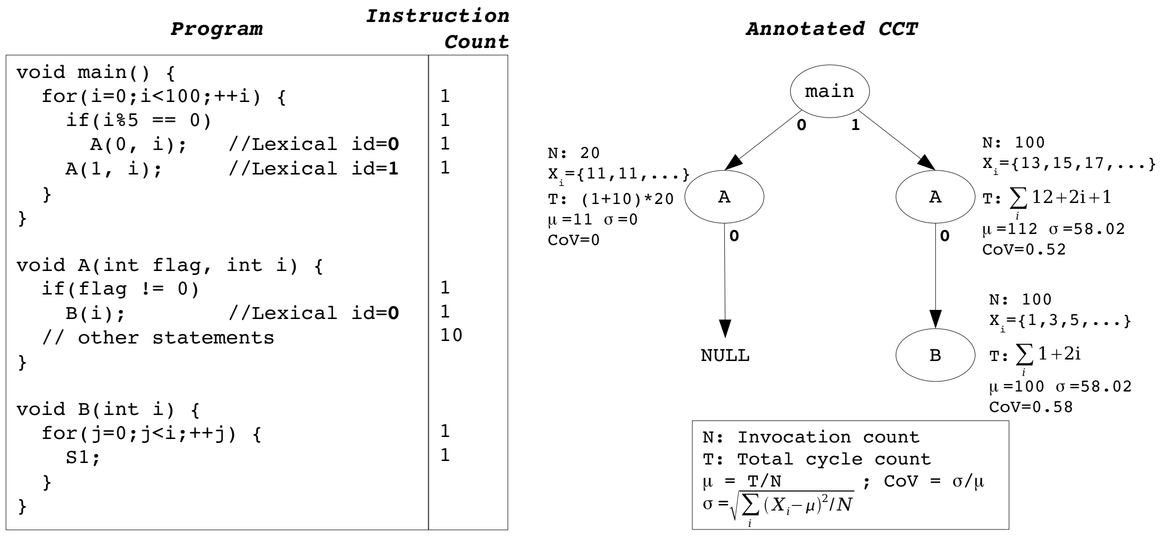
Overview Section 8.2 introduces the profile representation constructed from the raw stream of profile events. Section 8.3 introduces the relevant statistical theory and describes the analysis performed on the constructed profile representation to detect patterns. Section 8.4 provides experimental validation.

8.2 Profile Representation

We profile instrument the application and use the generated profile events to construct a *calling-context tree* (CCT). Ammons et al. [83] showed that a CCT representation succinctly captures the dynamic structure of the function calls executed by the application. It preserves the full call-chain context of invocation, and merges information along multiple identical contexts into a single context. This makes the CCT an ideal representation for investigating context-sensitive behavior.

We automatically profile instrument a C/C++ application using the LLVM [84] compiler infrastructure. We execute the application on test inputs and use the generated sequence of profile events to construct the CCT as described by Ammons et al. [83].

During CCT construction, we annotate statistics on each CCT node. These statistics are used by subsequent analysis for detecting patterns. Figure 8.1 shows an example program, the corresponding CCT and the annotated node statistics. Function `A` was invoked from two call sites within the parent function `main`. This leads to two children nodes for function `A`. Since function `B` was never invoked under the left `A` node, it only gets a *NULL* edge at its call site in A. The next subsection describes the node annotations required for our variant call-context analysis.



```
void main() {                                      1
  for(i=0;i<100;++i) {                             1
    if(i%5 == 0)                                   1
      A(0, i);    //Lexical id=0                   1
    A(1, i);      //Lexical id=1
  }
}

void A(int flag, int i) {
  if(flag != 0)                                    1
    B(i);         //Lexical id=0                   1
  // other statements                              10
}

void B(int i) {
  for(j=0;j<i;++j) {                               1
    S1;                                            1
  }
}
```

Figure 8.1: An example program, and the corresponding CCT with annotated node statistics.

8.2.1 Node Annotations

Each node is annotated with the following statistics about its corresponding function call:

1. `invocation count` N: The number of times the corresponding function call was invoked.

2. `mean` $\bar{X}$: The mean execution time across all invocations of the function call corresponding to the node. *This includes the execution time of all children function calls.*

3. `variance` σ^2: The statistical variance in the execution time of the function call across all invocations. The variance is the square of the standard deviation σ. Using the property $\sigma^2 = E(X^2) - \bar{X}^2$, a single pass over the profile data constructs the CCT and computes all node annotations, including the variance.

8.3 Detecting Patterns of Behavior

Once the CCT has been constructed and its node annotations calculated, the CCT is traversed in pre-order for analysis. Nodes whose total execution time constitutes a minuscule fraction (say, $< 0.02\%$) of the total execution time of the program are deemed as *insignificant*. All other nodes are deemed *significant*. Since a CCT node subsumes the execution time of all its children nodes, once a node is found to be insignificant, the nodes in its children subtree are all guaranteed to be insignificant as well.

8.3.1 Tagging Nodes

We examine the annotations of nodes to determine if the corresponding nodes exhibit high variance in their execution time within the context of the caller function (parent node). This is captured by the variance $P.\sigma^2$. We use Chebyshev's inequality [85], given below by Equation 69, to determine meaningful thresholds to compare a node's variance against. **Chebyshev's inequality** establishes conservative probability bounds on a given collection of data samples *while making no assumptions about the underlying probability distribution that generated the data.*

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{69}$$

In our experiments, we define a node to be high-variant if its execution time cannot be guaranteed to lie within 200% of its mean with at least 96% probability (i.e., for at least 96% of the invocations of the node). This implies $\frac{1}{k^2} = 1 - 0.96 = 0.04$ and $k\sigma = 2 \times \mu$. Therefore $\frac{\sigma}{\mu} \geq 0.4$ becomes the condition for high variance. Consequently, we use the Coefficient-of-Variability metric for classifying the variant nature of nodes: $CoV = \frac{\sigma}{\mu}$. The choice of the variance-window around the mean and the probability of samples falling within it can be tweaked by the user based on the method described above. As the programmer pushes the probability guarantee of samples falling within the $k\sigma$ variance window to 100%, $\frac{1}{k^2} \to 0$ and $k \to \infty$. This implies that the window $k\sigma \to \infty$ would trivially encompass all possible execution times. Therefore, it is practical to keep the probability not too close to 100%, and for almost all soft-real-time applications a probability guarantee of 96% would suffice, though this can be adjusted to the guarantees desired for any given application. Qualitatively, $k\sigma = 200\% \times \mu$ suggests a highly-variant behavior as the execution time can increase to over thrice the mean execution time (and reduce all the way down to 0). Based on how stringent the soft-real-time requirements are for an application, the programmer can adjust the threshold that defines high-variant behavior.

Once the CCT is constructed from the profile data, it is pre-order traversed in linear time and individual nodes may be *tagged* as being *high-variant*. As mentioned earlier, the traversal is restricted to significant nodes.

8.3.2 Signature Generation for Patterns

The previous subsection described how the significant nodes in the CCT were individually tagged if they exhibited statistical high-variance. The next step is to find the *patterns of call chains* whose presence on the call stack can be used to predict the occurrence of the high-variance behavior found at the tagged nodes. For a given tagged node P, we restrict the call-chain pattern to be some contiguous segment of the call chain that starts at `main` (the CCT root node) and ends at P.

The *sequence of names* of the function calls in the call-chain segment becomes the detection pattern arising from the tagged node. This particular detection pattern might occur at other places in the significant part of the CCT. Quite possibly, the occurrence of this detection pattern elsewhere in the CCT does not match the statistical behavior, i.e., the mean and the CoV values, that were observed at the tagged node. Therefore, our key criteria in generating the detection pattern is the following:

> All occurrences in the significant CCT of a detection pattern arising from a high-variance-tagged node must exhibit the same statistical behavior that was observed at the tagged node.

Notice that this condition is trivially satisfied if we allow our detection pattern to extend all the way to `main` from the tagged node, since this pattern cannot occur anywhere else as a result of its full call context being a unique path in the CCT. In many applications, patterns extending to `main` are likely to *generalize very poorly* to the *regression execution* of the application on arbitrary input data. Regression execution refers to the *real-world-deployed execution* of the application, as opposed to the *profile execution* of the application that produced the profile data used for constructing the CCT. In many applications, we expect the behavior of the function call at the top of the call stack to be correlated with only the function calls just below it in the call stack. This short call sequence would be expected to produce the same statistical behavior regardless of where it was invoked from within the program (i.e., regardless of what sits below it in the call stack). In this work we focus our attention on detecting just such call sequences. We call such a call sequence the *minimal distinguishing call-chain* (MDCC) pattern corresponding to any particular statistical behavior. MDCC patterns are the shortest-length detection sequences whose occurrence at the top of the call stack predicts the behavior at the tagged node, with no mis-prediction of behavior occurring in the significant CCT. An MDCC pattern is illustrated in Figure 8.2.

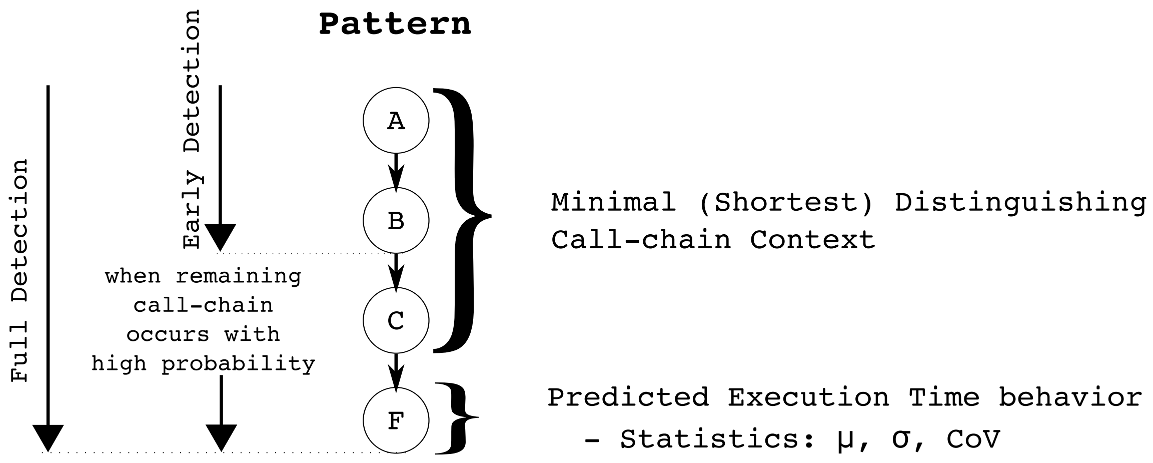
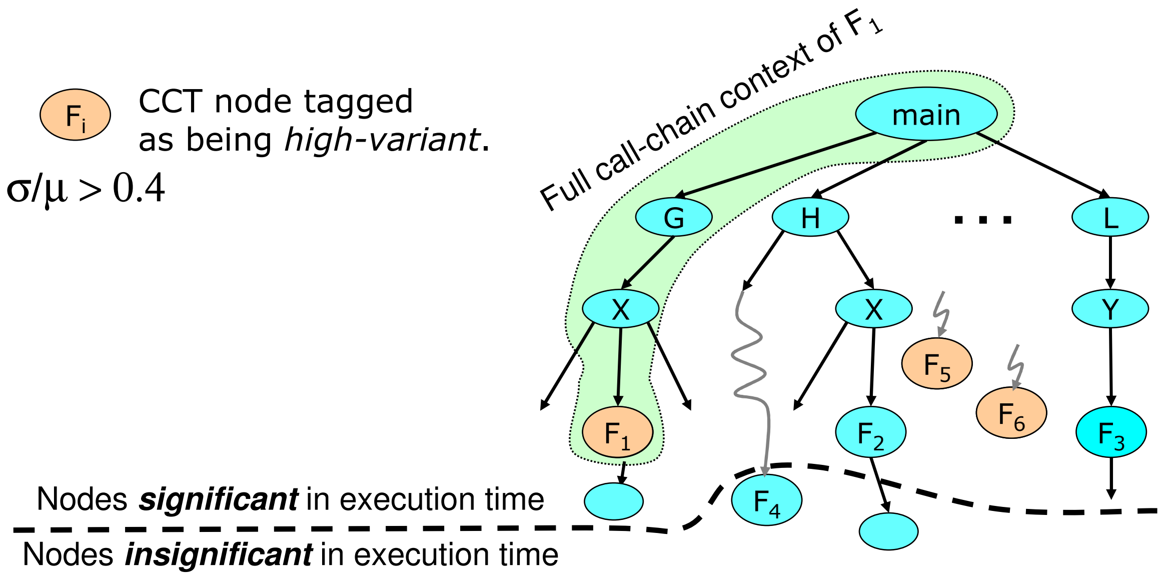

Pattern
Early Detection
Full Detection
A
B
C
F
when remaining call-chain occurs with high probability
Minimal (Shortest) Distinguishing Call-chain Context
Predicted Execution Time behavior
- Statistics: μ, σ, CoV

Figure 8.2: Minimal distinguishing call-chain context (MDCC) pattern.

Our paper [82] provides the algorithms for constructing MDCCs along the criteria discussed above. Figure 8.3 illustrates a CCT with the high-variant nodes for function F tagged. The figure also illustrates a full call-chain context from `main` for node F_1.



F_i
CCT node tagged as being *high-variant*.
$\sigma/\mu > 0.4$
Full call-chain context of F_1
main
G
H
L
X
X
Y
F_1
F_2
F_3
F_4
F_5
F_6
Nodes ***significant*** in execution time
Nodes ***insignificant*** in execution time

Figure 8.3: Illustration of a CCT with the high-variant nodes tagged.

Figure 8.4 illustrates how the full call-chains for the significant nodes of F (from Figure 8.3) are processed to extract MDCCs that distinguish the high-variant behavior of F from the low-variant behavior of F.

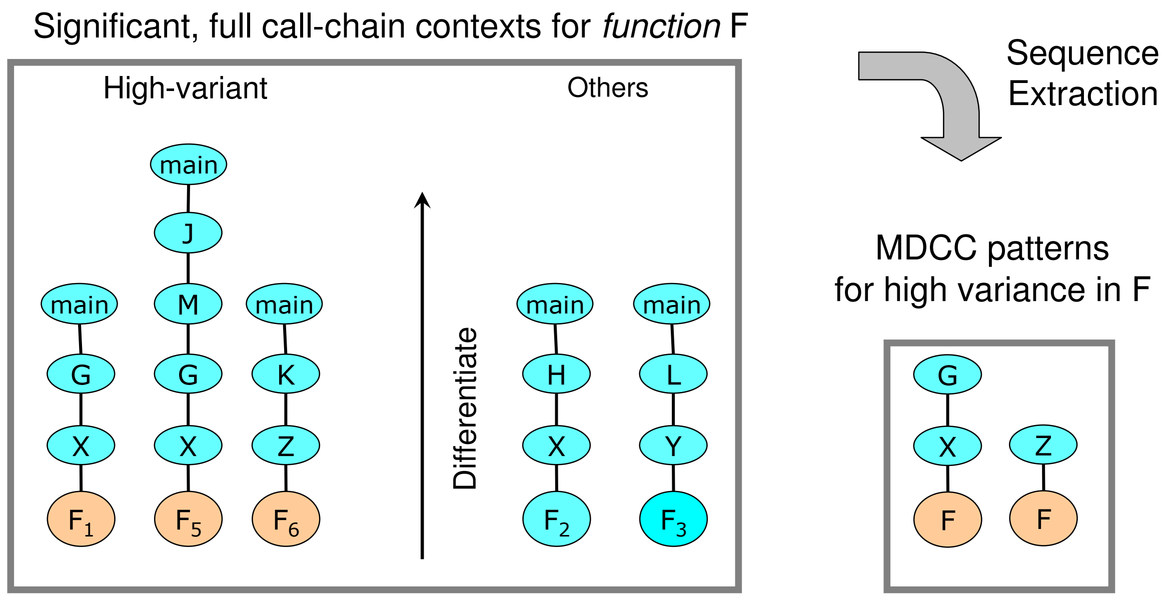


Figure 8.4: Illustration of MDCC extraction from full call-chain contexts.

8.3.3 Grouping and Distinguishing between Similar Patterns

In the previous discussion, we assumed that the programmer desired to distinguish between tagged nodes whenever their statistics (mean, CoV) did not match exactly. However, the exact matching of statistics may lead to very long detection patterns that *generalize poorly* to the regression runs. For example, if multiple high-variant-tagged nodes with somewhat different means require long call chains to distinguish between each of them, then it may be preferable to actually have a shorter call-chain pattern that does not distinguish between these tagged nodes.

Furthermore, if the same detection sequence occurs at multiple tagged nodes in the significant CCT and the nodes have matching statistics, we would like to combine the multiple occurrences of the detection sequence into a single detection sequence. Such detection sequences are likely to *generalize very well* to the regression run of the application, and are therefore quite important to detect.

In order to address the preceding two concerns in a unified framework, we first generate short call-chain patterns using only the "broad-brush" notions of high variance, without distinguishing between the tagged nodes using their statistics (mean, CoV). Then we group patterns with identical call contexts (arising from different tagged nodes) and use *pattern-similarity trees* (PSTs) to start differentiating between them based on their statistics. The initial group forms the root of a PST. We apply a *similarity measure* (`SM`) function on the group to see if it requires further differentiation. If the patterns in the group have widely different means or CoVs, and the programmer wants this to be a differentiating factor, then the similarity check with the appropriate `SM` will fail. In our experimental evaluation, we use an `SM` that checks if the corresponding means and CoVs of the two patterns being compared are within 10% of each other; the programmer can choose to plug in a different `SM`, say one that checks only on means.

Once the `SM` test fails on a group, all the patterns in the group are extended by one more parent function from their corresponding call chains (tagged CCT nodes are kept associated with the patterns they generate). This will cause the resulting longer patterns to start to differ from each other. Again, identical longer patterns are grouped together as multiple children groups under the original group. This process of *tree subdivision* is continued separately for each generated group until the `SM` function succeeds in all current leaf nodes. At this point, each of the leaf groups in the PST contain one or more identical patterns. The patterns across different leaf groups are however guaranteed to be different in some part of their prefixes. Patterns in different leaf groups may be of different lengths, even though the corresponding starting patterns in the root PST group were of the same length. All the identical patterns in the same leaf node are collapsed into a single detection pattern. For example, an `SM` function that differentiates on σ but not on means (or only weakly on means), will produce leaf nodes that contain patterns with a single σ but a collection of widely varying means.

Figure 8.5 illustrates how identical MDCC patterns identifying high-variant behavior can be selectively extended using PSTs to distinguish between high-variant behavior with differing statistics.

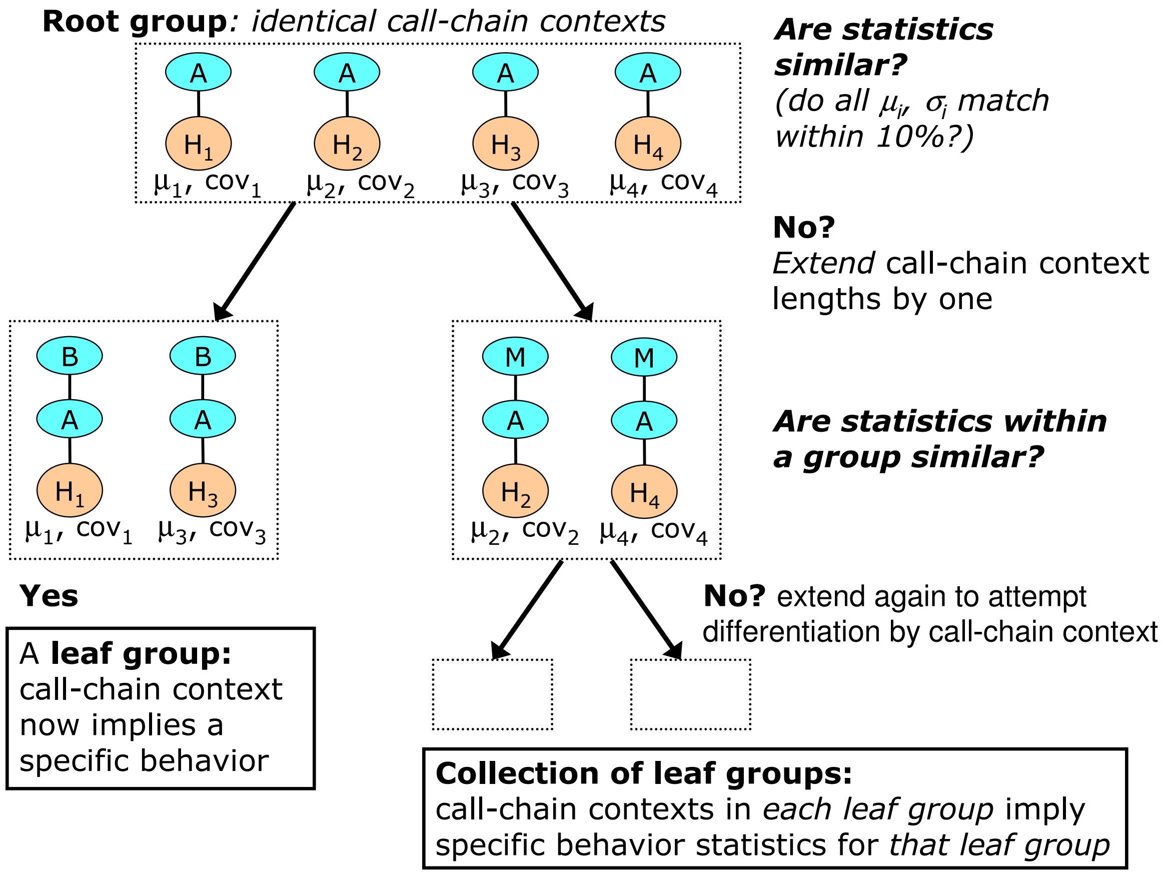

Root group: identical call-chain contexts
A
H1
μ1, cov1
μ2, cov2
μ3, cov3
μ4, cov4
Are statistics similar?
(do all μi, σi match within 10%?)
No?
Extend call-chain context lengths by one
B
M
Are statistics within a group similar?
Yes
A leaf group: call-chain context now implies a specific behavior
No? extend again to attempt differentiation by call-chain context
Collection of leaf groups: call-chain contexts in each leaf group imply specific behavior statistics for that leaf group

Figure 8.5: Illustration of selectively extending MDCC patterns using pattern-similarity trees (PSTs) to distinguish between differing high-variant behavior.

8.3.4 Ranking Impact of Patterns

The previous steps produce numerous patterns (11 to 46 patterns for our benchmarks) characterizing the variability in the application at multiple levels. It is highly desirable to rank the patterns to focus the programmer's attention on the ones that are the most likely to contribute variability to the program. For this purpose we introduce a metric that we call the *variability impact metric* or `VIM`. The Chebyshev inequality introduced earlier points us towards a suitable definition for `VIM`. While the CoV $= \frac{\sigma}{\mu}$ indicates whether the variations are large with respect to the mean, the $k\sigma$ term in the Chebyshev inequality indicates the absolute amount of variability. The variability per invocation multiplied by the total invocation count of that pattern gives the total amount of variability contributed by the innermost function in the pattern to its immediate parent. Therefore, we define `VIM` as follows, with N being the invocation count of the innermost function in the pattern:

$$\texttt{VIM} = k\sigma N. \tag{70}$$

While this metric indicates how much variance is contributed by the innermost function F to its immediate parent C (referring to the pattern in Figure 8.2), it is not necessarily implied that the pattern's variance contribution propagates up the call chain to A or B. For example, if B invokes C from inside a loop, then the `VIM` for C will measure the variance impact to the iterations of the loop, not to B directly. In fact, it is possible that B is not variant at all if each iteration of C consumes correspondingly lower execution time when the loop-count is high, and vice versa when the loop-count is low, leading to a constant execution time for the loop across all invocations of B. A similar situation can occur without loops if B invokes C inside a very infrequently executed branch.

Despite the limitation described above, the profile analysis technique is excellent for *eliminating* unlikely contributors of variance. Therefore, the correct way to interpret the produced patterns is to think of them as *highly likely contributors* of variance. This immediately allows the programmer to narrowly focus on very limited parts of the application in order to identify the causes of violations to the soft-real-time requirements. The programmer would, of course, have to examine *relative invocation counts* along a given pattern's call chain to infer how far up the call chain an innermost function is likely to be contributing variance.

8.4 Experimental Evaluation

Our statistical profile analysis tool has been written in python. We did not use any high-performance numerical or scientific libraries (such as NumPy, SciPy) in the python implementation. We profile instrumented a number of applications in the MediaBench II Video suite and a real-time object-recognition benchmark (mimas-findTux) from the Mimas computer-vision application-suite [86]. We generate profile data (a sequence of profile events) for each benchmark using the input data sets provided with the benchmark suites, or some larger external data sets if the profiles are too short. Specifically, we use two different input data sets for each benchmark, referred to as `D1` and `D2`.

First, we run profile analysis on `D1` to create patterns. Then, we use `D2` with the regression run to validate the statistics of the patterns found previously. The regression run simulates the application call stack using the profile events. No CCT is constructed and no analysis is performed during the regression run. We use a generic finite-state-machine sequence-detector to detect the occurrence of the patterns at the top-of-the-stack. Such a sequence detector needs to check the call stack for the possible occurrence of every pattern on every profile event. This is the cause of the significant slow-down seen in Table 8.1 in the pass times for the regression runs compared to the profile runs. *We would like to emphasize that the profile-analysis time consists entirely of the time to read and parse the profile-data file from disk. All of the actual analysis combined (variance tagging, minimum call-context detection, PSTs, etc.) consumes a fraction of a second.*

Table 8.1 shows the length of the `D1` profile (in terms of the number of `entry` / `exit` events) used to generate the patterns, the number of high-variance patterns found, and the length of the `D2` profile used during regression to simulate the real-world execution of an application. The *pass time* refers to the duration of time needed to complete profile analysis or regression.

Table 8.1: Attributes of the MDCC patterns for our benchmarks.

Benchmark	Profiling on `D1`: Pat. Generation				Regression on `D2`			
	# of steps	*Pass time (seconds)*	*# of patterns*	*Pattern set size*	*# of steps*	*Pass time (seconds)*	*Pattern set size*	*Pat. set overlap*
H.263enc	30000000	397	9	7	60000000	1245	7	100%
H.263dec	25000000	341	30	5	60000000	2194	6	100%
findTux	30000000	402	60	3	40000000	2833	3	100%
mpeg2enc	30000000	387	44	5	60000000	2943	5	100%
mpeg2dec	30000000	402	20	5	60000000	1657	5	80%

Clearly, during regression the input data set is different, which will lead to corresponding changes in the call chains invoked, their invocation counts, and their variant behaviors. However, in our validation we strive to demonstrate that the patterns capture the statistical behavior of the application at a more fundamental level, which tends to remain relatively constant across different data sets. In order to demonstrate this, we introduce the notion of a *pattern set* both for profiling and regression. We define the pattern set to consist of a subset of patterns that are found to be the most impactful as measured by their variability impact metric (`VIM`). Specifically, we limit the pattern set to only those patterns whose `VIM` is at least 10% of the `VIM` of the pattern with the highest `VIM`. The pattern set is constructed separately for profiling and regression, leading to the construction of two potentially disjoint sets. Table 8.1 shows that in fact the regression pattern set very closely mirrors the profiling pattern set (*pattern set overlap* column). This implies that the same set of patterns that were found to be the most impactful during profiling tend to remain the most impactful during regression as well. The pattern set spans an order-of-magnitude of the largest `VIM` values (i.e., $10\times$). We chose to define the pattern set as such because we expect the data-set-induced variations to cause *relative fluctuations* in the pattern statistics across data sets. These fluctuations are a result of the changes in the length of the data (number of events) and the type of the data (for example, encoding video with a constant background versus a moving background, different frame-dimensions, etc.). Despite these variations in the characteristics of the input data, the most impactful patterns found on `D1` tend to remain the most impactful on `D2` as well, validating our intuition that our patterns capture variant behavior in a statistically-sound manner. In mpeg2dec, the `VIM` of one pattern was just slightly smaller during regression causing it to be dropped from the regression pattern set. Similarly, a pattern that had barely missed inclusion in the profiling pattern set got included in the regression pattern set. However, both these patterns have similar `VIM`s (in the order-of-magnitude sense). Therefore, despite a pattern set overlap of only 80%, this result also shows that profiling and regression pattern sets match closely for mpeg2dec. In H.263dec, there was a pattern that barely missed inclusion in the profiling pattern set, but got included in the regression pattern set.

Figure 8.6 shows the distribution of the mean and CoV values for all the patterns discovered, on a per-benchmark basis. For each benchmark, the left side in the scatter plot shows the distribution found during profiling (on `D1`), and the right side shows during regression (on `D2`). No `VIM`-based distinction is made between patterns; the least varying pattern with low invocation count is given a point just like the most impactful pattern. For every benchmark, the distributions for profiling and regression are very similar, except for a uniform linear shift and a uniform scaling of one distribution with respect to the other. When we look at Figure 8.7, plotted using only the patterns in the profiling and regression pattern sets, we again see a close similarity between the profiling and regression distributions, indicating that the dominant patterns are fundamentally associated with the application behavior, regardless of data sets. For example, encoding raw video with a larger frame-image size quadratically increases the mean and possibly the CoV of a motion-estimation pattern, but motion-estimation remains dominant independent of the frame-image size.

The following is representative across the benchmarks of the *compaction of information* achieved in going from raw profile data to the final profile results: 800MB to 1.3 GB of raw profile-event data reduced to a CCT with 600 to 800 nodes, out of which 200 to 350 nodes were found significant, out of which 16 to 116 nodes were tagged high-variant, which were grouped down to 9 to 60 patterns with identical contexts and similar means and CoVs (using pattern-similarity trees), finally out of which 3 to 7 were dominant patterns (the pattern set).

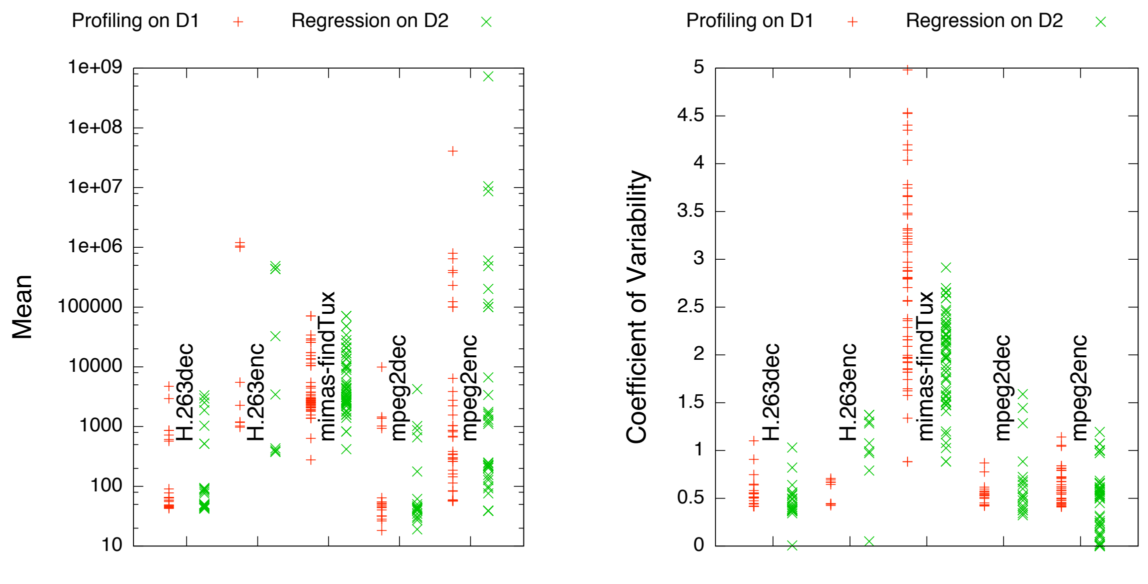


Figure 8.6: *Mean* and *CoV* scatter-plots comparing the profiling `D1` and regression `D2` data sets, constructed using **all the patterns**.

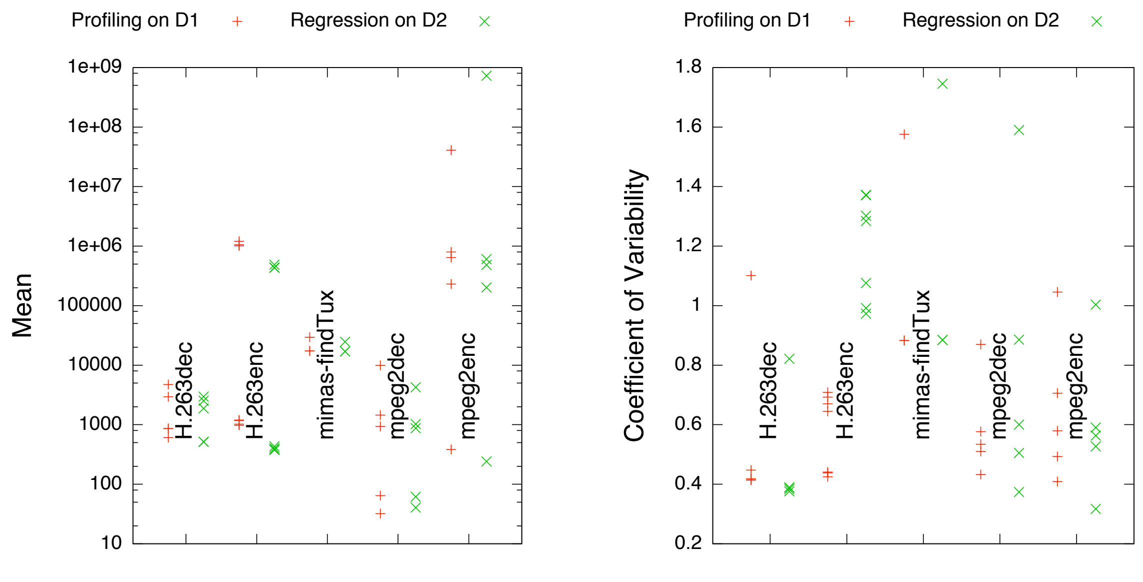


Figure 8.7: *Mean* and *CoV* scatter-plots comparing the profiling `D1` and regression `D2` data sets, constructed using **just the pattern set**.

8.4.1 Case Study: H.263enc

Figure 8.8 shows the profiling pattern set for the H.263enc benchmark, sorted from the most impactful to the least. The `VIM` found for each pattern is shown for the profiling and regression phases. Function names are shown in boxes, and the edge-annotations give the `lexical-id` (lexical position) of the call site of the callee (sink of arrow) within the body of the caller (source of arrow). The italicized number on top of each box gives the number of times the corresponding function was invoked as part of the pattern. A pattern's invocation count corresponds to the invocation count of the function in the left-most box. This is the innermost function of the pattern, and the entire pattern occurs only when the entire call-chain segment occurs on the stack. Therefore, the invocation count of the innermost function is the invocation count of the pattern.

The patterns in Figure 8.8 were automatically discovered by the profile analysis framework with no guidance from the user, and no application or domain knowledge. Yet, these patterns closely mirror conventional wisdom about the parts of video-encoding applications that are the most important with regards to meeting or violating soft-real-time requirements. Motion-estimation related macroblock search-spaces are known to be the most variant parts of video encoding [87], since the search space can be quite large and it is hard to know up front how quickly the search will terminate.

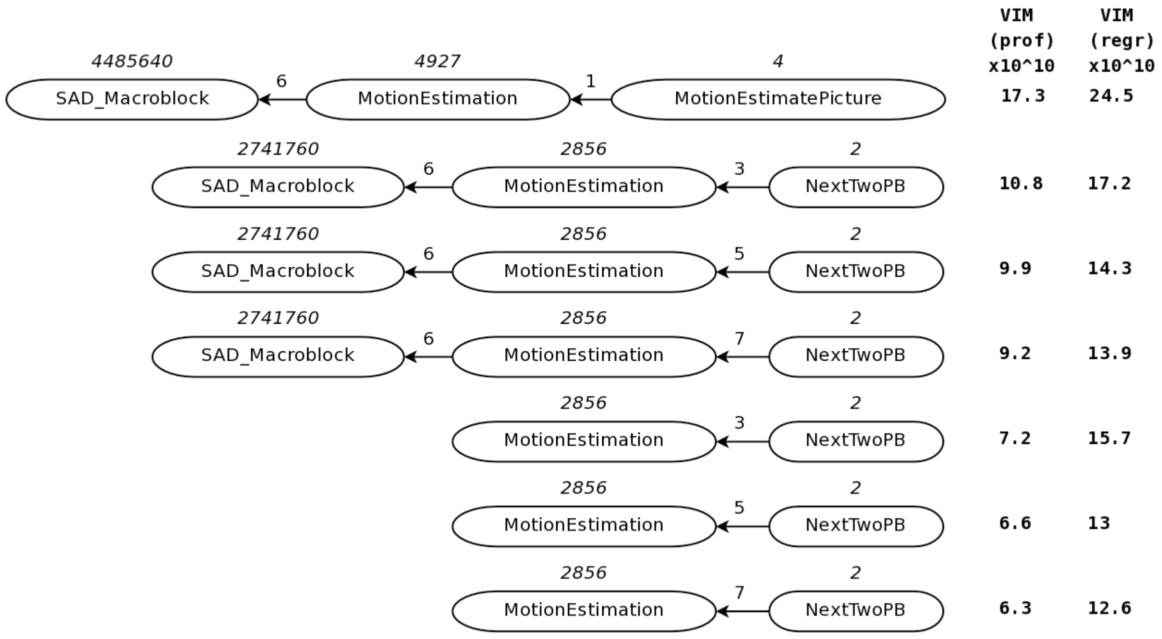


Figure 8.8: Pattern set for H.263enc.

Note that the middle three patterns and the bottom three patterns are identical except for a difference in `lexical-ids`. In both cases, the multiple identical patterns have very similar statistical characteristics (`VIM`s and also from their positions in the scatter plots). These could have been combined into a single pattern in both cases, but our analysis framework distinguishes based on `lexical-ids` within patterns. The downside here is having three patterns where one would suffice, but in general this produces greater resolving power between identical call chains whose behavior varies between call sites, such as with mpeg2enc.

CHAPTER IX

DOMINANT VARIANCE ANALYSIS

9.1 Introduction

One aspect that is not covered by existing profiling techniques is the characterization of the *variation* in execution time exhibited by components in the application. The benefits of such characterization include determining whether an interactive application can be expected to be responsive and detecting if a security application is vulnerable to timing attacks that guess the underlying dynamic control-flow based on the observed variations in its execution time to crack the secret key [88, 89].

Analyzing variance allows *long-range relations* to be revealed between functions not close together in the program graph whose behavior varies in synchrony. Further, analyzing variance rather than hot-spots reveals the functions where highly data dependent processing takes place, a problem difficult to solve using static or dynamic analysis.

Ultimately, we would like to identify groups of functions whose variant behaviors are related, and identify the *dominant modes of behavior* exhibited collectively by each group. In particular, we would like to quantify relationships between the functions where high variance is observed and the other functions that are the *principal underlying causes* of the observed high variance, exposing the *variance contribution structure* of the application. The overarching intent here is to provide application-wide summarization of the variant behavior, including *cause-effect relationships*. Using the generated summary representation, we show how a simple performance model can be derived and used in a controller to realize soft-real-time properties of an application.

9.1.1 Overall scheme and Contributions

In this work we propose a profile analysis framework that elicits dominant patterns of execution variance that are stable across different execution runs. First we profile the application, constructing a Call-Context Tree (CCT) [83] where the nodes collect per-function profiling data. Once the CCT is constructed, we identify instances of highly variant functions and explore their underlying causes within the CCT structure. We extract the relevant structure and function relationships from the CCT into our proposed Variance Characterization Graph (VCG) representation. Once in the VCG representation, we perform further statistical analysis to distill the dominant patterns of behavior and trim out statistically insignificant behavior, presenting the user with a succinct representation of the application's variant behavior and the functions that are its underlying cause.

9.1.2 Contributions

This work makes the following contributions:

- We motivate the optimization potential offered by the characterization of variant behavior in an application's functions.
- We propose a program representation, the Variance Characterization Graph (VCG), that succinctly captures the dominant variant behavior exhibited by the application.
- The VCG representation allows the programmer to easily and unambiguously map observed behavior both to the lexical code locations in the program and to the dynamic call structure of the application.
- We show how the variance metric, specific properties of the dynamic call structure of an application and statistical clustering techniques are used to construct a VCG representation that is statistically stable and provides meaningful results across different data-sets and profiling runs.
- We show how a simple performance model can be quickly devised from VCG and embed controllers into applications to improve the application soft-real-time properties.

The rest of the chapter is organized as follows. We first discuss variance and the factors to be considered when exposing the variance contribution structure of the application. Then we discuss the role of context sensitivity in our dominant variance analysis. This is followed by a description of the VCG representation and the algorithms that generate dominant patterns. Finally, we present evaluations of the VCG representation and the derived controller.

9.2 Variance and its Underlying Source

Our goal is to examine the statistical behavior of the execution times of functions and co-relate them. We first introduce some terms and naming conventions. In Listing 9.1, the functions are arranged in the lexical order G_1 to G_k within the body of F.

Listing 9.1: Example function F

```
void F(int R, int T) {
  G1();
  ...
  if(...)
      G2();
  for(i = 0; i < R; i++) {
      G3();
      S; //local code
      if(T >= 0)
          G4();
      else
          G5();
  }
}
```

```
void G1(...) {
  A1();
  SS; //local code
  A2();
}
void G3(...) {
  B1();
  SSS; //local code
  B2();
}
```

```
void G2(...) {
  if (...)
    D1();
  while (...)
    D2();
}
void D1(...) {
  P();
}
void D2(...) {
  Q();
}
```

Let random variable X represent the execution time of any *single* invocation of F. Let r.v. Y_i represent the *cumulative* execution time of G_i within a single invocation of F. Let r.v. Y_0 represent the *local* execution time of F, that is the execution time of F not spent inside any children calls G_i. Therefore, the following must hold between the observed values of X and Y_i's for any given invocation of F:

$$X = \sum_{i=0}^{k} Y_i \tag{71}$$

Let $\bar{X} = E[X]$ represent the *mean* execution time of F over N invocations of F. Let $\sigma_X^2 = E[(X - \bar{X})^2]$ represent the *variance* in the execution time of F over these invocations. Let $\mathbf{C}$ represent the *covariance matrix* between the random variables Y_0 to Y_k observed over these invocations.

$\mathbf{C}$ is a $(k+1) \times (k+1)$ dimensional matrix, with $C_{i,j} = E[(Y_i - \bar{Y_i})(Y_j - \bar{Y_j})]$, again with the expectation computed over the N invocations of F. A covariance matrix is always *symmetric*, i.e., $C_{i,j} = C_{j,i}, \forall\, i, j$.

Since X is a sum over Y_i, the following holds:

$$\begin{aligned}\sigma_X^2 &= \sum_{i=0}^{k}\sum_{j=0}^{k} C_{i,j} \qquad (72)\\ &= \underbrace{\sum_i C_{i,i}}_{\text{(self terms)}} \quad + \quad \underbrace{2 * \left(\sum_{i<j} C_{i,j}\right)}_{\text{(cross terms)}}\end{aligned}$$

Here we make no assumptions about the correlation between any Y_i and Y_j. Using Eq 72, the self-terms, $C_{i,i} = E[(Y_i - \bar{Y}_i)^2]$ may indicate whether the invocations of G_i within F contribute significantly to σ_X^2. For example, in Listing 9.1, if F is invoked N times with very different values for parameter R, Y_3 will show significant variation over the N invocations of F, causing $C_{3,3}$ to have a large positive value.

In addition, cross-terms are also important when determining the overall contributions of the invoked functions. For example, if F is invoked with both positive and negative values for parameter T, then both Y_4 and Y_5 will show significant variation, resulting in large positive $C_{4,4}$ and $C_{5,5}$ terms. However, the cross-terms $C_{4,5} = C_{5,4}$ will have large-magnitude negative values, indicating a strong negative correlation and canceling out each other. *Therefore, in order to determine which G_i's are the major causes of the variance in F, it is not sufficient to just determine if the corresponding $C_{i,i}$ terms contribute significantly to σ_X^2, but also need to examine if other G_j's are diminishing or enhancing the contribution through $C_{i,j}$ terms.*

Let us explicitly label the metrics to identify what function they correspond to — $\bar{X}^F$, $\sigma_{X^F}^2$ and $\mathbf{C}^F$ for F, and $\bar{X}^{G_1}$, $\sigma_{X^{G_1}}^2$ and $\mathbf{C}^{G_1}$ for G_1. Our interest in studying the variant behavior of a function F is to determine if other functions are the *root underlying source* of the variance in F.

Consider the following scenarios:

- Variations in G_1's execution time may actually reflect the variations in A_1's execution time, making G_1 simply a *transmitter* of A_1's variance to F. A_1 would then be the *underlying source* of F's variance, even though F does not invoke it directly. Further analysis may reveal that A_1 itself is merely a transmitter of variance originating further down the call-chain.

- Alternatively, the local code statements denoted by SS may be causing G_1's behavior to be variant, with A_1 and A_2 only playing a minor role. This would establish G_1 itself as an underlying origin of F's variance.

- Another alternative is that none of SS, A_1 and A_2 have large variances of their own. But if their behavior varies in synchrony, then their limited variance gets amplified. Again, this would establish G_1 as a root underlying source of F's variance.

To determine which of the above three scenarios apply, we use the fact that $C^F_{1,1} = \sigma^2_{X^{G_1}}$ (by definition), and directly compare the values of the $C^{G_1}_{i,j}$ terms against $\sigma^2_{X^F}$. This lets us determine if $\{A_1\}$, $\{A_2\}$ or $\{A_1, A_2\}$ could replace G_1 as a significant contributor of variance to F.

Unlike G_1, G_3 is called within a loop in F. Due to the loop, unfortunately, the availability of the $C^{G_3}_{i,j}$ terms does not permit a similar analysis to be possible for determining if B_1 or B_2 are the underlying sources of F's variance, with G_3 merely being the transmitter. The primary impediment is that a relation between $C^F_{3,3}$ and $\sigma^2_{X^{G_3}}$ cannot be established in general. We can get around this limitation by treating the body of G_3 as *implicitly inlined* into F. Then B_1 and B_2 will directly get terms in the $\mathbf{C}^F$ matrix instead of terms for G_3. The effect of the SSS block of statements would be absorbed into the Y^F_0 variable. In theory, we could recursively inline any of the G_i functions to any depth in order to determine the underlying sources of variance despite the presence of loops and conditionals around call-sites. However, each additional depth of inlining the call-chains originating at G_i requires the corresponding $\mathbf{C}$ matrix to be reconstructed, requiring a fresh pass over the profiling data. Additionally, the size and cost of constructing $\mathbf{C}^F$ grows as a square of the number of leaf call-sites in F after inlining. For any given F, the large number of combinations (for each depth, along multiple call-chains originating under F), and the growing cost of constructing $\mathbf{C}^F$ for each combination precludes an exhaustive examination of possibilities via inlining. Instead, we rely on heuristics. For example, is the total execution time of the next level inlined-function too small compared to F to contribute much variance? Is there a sufficient number of inline candidates identified across the CCT to make another profiling pass worthwhile?

If F has been implicitly inlined along various call-chains to depths where its body now contains call-sites for functions $A_1, A_2, ..., A_i, ...$, we need following modifications to $\mathbf{C}$.

- Variable $Y^{F|A_i}$ refers to the cumulative execution time spent in A_i during a given invocation of F. This is analogous to Y_i referring to the cumulative execution time of immediate call-site G_i.

- $\mathbf{C}^{F|A_1,A_2,...}$ refers to $\mathbf{C}^F$ that has been modified to remove terms for the G_i's that have been inlined away, and has new entries added for $A_1, A_2, ...$ That is, the new $Y^{F|A_i}$ variables get entries alongside the remaining Y_j's.

- $\mathbf{C}^{F|A_1,A_2,...}_{i,j}$ refers to terms between an unaffected Y_i and Y_j, i.e., it's the same as $\mathbf{C}^F_{i,j}$. However, $\mathbf{C}^{F|A_1,A_2,...}_{A_i,j}$, $\mathbf{C}^{F|A_1,A_2,...}_{i,A_j}$, and $\mathbf{C}^{F|A_1,A_2,...}_{A_i,A_j}$ refer to new terms between $Y^{F|A_i}$ and Y_j, between Y_i and $Y^{F|A_j}$, and between $Y^{F|A_i}$ and $Y^{F|A_j}$, respectively.

9.2.1 Classifying Variance

We use Chebyshev's inequality [85], given below, to determine if a function F's behavior can be considered highly variant over its N invocations. **Chebyshev's inequality** establishes conservative probability bounds on the statistics of a given collection of data samples *while making no assumptions about the underlying probability distribution that generated the data.*

$$Pr(|X - \bar{X}| \geq t\sigma) \leq \frac{1}{t^2} \tag{73}$$

We define a node in the CCT to be *high-variant* if its execution time cannot be guaranteed to lie within 200% of its mean with at least 96% probability. This implies $\frac{1}{t^2} = 1 - 0.96 = 0.04$ and $t\sigma = 2 \times \bar{X}$. Therefore $\frac{\sigma}{\bar{X}} \geq 0.4$ becomes the condition for high-variance. Consequently, we use the *Coefficient-of-Variability* metric for classifying the variant nature of nodes: $CoV = \frac{\sigma}{\bar{X}}$. Similarly, a CCT node is labeled *low-variant* if its execution time lies within 10% of its mean with at least 96% probability, with the test $CoV \leq 0.01$. Note that the variance characterization of CCT nodes is done based on the final profile statistics after the CCT has been fully constructed, not progressively as samples are collected.

9.3 Context Sensitivity of Behavior

In many parts of a program, behavior may be determined largely by the data or parameters passed from the calling context. We would like VCG to distinguish the variant behavior of a function if it differs based on the calling context. Otherwise, we would like VCG to provide the most succinct representation of a function's behavior possible. Context-sensitive analysis offers several benefits, including: specialization of functionality to context, either by the programmer or the compiler (code version selection [90]); detecting behavioral bugs (such as real-time deadline violations) that predominantly show up in some contexts of invocation of function F and not in others.

A context-aware analysis scheme would determine *i) whether* context affects behavior, *ii) what aspects* of the context affect behavior, and *iii)* how does behavior differ across contexts. Once the differing behaviors are identified for a function F of interest, we use the notion of Minimal Distinguishing Call-Chain Context (MDCC) [81, 82] to succinctly distinguish the occurrence of one type of F's behavior from another in the program call-graph. For now, consider a VCG pattern to be simply a tree of VCG nodes. Each VCG node n in a VCG pattern P will have associated with it a *set of MDCCs*, called the Call-Context Set for n, denoted $CCS(P, n)$, which identify the calling context under which the behavior of n in P is expected to occur. $CCS(P)$ consists of MDCCs for the root node of P, with the MDCCs needing to distinguish across all other VCG patterns which have the same function F in their root node. In contrast, an internal VCG node n only needs to distinguish itself from its siblings under their common parent VCG node, and the VCG tree structure often suffices for such disambiguation, typically making the $CCS(P, n)$ information unnecessary for disambiguating between internal nodes. Further, $CCS(P)$ contains only the shortest call-chain segments needed to distinguish against other patterns P_2, P_3, ... rooted on the same function. We use the MDCC-construction algorithms from [81, 82] to construct $CCS(P)$ and $CCS(P, n)$. The next section illustrates call-context sets.

```
void H(){
  L(…);    //lexical site [0] L1
  K(…);    //lexical site [1]
  J(…);    //lexical site [2]
  for(… i<Num …)
    L(…); //lexical site [3] L2
  U(…);    //lexical site [4]
}

void L(…) {
  If (…) {
    if (…)
      A(…);     //lexical site [0]
    else
      D(…);     //lexical site [1]
  }
  else N(…); //lexical site [2]
}
```

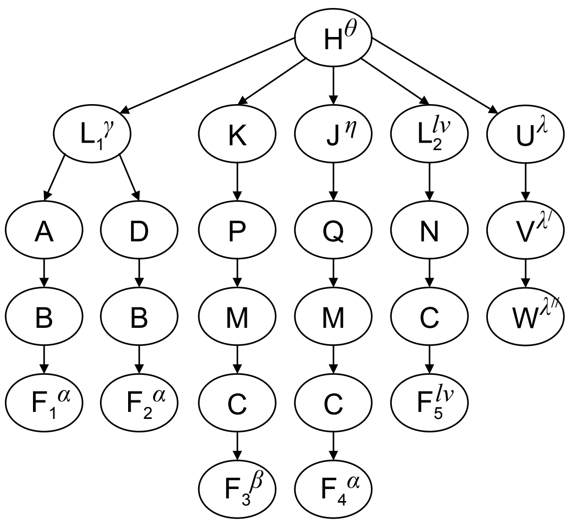
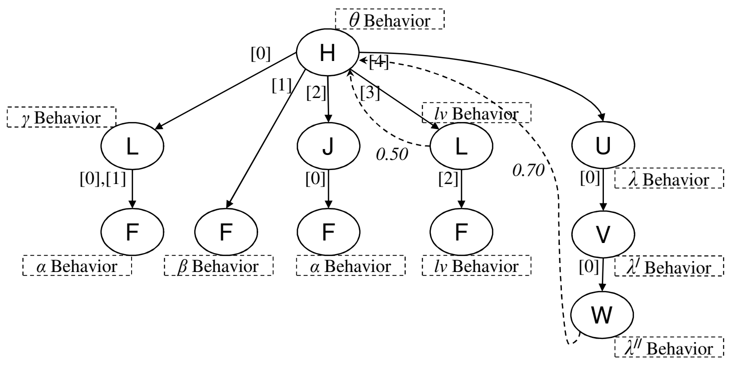


Figure 9.1: Example function H and its annotated CCT.

9.4 Constructing VCG from CCT

Figure 9.1 shows a subtree of the application CCT under a function H. Nodes with identical names differ in terms of the call-chain used to invoke them under H. Henceforth, let F_i be used to label the nodes for the same function F. We will now use N^{F_i}, $\bar{X}^{F_i}$, $\sigma^2_{X^{F_i}}$, $\mathbf{C}^{F_i}$ and CoV^{F_i} to refer to metrics on the corresponding *node instances* of F, and similarly for node instances of other functions.

The nodes annotated with Greek letters represent the occurrences of high-variant behavior. Whenever a function F has nodes exhibiting high variance (F_1, F_2, F_3 and F_4), it becomes important to *contrast* against the occurrences of low-variance behavior of the same function F (F_5) because it is crucial for *inferring the unique circumstances* under which each behavior occurs. Here, each greek letter identifies a distinct type of high-variant behavior, which is determined by comparing the $\left(\bar{X}, CoV\right)$ tuples for the corresponding nodes. All occurrences of low-variant behavior are annotated with lv. Functions that never exhibit high-variant behavior do not need their node instances distinguished for the purpose of variance analysis. Hence, their CCT nodes are not shown with subscript indices (A, B, M, etc.).



VCG Node	*Call-Context-Set*
H[0]-L	{[0]}
H[0]-L[0][1]-F	{[0]-A[0]-B[0], [1]-D[0]-B[0]}
H[1]-F	{[1]-K[0]-P[0]-M[0]-C[0]}
H[2]-J	{[2]}
H[2]-J[0]-F	{[0]-Q[0]-M[0]-C[0]}
H[3]-L	{[3]}
H[3]-L[2]-F	{[0]-N[0]-C[0]}
H[4]-U	{[4]}
H[4]-U[0]-V	{[0]}
H[4]-U[0]-V[0]-W	{[0]}

Figure 9.2: A VCG Pattern and its CCS representing variant behavior under H.

While CCT provides a representation that predicts the expected behavior of functions based on the call-chain, a few challenges from the point of view of facilitating *program understanding* become evident:

- **How to preserve structural relationships?** The CCT does not reveal any relationships between the behaviors of a group of functions. For example, it is not evident from the CCT that F exhibits behavior α when invoked under L exhibiting behavior γ, and F exhibits lv when invoked under L exhibiting lv.

- **How to eliminate/minimize redundant information?** The majority of the space in the CCT is occupied by long call-chain that do not themselves describe behavior. In fact, the behaviors of F can be simply distinguished using L_1, L_2, K and J in this example, with the added advantage of distinguishing the behavior closer to its *cause* (i.e., knowing which of L_1, L_2, K or J is invoked will determine F's behavior).

We propose the Variance Characterization Graph (VCG) representation to meet these challenges. Figure 9.2 shows the VCG representation of the variant behavior under H in a single VCG pattern. In general, there would be multiple VCG patterns extracted from the CCT of a program. The behavior α of CCT nodes F_1 and F_2 gets summarized into a single *VCG node*. The two lexical instances of L (on edges tagged [0] and [3] under H) along with J and the *absence* of K distinguish the spectrum of F's behavior. At the same time, the variant behavior of J and L is also fully characterized. The CCS for each

VCG node is shown. For internal VCG nodes a CCS only gives the call-chain paths to the VCG node from its parent VCG node. Each VCG node is tagged with its corresponding behavior. It is worth noting that *i)* only those functions that show variant behavior in at least *some* instances show up as VCG nodes (let's refer to them as *used functions*); *ii)* there is little to no loss in the ability to *locate* every distinct behavior for each used function in the VCG tree. The location of behavior is implicitly disambiguated by the tree structure itself, with a need to examine CCSs of children nodes only if a VCG node has more than one identically-named immediate-children VCG nodes exposing distinct behaviors.

Since VCG retains call-structure, we can now superimpose long-range variance contribution relationships over this structure. In Figure 9.1, let's say that the analysis described in Section 9.2 determined the *lv* L and W as the underlying sources of variance for H. This creates $H \rightsquigarrow L$ and $H \rightsquigarrow U \rightsquigarrow V \rightsquigarrow W$ as *linear variance-contributing segments* (or *linear segments* for short) in the VCG (Figure 9.2). The head of the linear segments, H, is the *target* high-variant node to which variance is being contributed by the *source* nodes L and W of the respective linear segments. The nodes *internal* to the linear segment (U and V) are merely the *transmitters* of underlying variance. Even a low-variant L may contribute variance to H, say if it is invoked inside a loop with a loop-count that varies over invocations of H. Strictly speaking, the source of the variance is the enclosing loop-block rather than L, but we make a design decision to restrict our analysis and the VCG representation to just functions for tractability dashed back-edges from L to H and W to H graphically capture the *variance contribution relationships* corresponding to the two linear segments and demarcate their spans. The $L \rightarrow H$ back edge is annotated with the *variance contribution fraction* $cf = 0.50$, and the back edge $W \rightarrow H$ with $cf = 0.70$. cf captures how much of the target's variance (i.e., H's) is contributed by the source of the back edge (i.e., L and W). Formally, $cf \leftarrow \mathbf{C}^{H|W}_{W,W}/\sigma^2_H$ for the $W \rightarrow H$ back edge. Note that cf may exceed 1 and the sum of cf's from multiple back edges to H may exceed 1. This is possible because of potential negatively correlated variant behavior under H (e.g., out-of-phase variations in invocations of L and U).

Scope of a VCG Pattern. The single VCG pattern shown in Figure 9.2 could conceivably have been multiple VCG patterns — one for H, one each for F with α, β and lv behaviors, one each for L with γ and lv behaviors, and so on. However, when the specific behaviors observed occur under the call-context of H, constructing the larger pattern retains the context. Note that without the context of H, the three instances of α behavior in the CCT (F_1, F_2 and F_4) could have been merged into a single VCG pattern. However, due to the need to capture the variant CCT nodes L_1 and J under H's context, the pattern could only merge F_1 and F_2. Additionally, the presence of linear segments also dictates the scope of a pattern (retain W under H).

ALGORITHM: Extract VCG patterns from CCT. After profiling, the CoV test is used to tag CCT nodes as high variant or low variant, ignoring nodes with too few invocations for statistical significance (default, < 6) or contributing an insignificant fraction of the program execution time (default, $< 0.02\%$). If a function F is *used*, i.e., at least one CCT node for F is tagged high variant, then all significant nodes of F (i.e., with $\geq 0.02\%$ of program execution time) must also be extracted, and tagged as either low variant (if indicated by the CoV test) or as *contrast* otherwise. Variance contributions of underlying nodes to their ancestor CCT nodes are examined, possibly invoking additional profiling passes to reconstruct the **C** matrices after each level of implicit inlining. Finally a pre-order traversal of the CCT extracts the initial set of VCG patterns — finding a tagged node starts a VCG pattern, and the VCG pattern extends to descendant nodes if they contribute significant variance to an ancestor VCG node in the same pattern (either directly or through a linear segment marked in the CCT).

9.5 VCG Pattern

Here we summarize the semantics of a VCG pattern in terms of what it represents and what can be inferred from it.

Tree Structure and Call-Context-Sets. A VCG pattern consists of a tree structure with annotated back-edges. The nodes represent functions invoked in the dynamic call structure of the application. A forward-edge establishes *context*: the function-call for the child VCG node was invoked during the execution of the function-call for the parent VCG node. The child function-call may have been invoked indirectly by the parent via a long chain of intermediate calls that have been omitted from the VCG (*unused* functions, Section 9.4). Each VCG node has an associated Call-Context-Set (CCS) providing the call-chain path-segments to the node's function-call from the parent VCG node's function-call. The node's CCS helps disambiguate the specific call-chain contexts under which the behavior associated with the node occurs (though usually the VCG tree structure itself sufficiently disambiguates context). The CCS for the root VCG node consists of MDCC segments to disambiguate under `main` the function-call paths to the VCG pattern.

Contribution Structure. A back-edge b establishes *variance contribution*, indicating that the source node is the root underlying cause of the execution-time variance observed in the target node. The zero or more nodes along the forward-edges from the target to the source of b (the linear segment of b) are the transmitters of the source's variance to the target.

Node Properties. Each VCG node can be one of three *node types*: Task, Contributor or Contrast. A Task is a node at which high-variant behavior is *principally observed*, rather than just being a transmitter of variance to an ancestor Task. A Task F may or may not have a back-edge to another Task J which is F's ancestor in P. A Contributor is a non-high-variant node that is nonetheless the underlying source of variance to an ancestor Task J via a back-edge. A Contrast node for F is a non-high-variant node that is not a Contributor. The purpose of a Contrast node is to *i)* help isolate the specific call-contexts under which F is high-variant, and *ii)* capture the range of behavior possible for F. Each VCG node maintains invoke count, mean and variance statistics.

Contribution Properties. A Task node may have incoming variance contribution back edges. A back edge is annotated with statistics extracted from the appropriate term in $\mathbf{C}$ to capture what fraction of the target node's variance is being contributed by the source of the back edge, referred to as the *contribution fraction.*

9.6 Dominant Behavior

The CCT is constructed with `main` as the root node. Therefore, the analysis described in Section 9.4 will find a *forest of VCG patterns*, as `main` itself is invoked only once and therefore invariant. In our experiments, the number of patterns detected in the CCT could range from a few dozen to several hundred. Some patterns may have a VCG node F that exhibits unusually high variance (i.e., large CoV^F), or F has an unusually large total execution time over the N^F invocations in the node (called *weight* W^F, where $W^F = N^F * \bar{X}^F$). Either one of these characteristics makes this node *dominant*, and makes it important that the containing pattern P be presented prominently in the analysis results. It is possible to have multiple patterns $P_1, P_2, ..., P_T$ with identical structure, such that F is not dominant in any of them. But the behavior of F becomes dominant if all of these structurally identical patterns are *considered collectively*, called *common recurring pattern of behavior*. Therefore, we need to be able to *distill the dominance of recurrent behavior* from across the application's CCT. We distill dominance by merging identical VCG patterns and creating a *higher-level VCG pattern* that retains the same structure while accumulating the statistics of the individual *lower-level VCG patterns* $P_1, P_2, ..., P_T$.

Summarization. In general, the desire to summarize information into as few patterns as possible needs to be counter-balanced with a need to preserve the distinctiveness of behaviors. We refer to this as the *summarization vs precision* tradeoff. To allow for maximal summarization, we need the ability to merge patterns that are *just similar rather than identical* in structure and corresponding metrics. When considering two patterns for merging, differences can arise at multiple points between them: *i)* a subtree present in one may be absent in the second, *ii)* the statistics associated with corresponding nodes in the two patterns may differ, and *iii)* the variance contributed by corresponding linear segments may differ. We take a weighted sum of these differences, and construct a single scalar *merge-cost* which encompasses all these factors. The merge-cost is compared against a threshold derived from the summarization-pressure to determine if the given two patterns are merged.

However, there are certain **structural conditions** that must be satisfied for a merger to be even considered. The conditions check whether the two patterns are structurally *compatible* (as opposed to identical) both in the tree structure that provides context, and in the contribution structure imposed on the tree structure (the linear segments, identified by the back-edges). When the compatibility conditions are satisfied, the resulting merged pattern meaningfully summarizes the combined behavior over the two patterns, including over subtrees that are present in only one lower-level pattern.

The summarization vs precision tradeoff is best served if the analysis achieves the maximum precision possible under any given setting of a **summarization-pressure parameter**, τ_S. We choose to use *hierarchical agglomerative clustering* [91] as the technique for merging patterns and distilling dominant behavior. Hierarchical agglomerative clustering treats each pattern as a point. The distance between every pair of points is computed (we use the merge-cost as the distance measure). A pair of points a and b with the smallest distance $d_{a,b}$ are merged, provided $d_{a,b} \leq \tau_S$. This *consumes* points a and b and produces a new point m as a result of the merger. Other *unconsumed* points c and d can then be similarly merged in the order of smallest distance. This process consumes a *layer* of points and produces a new layer of points. Every pair of points in the last layer is either incompatible (based on the compatibility conditions for merging two patterns) or the merge-cost exceeds τ_S.

Prioritization and Trimming. The patterns in the final layer are subsequently prioritized and trimmed. Prioritization sorts the patterns in the order of their dominance, where a pattern's dominance is computed as a sum of the root node's dominance and the recursive dominance of the subtrees. Prioritization presents the most important patterns first to the user. The next step of trimming is optional and eliminates entire patterns, or subtrees within patterns, that are substantially less dominant compared to the ones retained. We use the *Variance Impact Metric* introduced by Kumar et al. [82] (`VIM` $\triangleq \sigma N$ for a Task node, $= 0$ for other node types) as a node's dominance, computed recursively for a pattern/subpattern as described above. A **trim-threshold parameter** $0.0 \leq \beta < 1.0$

eliminates patterns (and its subpatterns) whose `VIM` is less than β times the `VIM` of the highest-`VIM` pattern/subpattern at that level, *and*, whose weight is less than β times the weight of the highest-weight pattern/subpattern at that level. The weight condition ensures that important low-variant patterns/subpatterns are retained for contrast.

Generality of the Dominant VCG patterns. Summarization, prioritization and trimming make the VCG results robust. If only the most re-inforced and weighty behavior is presented to the programmer, then chances are good that this behavior derived over a *profiling data-set* will also be the most dominant behavior that occurs over any future runs of the application on *regression data-sets*. Strictly speaking, the exact same metric values from profiling are unlikely to re-appear during regression (as with any profiling technique). However, we can expect that essentially the same pattern structures, in essentially the same order of relative dominance will be found in the regression runs. On the other hand, disabling or minimizing trimming (setting $\beta \approx 0$) will allow *data-set specific behavior* to become apparent. When comparing two sets of *untrimmed* VCG results generated for data-sets with differing characteristics, we can expect to find significant structural and ordering differences in the less dominant VCG patterns. These expectations are borne out by our experimental validation on real-world applications.

Structural Compatibility. The pattern merge algorithm first attempts to recursively merge the structure of two VCG patterns P_1 and P_2 into a candidate merged pattern P_m. P_1 and P_2 need not have identical structure: subtrees may exist in one but not in the other, linear segments may differ. Correspondence between P_1 and P_2 is established in a pre-order traversal: corresponding nodes in P_1 and P_2 are located (if present in both) by examining both the corresponding VCG trees and the corresponding CCS's, the corresponding node-types must match, the merged CCS computed, and the contribution structure merged if not incompatible. The contribution structure consists of back-edges superimposed on a VCG pattern's tree structure. The contribution structure mismatches if the backedges coming from P_1 and P_2 create an *overlap*. Recall that the semantics of a VCG back edge are that the source node X is the underlying cause of the variance observed in an ancestor node

Y, and all the nodes between X and Y are merely transmitters of this variance. If P_2 introduces a back edge with source as node Z lying between X and Y, the transmitter semantics for Z in P_1's back edge are violated — the two back edges represent incompatible behavior. To summarize, a mismatch in either *i)* the node-types for corresponding nodes (if present in both patterns), or *ii)* in the contribution structure, precludes a merge. Note that alternative techniques, such as approximate graph matching, may be used in place of our recursive merge algorithm, so long as matching node-types and compatible contribution structures are somehow ensured.

Merge cost. If patterns P_1 and P_2 are found to be structurally compatible, a merge-cost is computed to determine the *degree of similarity of the behavior statistics* expressed in the two patterns. The merge cost consists of the following factors — *i)* the degree of similarity of the node statistics, and *ii)* the degree of similarity of the contribution statistics of corresponding back edges.

The pattern merge-cost is *computed progressively* as a sum of the *node merge-costs* in a pre-order traversal of P_m and normalized by the number of nodes in P_m visited so far. If the normalized progressive merge cost exceeds the summarization pressure parameter τ_S at any point in the pre-order traversal the merge of P_1 and P_2 into P_m is rejected. Use of a normalized progressive merge cost allows us to prioritize mismatches near the root of P_m (recall that the root provides the context under which the subtrees exist), while at the same time allows *average similarity* of node statistics to be the ultimate factor determining the merge.

At a high-variant node n in P_m, the *Kolmogorov-Smirnov difference measure D* is used to determine the similarity of behavior of n in P_1 and P_2. The corresponding means and standard deviations are used to construct Gaussian distributions for $P_1.n$ and $P_2.n$. The choice of Gaussian is for simplicity to reconstruct the distribution; a more sophisticated approach would retain more detail about the shape of the distribution rather than just the mean and standard deviation — for example, a Gaussian Mixture Model. The KS D

metric is used because it can compute the degree of dissimilarity between any two probability distributions without making any assumptions about the nature of the underlying distributions and is known to be a robust metric in practice. $0.0 \leq D \leq 1.0$, where $D = 0.0$ indicates identical distributions, and $D = 1.0$ maximal dissimilarity (say, between two non-identical Dirac delta distributions). The merged statistics for $P_m.n$ can be computed using the invocation counts, means and standard deviations for $P_1.n$ and $P_2.n$. Note that the merged statistics can be computed *precisely*, that is they are mathematically identical to re-computing statistics over the combined X samples from $P_1.n$ and $P_2.n$. $D = 0$ is assumed when merging non-high-variant nodes, as the purpose of such nodes is to contrast against the behavior of high-variant nodes and identify underlying causes of variance; the behavior statistics of interest are really those of the high-variant nodes.

Each contribution back-edge b has a contribution-fraction statistic associated with it. Given the variance of the target node of b, we can calculate the actual variance being contributed and merge the statistics from $P_1.b$ and $P_2.b$ (weighed using the invoke-counts of the target nodes) to determine the merged contribution-fraction. The variance in the contribution-fractions (weighed by their respective invoke counts) is used a measure of dissimilarity for merging $P_1.b$ and $P_2.b$. For nodes with incoming back-edges, the node merge-cost is the average of the node statistics D metric and the average contribution-fraction merge-cost for all incoming back-edges.

Appendix D provides an implementation-level discussion of the merge algorithms and mathematical details.

9.7 Dominant Variance Analysis

In summary, Dominant Variance Analysis (DVA) consists of the following steps:

Step 1 **Profiling and CCT Construction**: In our experiments, we use LLVM [84] to instrument all function-call entries/exits to update *dynamic-instruction-count* (DIC as an *approximation* of execution time), and dump to profile files, as per [82], in the profiling phase. But in reality, our analysis framework works with any profiling and CCT construction methodology. The additional annotations needed by our analysis (mean, variance, etc.)

are straightforward to incorporate in any CCT construction scheme. Once the CCT is constructed, high-variant nodes are identified based on the CoV test in Section 9.2.1.

Step 2 **Identifying long-range underlying sources of variance on CCT**: Section 9.2 describes how additional profiling passes over the profile sequence allow the identification of low-level functions that contribute significant variance to functions several levels up in the CCT.

Step 3 **Initial Extraction of VCG patterns**: Once the annotated CCT has been constructed, the node variances characterized and the long-range underlying-contribution relationships identified, certain CCT nodes get extracted as VCG nodes of types Task, Contributor and Contrast (Section 9.5). Hierarchy in the CCT leads to the forward tree-edges in the VCG pattern. Contribution relationships become back edges between relevant VCG nodes. These *initially extracted* VCG patterns form the *base-layer of VCG patterns* on which further merging will be attempted subsequently.

Step 4 **Merging within and across VCG patterns**: The algorithms in Section 9.6 attempt merging *internally* within each base-layer VCG pattern, then perform merging *across* the subsequently produced patterns. The summarization pressure parameter τ_S controls whether the final VCG results are intended for generality to regression runs (high τ_S) or for extracting data-set specific details (low τ_S).

Step 5 **Prioritization and Trimming**: Using the `VIM` metric, the final merged layer of VCG patterns are sorted in priority-order. The β parameter sets the trimming threshold to eliminate non-dominant and non-weighty results (Section 9.6). Aggressive trimming ($\beta \approx 1.0$) will likely leave only the most dominant patterns that generalize best. Low levels of trimming ($\beta \approx 0.0$) will preserve more details, including "noise" specific to the profiling data set.

Once the initial VCG patterns have been extracted from the CCT, the user can repeatedly invoke Steps 4-5 of the Dominant Variance Analysis with different values for τ_S and β to gain different types of understanding about the application. Whereas Steps 1-2 can potentially be very time consuming as they involve scanning the entire profile data set, Steps 4-5 can be repeated in less than a second as they work with very sparse information.

9.8 Experimental Evaluation

The emerging importance of interactive media applications on consumer desktops, embedded systems and surveillance was a key motivation towards our developing the Dominant Variance Analysis. We choose four benchmarks from MediaBench II [70], two video encoders and two decoders: `mpeg2enc`, `mpeg2dec`, `h263enc` and `h263dec`; four vision algorithm benchmarks from San Diego Vision Benchmark Suite [92]: `svm`, `sift`, `stitch` and `tracking`; one benchmark from [75], a planar object detector: `ferns`; and one benchmark from [93], a real-time face tracker: `facetrack`. The number of functions these benchmarks have ranges from 89 to 829, and 227 on average. We ran all profile generation experiments and the Dominant Variance Analysis on an Intel Q6600 system (quad-core) clocked at 2.40GHz with 2GB of RAM. The input data-sets used for profiling were reference videos (for decoders) and corresponding decoded raw-image sequences (for encoders).

dolbyrain, hockey1_cif and baikonur_r7_over_flight are standard videos used to characterize video codecs. qos is a trailer of the *Quantum of Solace* action movie transcoded to two resolutions (320x240 and 640x480), since pixel-count dramatically affects execution-times [70]. We used between 100–300 frames of the video sequences for profile generation. fullhd, vga, cif and qcif are data sets shipped along with San Diego Vision Benchmark Suite. mousepad is a default test video clip for `ferns`, while epfl is multi-camera pedestrian videos from [94]. adam_sandler and al_gore are two video clips from YouTube celebrities face tracking and recognition data set.

All the steps (Section 9.7) of the Dominant Variance Analysis were implemented in python. Step 1 took between 1400–5800 seconds (over different benchmarks and data-sets). Step 2 took between 6000–16500 seconds. Steps 3–5 cumulatively consumed less than 1 second since they operated only on the constructed CCT and not on the profile sequence. We expect Steps 1 and 2 to be substantially faster if implemented in C/C++.

Table 9.1 shows the sizes of the representations at various stages of the analysis, with different settings of the summarization-pressure parameter τ_S (over its full range: 0.0–1.0)

Table 9.1: Number-of-nodes / Number-of-patterns: initially extracted, versus, after merging on τ_S without/with trimming

			Initially	**No trimming** ($\beta = 0$)						**With trimming** ($\beta = 0.1$)					
		CCT	**Extracted**	τ_S						τ_S					
Benchmark	**Data-set**	**size**	*(unmerged)*	0.01	0.2	0.4	0.6	0.8	1.0	0.01	0.2	0.4	0.6	0.8	1.0
mpeg2enc	qos320x240	707	78/6	42/5	31/5	25/5	25/5	25/5	25/5	32/1	23/1	17/1	17/1	17/1	17/1
	qos640x480	695	74/3	39/3	28/3	21/3	21/3	21/3	21/3	31/1	22/1	17/1	17/1	17/1	17/1
	dolbyrain	718	75/3	30/3	29/3	23/3	23/3	23/3	23/3	23/1	23/1	17/1	17/1	17/1	17/1
mpeg2dec	qos320x240	890	400/186	42/11	35/8	35/8	35/8	35/8	35/8	15/4	16/5	16/5	16/5	16/5	16/5
	qos640x480	920	407/184	55/12	49/10	49/10	49/10	49/10	49/10	16/4	17/5	17/5	17/5	17/5	17/5
	dolbyrain	898	394/181	42/7	41/6	41/6	41/6	41/6	41/6	14/2	15/3	15/3	15/3	15/3	15/3
h263enc	hockey1_cif	747	207/71	50/7	44/5	44/5	44/5	44/5	44/5	6/3	2/1	2/1	2/1	2/1	2/1
	baikonur_r7	745	45/12	27/9	18/6	18/6	18/6	18/6	18/6	8/4	2/1	2/1	2/1	2/1	2/1
h263dec	hockey1_cif	294	125/52	40/9	35/9	35/9	35/9	35/9	35/9	8/2	8/2	8/2	8/2	8/2	8/2
	baikonur_r7	296	134/55	51/13	43/13	35/12	35/12	35/12	35/12	22/4	23/4	15/3	15/3	15/3	15/3
ferns	mousepad	676	35/9	35/9	35/9	35/9	35/9	35/9	35/9	6/4	6/4	6/4	6/4	6/4	6/4
	epfl	676	35/9	35/9	35/9	35/9	35/9	35/9	35/9	6/4	6/4	6/4	6/4	6/4	6/4
svm	cif	522	35/20	21/6	19/6	19/6	19/6	19/6	19/6	8/1	6/1	6/1	6/1	6/1	6/1
	qcif	522	35/20	21/6	19/6	19/6	19/6	19/6	19/6	8/1	6/1	6/1	6/1	6/1	6/1
sift	fullhd	260	16/12	10/7	9/6	9/6	9/6	9/6	9/6	4/2	5/3	5/3	5/3	5/3	5/3
	vga	263	14/11	6/4	6/4	6/4	6/4	6/4	6/4	4/2	4/2	4/2	4/2	4/2	4/2
stitch	fullhd	314	8/8	4/4	4/4	4/4	4/4	3/3	3/3	3/3	3/3	3/3	3/3	2/2	2/2
	vga	314	8/8	4/4	4/4	4/4	4/4	3/3	3/3	3/3	3/3	3/3	3/3	2/2	2/2
tracking	fullhd	522	8/8	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
	vga	525	8/8	3/3	3/3	3/3	2/2	2/2	2/2	3/3	3/3	3/3	2/2	2/2	2/2
facetrack	adam_sandler	6183	10/2	10/2	10/2	10/2	10/2	10/2	10/2	5/1	5/1	5/1	5/1	5/1	5/1
	al_gore	6163	10/2	10/2	10/2	10/2	10/2	10/2	10/2	5/1	5/1	5/1	5/1	5/1	5/1

and with trimming enabled and disabled. $\beta = 0.1$ performs a relatively non-aggressive trimming (Section 9.6), only eliminating patterns/sub-patterns at a level that are at least an order to magnitude (10×) less dominant than the most dominant pattern/sub-pattern at that level. The constructed CCT is a tree rooted at `main` with approx. 300–6000 nodes (across different benchmarks and data-sets). The initial layer of VCG patterns extracted from the CCT range from 10 nodes in 2 patterns to 407 nodes in 184 patterns. Now consider the final merged VCG patterns produced. With *trimming disabled* (under $\beta = 0$), only the summarization pressure can force a reduction in the number of VCG nodes and patterns. The achievable reduction is limited by the structural compatibility (Section 9.6) which must be satisfied for merger to be possible, regardless of summarization pressure. $\tau_S = 0.01$ represents *very low summarization pressure* (recall, overall merge-cost of two patterns is the *average* of all the D difference measures between corresponding nodes/back-edges, and that $0.0 \leq D \leq 1.0$). Since $\tau_S = 0.01$ shows substantial reduction in number of nodes and patterns for all benchmarks, this suggests that the initial extracted layer of VCG patterns had a large number of patterns that matched closely in structure and in corresponding annotated statistics (i.e., behavior). This result demonstrates that the CCT representation had *significant recurrent patterns of behavior* that were merged by our analysis to *distill the dominance of the associated recurrent behavior*. Under larger τ_S relatively limited additional merger occurs, producing *stable* VCG results in most cases with $\tau_S \geq 0.2$, and in all cases with $\tau_S \geq 0.4$.

The trends discussed with trimming disabled also hold true when trimming is enabled (under $\beta = 0.1$). In addition to the results *stabilizing* with $\tau_S \geq 0.2$ for most cases, and $\tau \geq 0.4$ for all cases, we observe **a *convergence* of the results *across different data-sets***: the number of VCG nodes and patterns becomes almost identical under any fixed setting of $\tau_S \geq 0.2$ (compare the rows of results for same benchmark over different data-sets, under trimming enabled; discrepancy for `h263dec` is explained later). Therefore, we see **stability** moving across the rows and convergence moving along columns for a given benchmark. We also find that under stability the resulting VCG graphs are *identical in structure and statistics*. This is expected since all results in a row of the table are constructed from the same profile sequence, just summarized or trimmed to different extents. On the other hand, under convergence the *structure of the VCG graphs* is *virtually identical*, differing perhaps with the addition/deletion of one or two nodes and edges. The annotated statistics differ substantially over different data-sets, but they still produce similar *contribution fractions* on corresponding back-edges. This is a more surprising result since the data-sets used (and hence the profiles produced) were quite different for results in the same column. The combined effect of stability and convergence is that under even a low trimming ($\beta = 0.1$), all results (over multiple data-sets) for a benchmark with $\tau \geq 0.4$ are essentially identical in structure and contribution-back-edge-structure, thereby revealing something fundamental about an application's behavior.

9.8.1 Illustrations of VCG Analysis Results

Figure 9.3 shows the VCG for `mpeg2enc`, consisting of 17 nodes in a single pattern. Boxes with solid outlines present Task nodes, ovals present Contrast nodes. To ease reference, a unique id is attached to each VCG node. For example, the root node is *motion_estimation* with id **(0)**. Further, the VCG reveals what is well-known in literature [70, 95] for `mpeg2enc`: motion-estimation is by far the most significant cause of per-frame variability in execution-time, and this variability occurs due to searching for matching image-blocks in adjacent

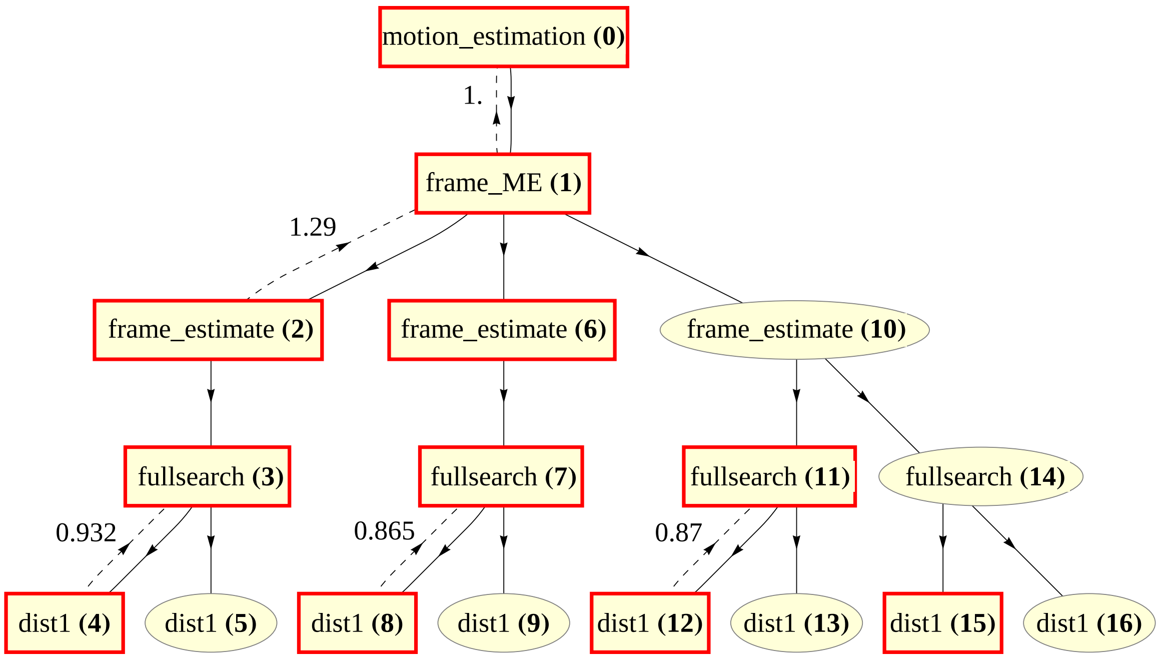


Figure 9.3: VCG for `mpeg2enc` with: `qos640x480`, $\tau_S = 0.4$, $\beta = 0.1$.

video-frames ($fullsearch$ nodes). The $dist1$ function computes a distance measure between offset image-blocks, performing half-pixel interpolation if needed, and accounts for most of the variance of the parent $fullsearch$ (seen as contribution back-edges, with contribution fractions annotated). Examining `mpeg2enc`'s source code reveals that $frame\_ME$ has multiple call-sites for $frame\_estimate$, and usually conditionally invokes any one call-site per invocation of $frame\_ME$. This explains $frame\_estimate$ **(2)** contributing 129% (i.e, $> 100\%$) of the variance in $frame\_ME$ **(1)**, since the different call-sites of $frame\_estimate$ also contribute large negative cross-correlation terms (Eq 72).

Figure 9.4 shows `mpeg2dec` with 15 nodes in 4 patterns, 3 of the patterns being trivial but sufficiently dominant to avoid being trimmed compared to $decode\_macroblock$ **(0)**.

Figure 9.5 and Table 9.1 show the extremely dominant variability of $MotionEstimation$, and $SAD\_Macroblock$ within it, leading to all other functions being trimmed out in `h263enc`.

Figure 9.6 (for `h263dec`) mainly shows pattern trees for $reconstruct$ **(0)**, the per-frame reconstruction function, and $getblock$ **(6)** which performs frame-data-I/O. $recv$ **(2)** and

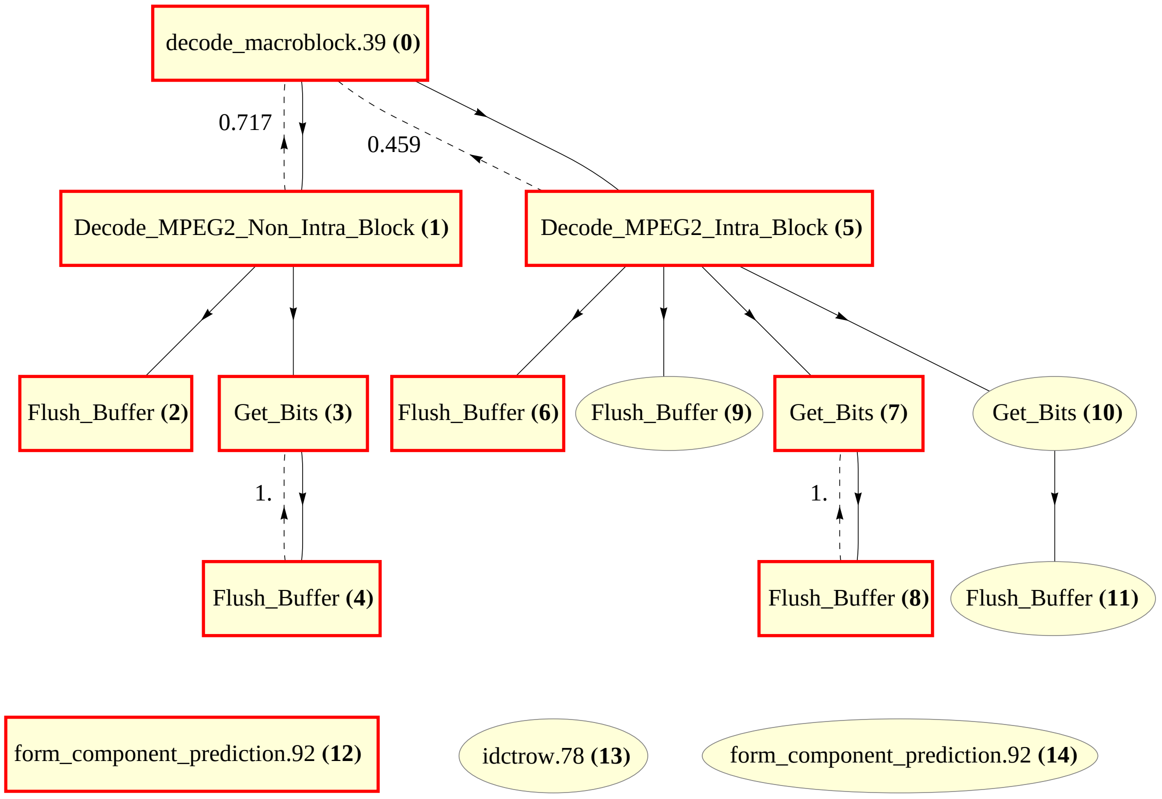


Figure 9.4: VCG for `mpeg2dec` with: `qos320x240`, $\tau_S = 0.01$, $\beta = 0.1$.

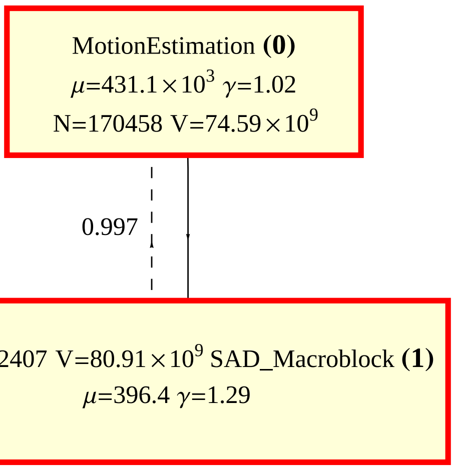


Figure 9.5: VCG for `h263enc` with: `hockey1_cif` $\tau_S = 0.2$ $\beta = 0.1$.

*rec*4 **(3)** are examples of Contributor nodes (dashed outlines), which have low CoV but contribute variance (and large negative cross-correlation terms) to *recon_comp* **(1)** due to mutually exclusive conditional invocation. Also, *recon_comp* **(1)** and **(4)** subtrees seem to exhibit very different behaviors. The corresponding call-sites can be disambiguated based on the Call-Context-Sets (dumped separately). Here **(1)** and **(4)** correspond to lexical call-sites {3} and {4, 5}, respectively, in the body of *reconstruct*.

Figures 9.7-9.12 show the VCGs for additional benchmarks.

9.8.2 Controller

To demonstrate the utility of VCG, we build a controller to reduce the application's variance in execution time by inferring a tight performance model of the source of variance (e.g., the root node in the VCG), which centers around the application's tunable algorithmic parameters. VCG narrows the search space by orders of magnitude and makes the model generation efficient, accurate and tractable.

The controller has an offline component and an online component. The offline component runs once before deploying the application. It synergistically combines program analysis and machine learning to build a performance model for the root node function f_r in the application's VCG, which automatically chooses the features that are the best predictors of its execution time. To obtain these features, the offline component requires the

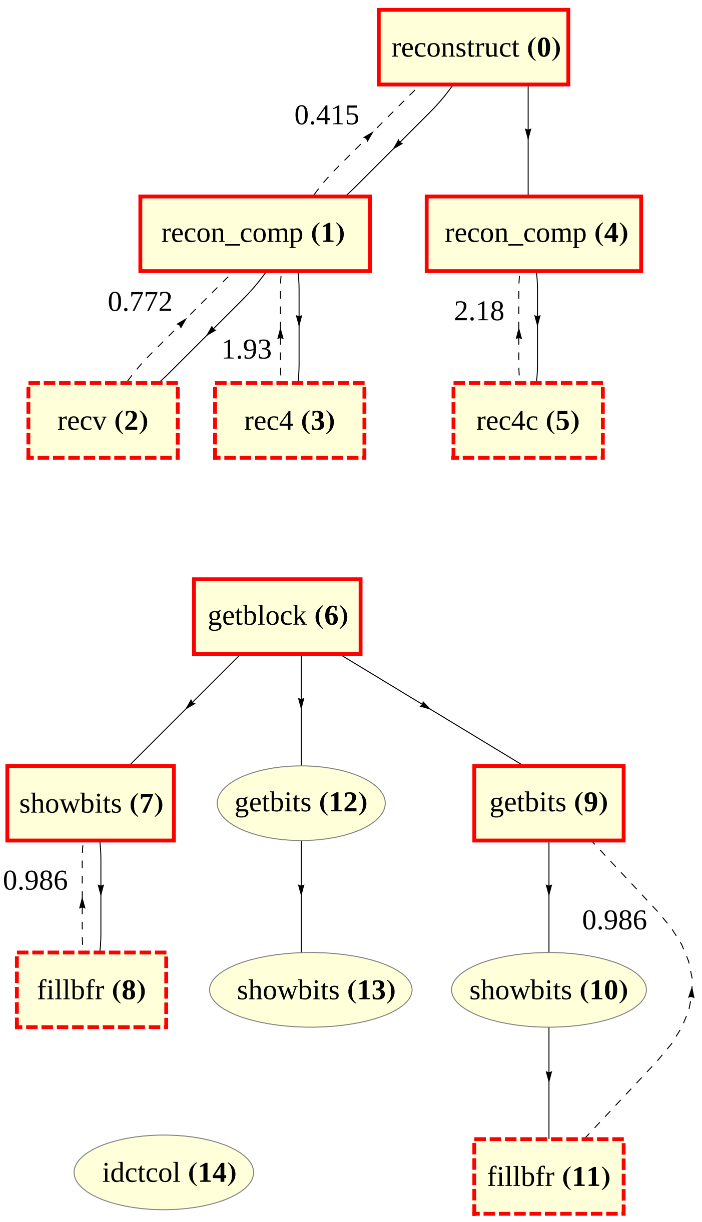


Figure 9.6: VCG for `h263dec` with: `baikonur_r7_overflight`, $\tau_S = 0.4$, $\beta = 0.1$.

application's binary P as well as a set of training inputs $d_1, \ldots, d_N$. Besides performance models, this component also produces as output an augmented binary Q that tracks in a lightweight style the values of features needed by the models to predict execution time on new inputs. We describe the offline component in more detail in Section 9.8.2.1.

The online component runs during the execution of binary Q. It tracks the instrumented feature values and once any feature value changes, it estimates the execution time of that

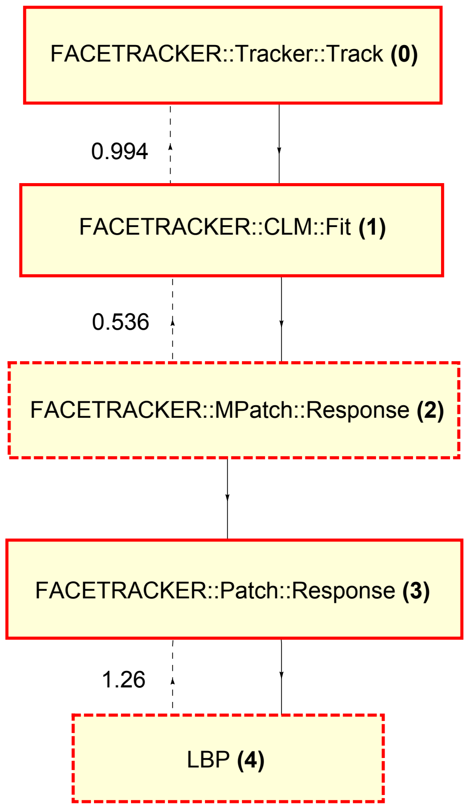

FACETRACKER::Tracker::Track **(0)**
0.994
FACETRACKER::CLM::Fit **(1)**
0.536
FACETRACKER::MPatch::Response **(2)**
FACETRACKER::Patch::Response **(3)**
1.26
LBP **(4)**

Figure 9.7: VCG for `facetrack` with: `adam_sandler` $\tau_S = 0.2$ $\beta = 0.1$.

invocation of f_r by applying the performance model of that function, provided by the offline component, on the current feature values. Finally, tuning logic uses these estimates to decide whether to adjust the values of features that are identified by programmers as tunable algorithmic parameters based on how much the predicted execution time deviates from the average execution time seen so far (e.g., 10% as threshold). We describe the online component in more detail in Section 9.8.2.2.

It is worth noting that the controller construction methods are general enough for any application friendly to the dominant variance analysis.

9.8.2.1 Offline Component

The high-level idea of offline component is to only instrument functions appearing in the application's VCG with features that are potentially good predictors of the execution time of the root function f_r in VCG (Instrumentor), then to profile the instrumented application

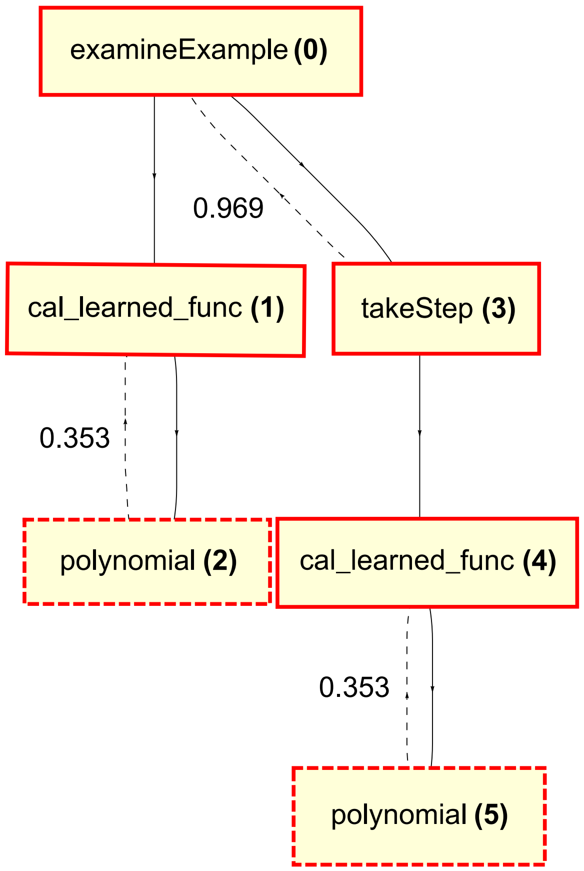


Figure 9.8: VCG for `svm` with: `cif` $\tau_S = 0.2$ $\beta = 0.1$.

to collect values of the features and execution times of the function f_r on a set of given inputs (Profiler), and lastly to use machine learning to build a performance model based on a few features that are the best predictors (Model Generator). We run them twice and obtain one machine learning model each time: one as initial model and the other as final model which is derived from the initial model and deployed in the controller. We next describe each of the above three steps in detail.

To find features that are good predictors, the Instrumentor uses the following five feature schemes, each of which generates a set of features of a particular kind from the application:

- **Branch Counts:** This scheme generates two separate features to track the number of times each conditional branch evaluates to true and false.

- **Loop Counts:** This scheme generates a separate feature to track the number of times

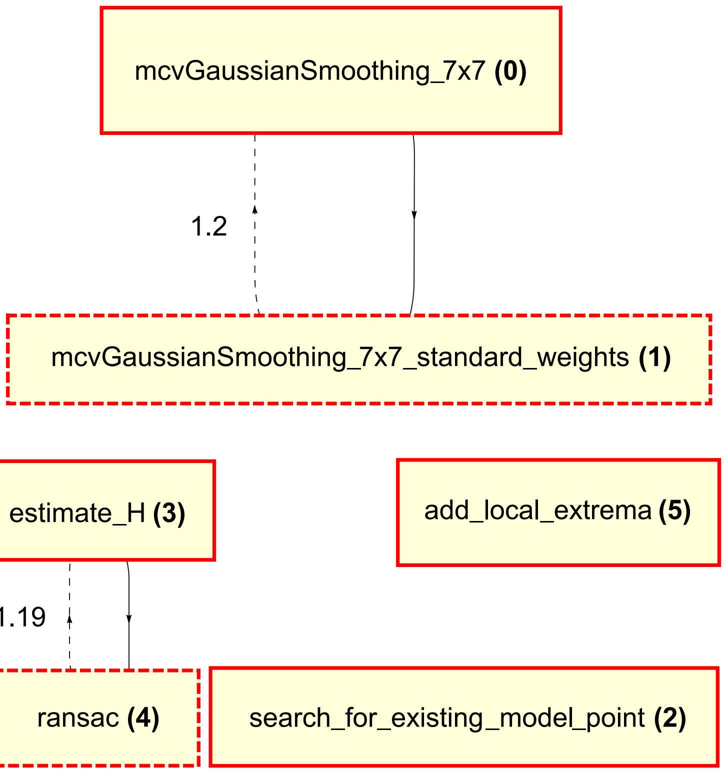

mcvGaussianSmoothing_7x7 (0)
1.2
mcvGaussianSmoothing_7x7_standard_weights (1)
estimate_H (3)
add_local_extrema (5)
1.19
ransac (4)
search_for_existing_model_point (2)

Figure 9.9: VCG for `ferns` with: `mousepad` $\tau_S = 0.2$ $\beta = 0.1$.

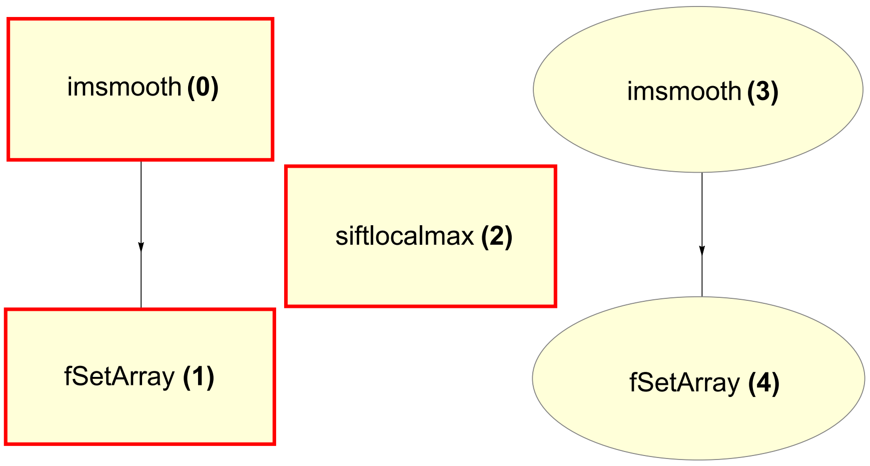


Figure 9.10: VCG for `sift` with: `fullhd` $\tau_S = 0.2$ $\beta = 0.1$.

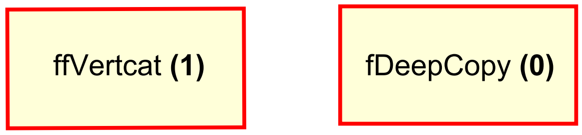


Figure 9.11: VCG for `stitch` with: `fullhd` $\tau_S = 0.8$ $\beta = 0.1$.

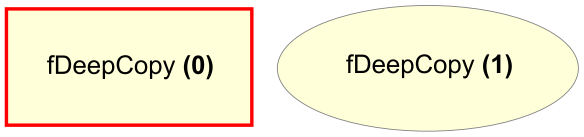


Figure 9.12: VCG for `tracking` with: `fullhd` $\tau_S = 0.2$ $\beta = 0.1$.

each loop iterates.

- **Method Call Counts:** This scheme generates a separate feature to track the number of times each function call is invoked.
- **Return Values:** This scheme generates a separate feature to track the return value of each function call site, provided it is of integer type.
- **Parameter Values:** This scheme generates a separate feature to track each parameter value of integer type of each function call site.

All five schemes in the Instrumentor are implemented using LLVM [84] by iterating over the body of each function appearing in the VCG of that application. More specifically, the branch counts, loop counts and return values schemes track the cumulative value of each feature, i.e., they accumulate these values over all executions of the branch, all executions of the loop and all executions of the call site, respectively, in a particular invocation of the root function f_r in VCG, whereas the parameter values scheme tracks only the most recent value of each feature. And all the features are re-initialized at the entry of function f_r. Finally, the Instrumentor also injects code at the entry and exit of function f_r for the Profiler to measure the execution time. The application augmented in this way is denoted by Q', which is used to produce the initial performance model M'.

The profiler runs the instrumented application Q' produced by the Instrumentor on each of given inputs $d_1, \ldots, d_N$. Whenever function f_r finishes execution, the Profiler records the values of all instrumented features $v_1, \ldots, v_M$ as well as its execution time t of this just finished instance, and outputs a tuple $\langle t, v_1, \ldots, v_M \rangle$. Note that the function f_r may be called multiple times on a single input d_i, providing multiple data points for the Model Generator in a single run.

Finally, the Model Generator uses the tuples generated by the Profiler, and builds a performance model for the function f_r. This model is a polynomial function that approximates the execution time of f_r in terms of features $v_1, \ldots, v_M$. An example model is:

$$0.1 + 0.2v_2 - 1.2v_2v_{125} + 0.5v_{125}^2$$

The model is inferred using the regression algorithm from [96]. Like any regression algorithm, this algorithm fits as closely as possible the feature values $v_1, \ldots, v_M$ and execution time t in each tuple output by the Profiler for function f_r. In addition, it has two salient properties suitable for performance prediction: *sparsity* and *non-linearity*. The sparsity property concerns minimizing the number of features selected in the performance model (e.g., v_2 and v_{125} in the example) which is beneficial for two reasons: *i*) it prevents the model from overfitting problems; and *ii*) the fewer number of features selected, the less overhead in tracking features in the online component. Finally, the non-linear property allows nonlinear terms (e.g., v_2v_{125}) in the model, allowing to approximate the execution time more accurately.

Once the initial performance model M' is generated, our controller requires programmers to partition all the selected features into two categories: *i*) tunable features, $V_{tunable}$ and *ii*) untunable features, $V_{untunable}$. A feature is deemed as tunable if its value derives from at least one algorithmic parameter of the application which makes sense to change in the application's domain. Such an algorithmic parameter is also called tunable algorithmic parameter. Obviously, this step needs human intervention due to requiring domain knowledge, thus preventing us from automation. However, backward program slicing with selected features as slicing criteria facilitates the partition and reduces the manual effort. Therefore we claim it won't be much effort involved. Let us denote the tunable algorithmic parameters that correlate to $V_{tunable}$ as $A_{tunable}$. Then the final feature set is composed of

$V_{untunable}$ and $A_{tunable}$, and a new performance model M with that feature set is generated in a similar way. Meanwhile, an augmented application Q is produced by the Instrumentor, which tracks the values of features in final feature set. Finally, the online component utilizes that information to tune the run-time performance of the instrumented application Q by changing values of specified tunable algorithmic parameters $A_{tunable}$.

9.8.2.2 Online Component

The online component is relatively straightforward. It runs as part of the modified application binary Q produced by the offline component that tracks the values of features used in the performance model M of function f_r, and invokes tuning logic whenever any feature value gets updated. Once tuning logic is invoked, it first checks the deviation between estimated execution time based on current feature values and the average execution time so far. If the difference gets beyond the threshold, tuning logic makes its best efforts to adjust the tunable algorithmic parameters, $A_{tunable}$, to pull back the estimated execution time within bound.

9.8.2.3 Results

We evaluated the potential benefits of VCG by imposing a controller built out of it on four soft-real-time applications described in Table 9.2: two encoders from [70], one planar object detector from [75] and one face tracker from [93]. We deployed controllers to tune online the execution time of the root functions in VCGs since they are the most significant causes of the variance in per-frame execution time.

In Table 9.2, the fourth column shows for each application the number of user-defined functions, total functions (plus library functions) and the functions instrumented for performance model generation w/ and w/o the guidance of VCG. Under the guidance of VCG,, the number of functions instrumented for model generation reduces dramatically, facilitating the model generation and increasing the possibility of capturing algorithmic parameters by the model. The following three columns depict the statistics for performance models: the number of features selected by the performance model, the total number of features instrumented initially, the number of data points for training and testing, and whether the

Table 9.2: Benchmark characteristics and statistics for performance model generation of all benchmarks.

		LOC	# functions				# features				# data		capture	
			app	total	instrumented		selected		total		train	test	w/o	VCG
					w/o	VCG	w/o	VCG	w/o	VCG				
ferns	planar object detector	7.1K	458	579	458	2	–	3	2244	62	3500	1000	✗	✓
mpeg2enc	MPEG-2 video encoder	7.6K	95	123	95	5	–	2	2323	280	2030	1015	✗	✓
h263enc	H.263 video encoder	8.1K	96	122	96	2	–	2	2538	142	3600	2500	✗	✓
facetrack	face tracker	2.7K	770	909	770	5	–	2	1455	146	2766	2555	✗	✓

final model captures the key tunable algorithmic parameters. The tunable parameter for each application is as follows: maximum iteration number for `ferns`, and search window size for the rest. Clearly, VCG captures highly summarized variant patterns in the application, making the model generation tractable over a large scale of data sets while the blind search procedure without VCG makes the machine learning algorithm blow up in complexity due to a huge search space. In addition, it shows the selected regression algorithm minimizes the number of selected features, suitable for performance prediction. The prediction accuracy of four performance models are as follows: 88.6% for `ferns`, 73.4% for `mpeg2enc`, 92.8% for `h263enc` and 97.3% for `facetrack`.

Table 9.3 shows the root functions in VCGs of three applications, the average execution time on per frame basis, the execution time span, the variance of the execution time across all invocations of the root functions, the average rate of change between two consecutive frames in execution time, and the overhead introduced by the controller. We can easily find that with the controller, the changing rate of the execution time of the root function gets smaller, and the variance gets reduced by 14% for `ferns`, 17% for `mpeg2enc`, 8% for `h263enc` and 9.1% for `facetrack`. It is worth mentioning that the three metrics (per-frame execution time, variance and rate) are capturing the aggregated behaviors and thus more faithful. Finally, negative overhead in most cases indicates that such savings not only compensate for the cost of controller deployment but provide extra performance gains as well.

Table 9.3: Results showing statistics about runtime behavior of root functions in VCGs exhibiting highly variant execution time on per frame basis.

	function	**avg per-frame execution time (ms)**		**range (ms)**		**variance**		**rate ($\frac{dt}{dn}$)**		**over-head**
		w/o	controller	w/o	controller	w/o	controller	w/o	controller	
ferns	estimate_H	597.0	553.0	1935	1383	282.6	242.8	0.9	0.8	-3.7%
mpeg2enc	motion_estimation	7449.6	6947.8	4775	3328	897.8	744.8	10.5	7.4	-6.7%
h263enc	MotionEstimation	120.6	118.4	47	44	46.3	42.8	-0.6	-0.6	-1.7%
facetrack	Track	42585.1	35978.7	19483	12812	2140.4	1945.0	11.5	7.9	1.9%

9.9 Related Work

Existing application profiling techniques look for program hot-spots and hot-paths [58, 59, 60]. These techniques attempt to find performance bottlenecks in an application, and do not attempt to characterize patterns of variant behavior. Calder et al. have used statistical techniques to characterize large scale program behavior using few recurrent intervals of code [61] and to find phase change points in the dynamic execution of a program [62]. However, their work does not attempt to characterize the variant behavior in terms of the *functional decomposition* of the application. In particular, [61] focuses on detecting the phase change in programs whereas ours focuses on detecting timing variances of functions. They detect the phase change by noting the differences in basic block vectors in terms of execution frequencies recorded via sampling over fixed intervals. While such a method is adequate to detect a "different phase behavior", it is not sufficient to detect timing variance in that difference in such basic block vectors and their frequencies may or may not imply timing variance and vice versa. In addition, [61] is based on the sampling on the number of instructions, which is unlikely to fall exactly on the boundary of objects (e.g., function), and thus is never applicable for capturing behaviors of functions. Most importantly, existing basic-block technique is not sufficient for inferring VCG over basic-blocks because it does not construct covariance matrices nor maintain difference in behavior of a given basic-block when its parent is invoked under different call-contexts. Variability Characterization Curves (VCCs) and Approximate VCCs [65] have been used to characterize the variability in the workloads of multimedia applications. Such analysis techniques require domain-specific knowledge of the application before they can be applied. Similarly, there are custom techniques for improving the QoS of each type of application, such as by Roitzsch et al. [47] that develop a higher-level representation model of a generic MPEG decoder, and based on this predict video-decoding times with high accuracy. In contrast, our framework characterizes the variant behavior in the application in a completely domain-independent manner, with no assistance from the user. Perelman et al. [97] introduced Variational Path Profiling (VPP), that identifies acyclic program paths that exhibit a lot of variability in execution-time over their multiple invocations. Since the sequence of instructions is identical for each execution of a given path, variability in execution time is only caused by architectural or system variabilities, such as cache-misses or interrupts. In contrast, our work primarily characterizes the *variation in functionality* corresponding to functions performing varying levels of computation over their repeated invocations, depending on the characteristics of the data they are processing. A further contrast is that VPP applies to acyclic paths, which are typically of limited length. Hence, VPP is fixed in scope of applicability, whereas our analysis applies to functionality at all levels of hierarchy in the application, and over all levels of granularity of execution time. Worst-case-execution-time (WCET) [68] is an analysis methodology applicable to monolithic applications, and has been incorporated into commercial products such as from AbsInt [69]. However, for non-safety-critical, compute-intensive applications like gaming and video, knowledge of the *likely range* of real-time behavior is more important for driving design optimization than knowledge of worst-case behavior. The likely range (detected by our technique) can be substantially removed from the worst case, thereby diminishing the value of characterizing the worst-case behavior for such applications.

9.10 Conclusion

In this chapter we have illustrated the value of profiling the variant behavior of an application. We introduce the VCG representation to readily allow the application-wide variant behavior of the application to be succinctly represented. We introduce the Dominant Variance Analysis framework that allows precision-of-results versus summarization-of-results trade-offs to be readily made by tweaking a single parameter τ_S. The VCG representation captures the variant structure of the application that we expect corresponds closely to the high-level design of the application. We have showed empirically that the VCG remains stable across different executions. The methods for β-based trimming and τ_S-based summarization attempt to distill what is dominant behavior, either observed by itself or inferred by combining multiple instances of similar behavior. Stability of analysis results comes naturally from our technique when there are application behaviors that dominate and we are able to highlight them while suppressing behaviors that are not comparatively dominant. We have demonstrated that VCG serves as a very useful summarization to build the controllers for reducing variance. In particular, the use of VCG made the problem of building models tractable and secondly, the built models allowed reduction in the variance of the application.

CHAPTER X

CONCLUSION

In this thesis we showed that meeting the QoS needs of immersive applications, such as computer gaming, multimedia and computer vision applications, poses difficult challenges for existing techniques. This thesis identified common characteristics shared by a wide variety of immersive applications, and showed how these characteristics can be relied upon to construct new profiling and control techniques that non-expert programmers may easily apply to their applications.

The proposed profiling technique, dominant variance analysis (DVA), helps the programmer understand the dominant patterns of program behavior across all levels of functional granularity. We define the behavior of application components in terms of their execution-time variance statistics, as this definition of behavior allows us to easily identify the application components most likely to impact QoS without requiring the profiling technique to have knowledge about the nature of the application and its particular notion of QoS. The use of variance naturally allows the identification of repeatedly invoked components that impact, for example, the application frame rate and various image-block processing rates, as well the identification of components with sensitivity to their input data or their call-context of invocation. When applied to immersive applications, DVA presents non-expert programmers with the application components most likely to impact the application QoS, and the patterns of behavior exhibited by those components. More generally, beyond immersive applications, DVA helps identify application components with sensitivity to their call-context and their data.

We have proposed two controllers, a very light-weight uni-variate controller, and a much more compute-intensive multi-variate controller, to allow non-expert programmers to easily tune the QoS of their immersive applications. The complexity and emergent nature of behavior of immersive applications generally makes it infeasible for non-expert programmers to provide a priori models of their application's behavior. Further, immersive applications often exhibit rapidly time-varying and highly data-set sensitive behavior, that makes it unlikely that a fixed behavior model could even exist. The lack of offline models and inability to assume the existence of a fixed, albeit unknown, behavior model poses challenges in the application of existing general-purpose adaptive control techniques. Our proposed controllers rely on the common properties of immersive applications to make the dynamic control of application QoS tractable without the availability/assumption of a priori fixed or parametric models. The performance and robustness properties of the controllers are probabilistic best-effort, reflecting the probabilistic properties that hold true over the highly time-varying, data-set dependent behavior of immersive applications. The controllers provide APIs where the programmer can express the QoS goals of their application as a variance minimization/bounding problem. Both controllers rely on the probabilistic properties of immersive applications to maximize the number of frames where the QoS variance is limited to acceptable bounds, or failing which, the QoS variance is minimized.

The uni-variate controller performs adaptive gain-scheduling, but with the gain-scheduling driven by metrics that track properties relevant for immersive applications. The multi-variate controller is categorized under model-identification adaptive control (MIAC), but with the structure and cost-functions of model estimation and regulator construction dictated by the common properties of immersive applications. Additionally, the orchestration of various critical activities within the controller is dictated by the common properties — in particular, *i)* the generation of high quality online training data to allow estimation of models that more accurately capture the current application behavior, *ii)* the detection of changes in application behavior to trigger new model estimation, and *iii)* the balance between dedicating some application frames to the exploration of application behavior to find models that may deliver higher QoS than the current model, versus, the exploitation of the current model on as many frames as possible to allow the current model to deliver the best QoS it can. The controllers are effective at providing tight QoS control for an application over a variety of data sets and execution platforms provided the programmers verify that their application satisfies the common properties we identify for immersive applications.

10.1 Future Work

The light-weight uni-variate controller can be custom-extended to tune special platform parameters in addition to tuning a given application algorithmic parameter. The special platform parameters include the number of cores allocated to the application and dynamic voltage and frequency scaling (DVFS). The dynamic tuning of the special platform parameters in addition to the application parameter will allow the controller to save power or reduce the likelihood of hitting thermal throttling limits while allowing the platform to provide just sufficient compute capabilities to meet the application QoS requirements. This future work will differ from numerous prior work on DVFS and the control of degree-of-parallelism in that it will allow tight QoS and power control for immersive applications, which would be challenging for prior techniques for all the reasons mentioned previously about the nature of immersive applications. However, the multi-variate controller may be directly applied to tune DVFS and the degree of concurrency, as the impact of tuning these platform parameters will often satisfy the required common properties.

It is possible to significantly reduce the runtime overhead of the multi-variate controller, possibly by an order of magnitude or more. We will explore approximation techniques, re-factoring to minimize re-computation of various controller coefficients, improved cache utilization by the various controller data-structures, and better use of specialized architecture ISAs to speed up linear algebra, such as SIMD, vectorization and GPGPUs.

We will explore the extension of the DVA profiling technique to program and architecture properties other than execution time. With minor changes DVA may be applied to study the dominant patterns of cache accesses, memory traffic, network accesses, etc.

Finally, we seek to re-apply the approach of identifying common properties of a domain of applications to domains other than immersive applications. We seek to ease the QoS characterization and control of additional software applications where the application behavior does not allow traditional adaptive control techniques to be applied. In general, we seek to create controllers whose notion of performance and robustness reflects the common properties of the application domain (e.g., probabilistic and statistical properties) rather than being limited to the traditional guarantees.

APPENDIX A

DERIVATIONS

A.1 Incremental Update of History Statistics

Incremental update of mean and standard deviation $\vec{\mu}$ and $\vec{\sigma}$. We derive the incremental update formulae for the scalar case, as the multi-variable problem just applies the scalar solution independently to each dimension of $\vec{x}$. Let $\{a_1, a_2, \dots, a_{n-1}\}$ represent an *online sequence* of scalar data samples, i.e., at the next step sample a_n would be added. The mean and variance can be updated incrementally as follows.

$$\begin{aligned}\bar{a}_n &= \bar{a}_{n-1} + \frac{a_n - \bar{a}_{n-1}}{n} \\ M_{2,n} &= M_{2,n-1} + (a_n - \bar{a}_{n-1})(a_n - \bar{a}_n) \\ \sigma_n^2 &= \frac{M_{2,n}}{n}\end{aligned}$$

Our use case needs the samples $\{a_1, a_2, \dots, a_{n-1}\}$ to contribute to the statistics with weights $\left\{\gamma^{n-2}, \gamma^{n-3}, \dots, 1\right\}$, respectively. When a new sample a_n gets added, we need the samples $\{a_1, a_2, \dots, a_{n-1}, a_n\}$ to be *re-weighted* with $\left\{\gamma^{n-1}, \gamma^{n-2}, \dots, \gamma, 1\right\}$. The weighted mean and variance are defined as follows.

$$\bar{a}_n \triangleq \frac{a_n + \gamma a_{n-1} + \gamma^2 a_{n-2} + \ldots + \gamma^{n-1} a_1}{1 + \gamma + \gamma^2 + \ldots + \gamma^{n-1}} \tag{74}$$

$$M_{2,n} \triangleq (a_n - \bar{a}_n)^2 + \gamma(a_{n-1} - \bar{a}_n)^2 + \gamma^2(a_{n-2} - \bar{a}_n)^2 + \ldots + \gamma^{n-1}(a_1 - \bar{a}_n)^2 \tag{75}$$

$$\sigma_n^2 \triangleq \frac{M_{2,n}}{1 + \gamma + \gamma^2 + \ldots + \gamma^{n-1}} \tag{76}$$

The following modified formulae perform a *weighted incremental update* of the statistics when sample a_n is added.

$$\begin{aligned}
\bar{a}_n &= \frac{(1-\gamma)a_n + \gamma(1-\gamma^{n-1})\bar{a}_{n-1}}{1-\gamma^n} \\
M_{2,n} &= \gamma M_{2,n-1} + (a_n - \bar{a}_{n-1})(a_n - \bar{a}_n) \\
\sigma_n^2 &= \frac{1-\gamma}{1-\gamma^n} M_{2,n}
\end{aligned}$$

In addition to adding a new sample a_n at each time step, we may also need to delete one or more of the oldest samples (in the order of oldest first). With each deletion, we want to transform the sample sequence $\{a_1, a_2, \ldots, a_{n-1}, a_n\}$ weighted with $\left\{\gamma^{n-1}, \gamma^{n-2}, \ldots, \gamma, 1\right\}$ to the sample sequence $\{a_2, \ldots, a_{n-1}, a_n\}$ weighted with $\left\{\gamma^{n-2}, \ldots, \gamma, 1\right\}$, i.e., drop the earliest sample without re-weighting the remaining samples. When a_1 is dropped, the statistics for the shortened sequence are as follows.

$$\bar{a}'_{n-1} = \frac{a_n + \gamma a_{n-1} + \gamma^2 a_{n-2} + \ldots + \gamma^{n-2} a_2}{1 + \gamma + \gamma^2 + \ldots + \gamma^{n-2}} \tag{77}$$

$$M'_{2,n-1} = (a_n - \bar{a}'_{n-1})^2 + \gamma(a_{n-1} - \bar{a}'_{n-1})^2 + \gamma^2(a_{n-2} - \bar{a}'_{n-1})^2 + \ldots + \gamma^{n-2}(a_2 - \bar{a}'_{n-1})^2 \tag{78}$$

$$\sigma'^2_{n-1} = \frac{M'_{2,n-1}}{1 + \gamma + \gamma^2 + \ldots + \gamma^{n-2}} \tag{79}$$

Once the deletion is done, we will just re-label the shortened sequence $\{a_2, \ldots, a_{n-1}, a_n\}$ as $\{b_1, \ldots, b_{n-2}, b_{n-1}\}$, back to the canonical form on which we can perform additional decremental (deletion) or incremental updates. The following formulae perform a *weighted decremental update* of the statistics, with statistics re-labeled.

$$\begin{aligned}
\bar{a}_{n-1} &= \frac{(1-\gamma^n)\bar{a}_n - (1-\gamma)\gamma^{n-1} a_1}{1-\gamma^{n-1}} \\
M_{2,n-1} &= M_{2,n} + \frac{1-\gamma^n}{1-\gamma}(\bar{a}_n - \bar{a}_{n-1})^2 - \gamma^{n-1}(a_1 - \bar{a}_{n-1})^2 \\
\sigma_{n-1}^2 &= \frac{1-\gamma}{1-\gamma^{n-1}} M_{2,n-1}
\end{aligned}$$

Determining Coverage Threshold f. Consider a reference sequence $\{a_1, a_2, \ldots, a_{n-1}\} = \{+\alpha N_j, -\alpha N_j, +\alpha N_j, -\alpha N_j, \ldots\}$ for input dimension x_j, with $0 < \alpha < 1$. If αN_j is a significant portion of N_j (say, $\alpha = 0.5$), the statistics of the reference sequence would provide a threshold against which an arbitrary training sequence can be compared to determine if the training sequence has coverage. By simple substitution into Eqs 74-76 we get the standard deviation of the reference sequence as follows.

$$\sigma_{\texttt{ref}} = \frac{2\alpha\sqrt{\gamma}}{1+\gamma} N_j \tag{80}$$

Therefore, for any choice of $0 < \alpha < 1$ we deem appropriate and for any γ used, we get $f = \dfrac{2\alpha\sqrt{\gamma}}{1+\gamma}$ for use in Eq 47. We define $\alpha = 0.5$ as providing good coverage, producing $f = \frac{\sqrt{\gamma}}{1+\gamma}$. Now $f \approx 0.5$ for $\gamma > 0.4$, i.e., unless history length L_γ is picked very short (< 3 samples).

Determining Coverage Threshold g. For input dimension x_j, we define a swing from $+\frac{N_j}{2}$ to $-\frac{N_j}{2}$ (or, vice versa) in the two most recent frames as the minimum threshold for a significant swing. Since the sample at timestep $t-k$ is weighed by γ^{k-1}, the least value of the threshold occurs when at timesteps $t-1$ and $t-2$, x_j takes values $+\frac{N_j}{2}$ and $-\frac{N_j}{2}$, respectively.

$$\text{Threshold} = \frac{N_j}{2} - (-\frac{N_j}{2})\gamma = \frac{1+\gamma}{2} N_j$$

Therefore, $g = \dfrac{1+\gamma}{2}$ in the significant swing test $\chi_j \geq g\, N_j$.

Incremental update of maximum swing $\vec{\chi}$. We derive the incremental update formulae for the scalar case, as the multi-variable problem just applies the scalar solution independently to each dimension of $\vec{x}$. Let $A = \{a_1, a_2, \ldots, a_{n-1}\}$ represent an *online sequence* of scalar data samples, i.e., at the next step sample a_n would be added producing $B = \{a_1, a_2, \ldots, a_{n-1}, a_n\}$. We use the following notation in this derivation. Let $d(A, k)$ represent the largest swing between any two samples in A excluding the most recent k samples (i.e., excluding $\{a_{n-k}, a_{n-k+1}, \ldots, a_{n-1}\}$).

$$d(A,k) \triangleq \max_{k<i<j\leq n-1} \left|\gamma^{i-1}a_{n-i} - \gamma^{j-1}a_{n-j}\right|$$

Similarly, let $d(B,k)$ represent the largest swing between any two samples in B excluding the most recent k samples (i.e., excluding $\{a_{n-k+1}, a_{n-k+2}, \ldots, a_n\}$).

$$d(B,k) \triangleq \max_{k<i<j\leq n} \left|\gamma^{i-1}a_{n+1-i} - \gamma^{j-1}a_{n+1-j}\right|$$

When $\mathcal{H}$ has the input samples A, $\chi = d(A,0)$. When sample a_n is appended to $\mathcal{H}$, we get $\chi = d(B,0)$.

$$\begin{aligned}
\text{By definition, } d(B,1) &= \max_{1<i<j\leq n} \left|\gamma^{i-1}a_{n+1-i} - \gamma^{j-1}a_{n+1-j}\right| \\
&= \max_{0<i<j\leq n-1} \left|\gamma^{i}a_{n-i} - \gamma^{j}a_{n-j}\right| \\
&= \gamma \max_{0<i<j\leq n-1} \left|\gamma^{i-1}a_{n-i} - \gamma^{j-1}a_{n-j}\right| \\
&= \gamma\, d(A,0).
\end{aligned}$$

Let $v_{min} = \min\limits_{1\leq j\leq n-1} \gamma^{j-1}a_{n-j}$ and $v_{max} = \max\limits_{1\leq j\leq n-1} \gamma^{j-1}a_{n-j}$. That is, the smallest and the largest weighted samples in A, respectively.

$$\begin{aligned}
\text{Further, } d(B,0) &= \max\left\{d(B,1),\ \max_{1<j\leq n} \left|a_n - \gamma^{j-1}a_{n+1-j}\right|\right\} \quad \text{(separating out } i=1\text{)} \\
&= \max\left\{d(B,1),\ \max_{0<j\leq n-1} \left|a_n - \gamma^{j}a_{n-j}\right|\right\} \quad \text{(replacing } j-1 \text{ with } j\text{)} \\
&= \max\left\{d(B,1),\ |a_n - \gamma\, v_{min}|,\ |a_n - \gamma\, v_{max}|\right\}.
\end{aligned}$$

Therefore, we get the following *incremental update formulae after a new sample x is appended to $\mathcal{H}$.*

$$\chi \leftarrow \max\left(\gamma\,\chi,\ |x - \gamma\, v_{min}|,\ |x - \gamma\, v_{max}|\right) \tag{81}$$

$$v_{min} \leftarrow \min\left(x,\ \gamma\, v_{min}\right) \tag{82}$$

$$v_{max} \leftarrow \max\left(x,\ \gamma\, v_{max}\right) \tag{83}$$

Now consider sequence $C = \{a_2, \ldots, a_n\}$ left after dropping the oldest sample a_1 from B. We get the following *decremental formulae just before the oldest sample* $a_1 = \mathcal{H}[\,|\mathcal{H}|\,]$ *is dropped from* $\mathcal{H}$.

$$v_{min} \leftarrow \begin{cases} v_{min}, & \text{if } v_{min} < \gamma^{|\mathcal{H}|-1}\, a_1 \\ \min\limits_{1 \le j < |\mathcal{H}|} \gamma^{j-1}\mathcal{H}[j], & \text{otherwise} \end{cases} \tag{84}$$

$$v_{max} \leftarrow \begin{cases} v_{max}, & \text{if } v_{max} > \gamma^{|\mathcal{H}|-1}\, a_1 \\ \max\limits_{1 \le j < |\mathcal{H}|} \gamma^{j-1}\mathcal{H}[j], & \text{otherwise} \end{cases} \tag{85}$$

$$\chi \leftarrow v_{max} - v_{min} \tag{86}$$

Here, the updated v_{min} and v_{max} are the smallest and the largest weighted samples in C (i.e., in B excluding a_1), respectively. The updates of v_{min} and v_{max} sometimes require a scan over all the samples of $\mathcal{H}$ when the smallest or the largest weighted sample is being dropped, but are usually incremental updates.

A.2 Estimation of PFE Cluster Length $L_{\mathbf{PFE}}$

Consider history $\mathcal{H}$ when it has insufficient coverage. Let x_j represent the input dimension exhibiting the least coverage, i.e., the least $\dfrac{\sigma_j}{N_j}$. Consider a PFE cluster of length L_{PFE}. Given the coverage metrics over $\mathcal{H}$, what is the least value of L_{PFE} that will achieve coverage on x_j assuming that the PFE cluster will consist of maximally variant samples $\{+N_j, -N_j, +N_j, -N_j, \ldots\}$?

For simplicity, let μ_o and σ_o represent the *observed* statistics for samples of x_j in $\mathcal{H}$. Let μ_e and σ_e represent the *estimated* statistics once L_{PFE} maximally variant samples are added to $\mathcal{H}$, with perhaps the oldest samples of $\mathcal{H}$ dropped to maintain $|\mathcal{H}| \le L_\gamma$. Let K represent the final length of $\mathcal{H}$ after the addition of the PFE cluster and the dropping of the oldest samples, E represent the excess samples of $\mathcal{H}$ that will be dropped, and R the number of samples of $\mathcal{H}$ retained.

Consider the following cases:

- If $L_{\mathrm{PFE}} + |\mathcal{H}| \leq L_\gamma$ (no samples need to be dropped):

$$\begin{aligned} K &= L_{\mathrm{PFE}} + |\mathcal{H}|, \\ E &= 0, \\ R &= K - L_{\mathrm{PFE}} = |\mathcal{H}| \end{aligned}$$

- Otherwise, samples must be dropped:

$$\begin{aligned} K &= L_\gamma, \\ E &= L_{\mathrm{PFE}} + |\mathcal{H}| - L_\gamma, \\ R &= K - L_{\mathrm{PFE}} = |\mathcal{H}| - E = L_\gamma - L_{\mathrm{PFE}} \end{aligned}$$

We make a simplifying assumption that the R retained samples exhibit the same μ_o and σ_o statistics as the original full sequence of samples in $\mathcal{H}$. Then, from definitions,

$$\begin{aligned} \frac{1-\gamma^K}{1-\gamma}\mu_e &= \sum_{l=0}^{L_{\mathrm{PFE}}-1} (-1)^l N_j \gamma^l + \sum_{l=L_{\mathrm{PFE}}}^{K-1} \mathcal{H}[l - L_{\mathrm{PFE}} + 1]\gamma^l \\ &= \sum_{l=0}^{\frac{L_{\mathrm{PFE}}}{2}-1} (1-\gamma) N_j (\gamma^2)^l + \sum_{l=0}^{K-L_{\mathrm{PFE}}-1} \mathcal{H}[l+1]\gamma^{L_{\mathrm{PFE}}+l} \quad (\text{for } L_{\mathrm{PFE}} = \text{ even}) \\ &= (1-\gamma)\frac{1-\gamma^{L_{\mathrm{PFE}}}}{1-\gamma^2} N_j + \gamma^{L_{\mathrm{PFE}}} \sum_{l=0}^{K-L_{\mathrm{PFE}}-1} \mathcal{H}[l+1]\gamma^l \\ &= \frac{1-\gamma^{L_{\mathrm{PFE}}}}{1+\gamma} N_j + \gamma^{L_{\mathrm{PFE}}} \frac{1-\gamma^R}{1-\gamma}\mu_o \quad (\text{as per the simplifying assumption}). \end{aligned}$$

Similarly,

$$\begin{aligned}
\frac{1-\gamma^K}{1-\gamma}{\sigma_e}^2 &= \sum_{l=0}^{L_{\mathrm{PFE}}-1}((-1)^l N_j-\mu_e)^2\gamma^l + \sum_{l=L_{\mathrm{PFE}}}^{K-1}(\mathcal{H}[l-L_{\mathrm{PFE}}+1]-\mu_e)^2\gamma^l \\
&= \sum_{l=0}^{\frac{L_{\mathrm{PFE}}}{2}-1}\left[(N_j-\mu_e)^2+\gamma(N_j+\mu_e)^2\right](\gamma^2)^l \\
&\quad + \gamma^{L_{\mathrm{PFE}}}\sum_{l=0}^{K-L_{\mathrm{PFE}}-1}(\mathcal{H}[l+1]-\mu_o+\mu_o-\mu_e)^2\gamma^l \\
&= \frac{1-\gamma^{L_{\mathrm{PFE}}}}{1-\gamma^2}\left[(N_j-\mu_e)^2+\gamma(N_j+\mu_e)^2\right] \\
&\quad + \gamma^{L_{\mathrm{PFE}}}\Big[\sum_{l=0}^{K-L_{\mathrm{PFE}}-1}(\mathcal{H}[l+1]-\mu_o)^2\gamma^l \\
&\qquad + 2(\mu_o-\mu_e)\sum_{l=0}^{K-L_{\mathrm{PFE}}-1}(\mathcal{H}[l+1]-\mu_o)\gamma^l + (\mu_o-\mu_e)^2\sum_{l=0}^{K-L_{\mathrm{PFE}}-1}\gamma^l\Big] \\
&= \frac{1-\gamma^{L_{\mathrm{PFE}}}}{1-\gamma^2}\left[(N_j-\mu_e)^2+\gamma(N_j+\mu_e)^2\right] \\
&\quad + \gamma^{L_{\mathrm{PFE}}}\left[\frac{1-\gamma^R}{1-\gamma}{\sigma_o}^2+(\mu_o-\mu_e)^2\frac{1-\gamma^R}{1-\gamma}\right].
\end{aligned}$$

Therefore, for any given choice of L_{PFE} ($\leq L_\gamma$) the following equations hold.

$$K = \begin{cases} L_{\mathrm{PFE}}+|\mathcal{H}|, & \text{if } L_{\mathrm{PFE}}+|\mathcal{H}| \leq L_\gamma \\ L_\gamma, & \text{otherwise} \end{cases} \tag{87}$$

$$R = \begin{cases} |\mathcal{H}|, & \text{if } L_{\mathrm{PFE}}+|\mathcal{H}| \leq L_\gamma \\ L_\gamma - L_{\mathrm{PFE}}, & \text{otherwise} \end{cases} \tag{88}$$

$$\mu_e = \frac{1}{1-\gamma^K}\left[\frac{1-\gamma^{L_{\mathrm{PFE}}}}{1+\gamma}(1-\gamma)N_j + \gamma^{L_{\mathrm{PFE}}}(1-\gamma^R)\mu_o\right] \tag{89}$$

$$\begin{aligned}
{\sigma_e}^2 = \frac{1}{1-\gamma^K}\Bigg[&\frac{1-\gamma^{L_{\mathrm{PFE}}}}{1+\gamma}\left[(N_j-\mu_e)^2+\gamma(N_j+\mu_e)^2\right] \\
&+ \gamma^{L_{\mathrm{PFE}}}(1-\gamma^R)\left[{\sigma_o}^2+\gamma^{L_{\mathrm{PFE}}}(\mu_o-\mu_e)^2\right]\Bigg]
\end{aligned} \tag{90}$$

Algorithm Sketch for L_{PFE} Estimation. When $\mathcal{H}$ lacks coverage, L_{PFE} is estimated as follows.

1. Identify x_j with the least $\dfrac{\sigma_j}{N_j}$. Let $\mu_o = \mu_j$ and $\sigma_o = \sigma_j$.

2. Perform binary search to determine least L_{PFE} that will produce μ_e and σ_e satisfying $\sigma_e \geq 0.5N_j$ (coverage threshold).

   - Each step picks candidate length L and computes σ_e using Eqs. 87 - 90.
   - The bounds for the binary search are $L \in \left[1, \left\lceil \frac{L_\gamma}{2} \right\rceil\right]$.
   - **Invariant 1**: *upper* is the least value for L_{PFE} known so far (i.e. least cluster length that will confer coverage with maximally variant samples). Initially *upper* $\leftarrow \left\lceil \frac{L_\gamma}{2} \right\rceil$.
   - **Invariant 2**: *lower* is the largest known cluster length that does not confer coverage even with maximally variant samples. Initially *lower* $\leftarrow 1$.
   - Algorithm terminates when *upper* $=$ *lower*, violating invariant 2.

Estimating the number of PFE clusters to split L_{PFE} into. L_{PFE} is the estimated length of a *single* PFE cluster to achieve coverage. We expect the controller to determine a non-trivial probability q of starting a PFE cluster when the current model's QoS performance is poor, so a better model may be estimated expeditiously. When we expect the controller to achieve coverage in a single PFE cluster with high probability, there are *countervailing scenarios* when L_γ is large (described in Section 6.4.1), where a large θ does not produce a large q — the controller determines that a large cluster is needed but the probability q of starting the cluster becomes too small. Instead, the controller must exercise *multiple PFE clusters in close succession* (a cluster group), with each cluster of a much more limited expected length $E\{d\}$, but which collectively are likely to achieve coverage in close to L_{PFE} frames. Splitting L_{PFE} into a sufficient number of clusters allows $E\{d\}$ to become sufficiently small that a high θ can produce a high q.

The controller maintains a metric numClusters and uses $d_{\text{peak}} \leftarrow \dfrac{L_{\text{PFE}}}{\text{numClusters}}$ (Eq. 56) to shape the PFE cluster length distribution and adjust its expected value $E\{d\}$. As a heuristic, we require numClusters to be the smallest positive integer that produces $q \geq \theta/5$ while respecting the practical constraint $E\{d\} \geq 1$. The motivation for the heuristic is to have at least a 10% probability of starting a PFE cluster (i.e., $q = 0.10$) when θ has its maximum value of 0.50 (i.e., when the controller is maximally biased towards finding an alternative model).

A.3 Justification for scaling $\varepsilon\vec{y}_{t-1}$ with $\vec{\beta}$

Here we demonstrate that the changes applied by the regulator to every control input x_j in the next time step t vary monotonically (in fact, linearly) with the tracking error $\varepsilon y_{i|t-1}$ for each output y_i from the previous time-step (see Section 6.5.2). Establishing this property is a pre-condition for applying the adaptive-integral controller from Chapter 4 to our problem here of dynamically adjusting the tracking error reported to the LQR regulator. The dynamic adjustment of the tracking error overcomes a limitation of the LQR-constructed regulator — the underlying application-response may be locally non-linear and/or may locally deviate from the globally-fitted linear model used to create the regulator.

A linear dynamical system is modeled as (Eq. 17)

$$\vec{s}_{t+1} = A\,\vec{s}_t + B\,\vec{u}_t$$

Our objective is to have a transformed state $C\,\vec{s}_t$ track a desired specified trajectory $\vec{r}_t$. We choose a suitable state representation and construct C so that $C\,\vec{s}_t = \vec{y}_{t-1}$ and $\vec{r}_t = \vec{\bar{y}}$.

Due to linearity, an input sequence $\vec{u}_t^{\,a}, \vec{u}_{t+1}^{\,a}, \cdots, \vec{u}_{t+N}^{\,a}$ and corresponding state sequence $\vec{s}_t^{\,a}, \vec{s}_{t+1}^{\,a}, \cdots, \vec{s}_{t+N}^{\,a}$ satisfying Eq. 17 can be added to any other input sequence and corresponding state sequence satisfying Eq.17, $\vec{u}_t^{\,b}, \vec{u}_{t+1}^{\,b}, \cdots, \vec{u}_{t+N}^{\,b}$ and $\vec{s}_t^{\,b}, \vec{s}_{t+1}^{\,b}, \cdots, \vec{s}_{t+N}^{\,b}$ respectively, and produce a new input sequence and corresponding state sequence that also satisfies Eq. 17, as shown below.

$$\begin{aligned} &\text{If } \vec{s}_{t+1}^{\,a} = A\,\vec{s}_t^{\,a} + B\,\vec{u}_t^{\,a} \text{ and } \vec{s}_{t+1}^{\,b} = A\,\vec{s}_t^{\,b} + B\,\vec{u}_t^{\,b}, \\ &\qquad\qquad \text{then } \vec{s}_{t+1}^{\,c} = A\,\vec{s}_t^{\,c} + B\,\vec{u}_t^{\,c} \\ &\qquad\qquad\qquad\qquad \text{for } \vec{u}_t^{\,c} = \vec{u}_t^{\,a} + \vec{u}_t^{\,b} \text{ and } \vec{s}_t^{\,c} = \vec{s}_t^{\,a} + \vec{s}_t^{\,b}. \end{aligned} \tag{91}$$

Linearity also applies to the outputs:

$$\vec{y}_t^{\,c} = \vec{y}_t^{\,a} + \vec{y}_t^{\,b} \text{ where } \vec{y}_t^{\,a} = C\,\vec{s}_{t+1}^{\,a},\; \vec{y}_t^{\,b} = C\,\vec{s}_{t+1}^{\,b},\; \vec{y}_t^{\,c} = C\,\vec{s}_{t+1}^{\,c}. \tag{92}$$

Consider the solution to the trajectory tracking LQR problem, given by Eqs. 21-25:

$$\begin{aligned}
K_t &:= (B^T W_{t+1} B + R)^{-1} B^T W_{t+1} A \\
W_t &:= A^T W_{t+1}(A - BK_t) + C^T Q C\,, && \text{with } W_N := C^T Q_f C \\
\vec{v}_t &:= (A - BK_t)^T \vec{v}_{t+1} + C^T Q \vec{r}_t\,, && \text{with } \vec{v}_N := C^T Q_f \vec{r}_N \\
K_t^v &:= (B^T W_{t+1} B + R)^{-1} B^T \\
\vec{u}_t &:= -K_t \vec{s}_t + K_t^v \vec{v}_{t+1}
\end{aligned}$$

At time t, let N represent the number of time-steps remaining in the LQR control horizon. For convenience, let $V_{t+1} = (B^T W_{t+1} B + R)^{-1} B^T$. The next control-input determined by LQR for time t has the form:

$$\begin{aligned}
\vec{u}_t \quad &:= \quad \underbrace{-K_t \vec{s}_t}_{\text{Zero-driving input}} \quad + \quad \underbrace{K_t^v \vec{v}_{t+1}}_{\text{Trajectory offset}} \\
&= \quad \vec{z}_t \quad + \quad \vec{o}_t
\end{aligned} \tag{93}$$

Note that $\vec{z}_t$ does not depend on the trajectory $\vec{r}_t$ at all. Instead, this is the part of the LQR control-input that attempts to drive the system state $\vec{s}_t$ to zero. On the other hand, $\vec{o}_t$ offsets the input so that system state tracks $\vec{r}_t$ instead of tracking zero.

Taking advantage of the linearity described in Eq. 91, let's partition the input and state sequences as follows, over a time-interval $\tau = t$ to $t + N - 1$:

$$\vec{u}_\tau^{\,a} = \vec{z}_\tau = -K_\tau\, \vec{s}_\tau^{\,a}, \text{ with } \vec{s}_t^{\,a} = \vec{s}_t \tag{94}$$

$$\vec{u}_\tau^{\,b} = \vec{o}_\tau = K_\tau^v\, \vec{v}_{\tau+1}^{\,a}, \text{ with } \vec{s}_t^{\,b} = 0 \tag{95}$$

Let's recursively substitute the LQR solution for $\vec{u}_t^{\,a}$:

$$\begin{aligned}
\vec{u}_t^{\,a} &:= -K_t\, \vec{s}_t^{\,a} = -V_{t+1} W_{t+1} A \vec{s}_t \\
&= -V_{t+1} W_{t+1} \vec{\hat{s}}^{\,a}_{t+1|\vec{u}_t^{\,a}=0}
\end{aligned}$$

where $\hat{\vec{s}}^{\,a}_{t+1|\vec{u}_t^a=0} = A\vec{s}_t$ is the system state *estimated* for time $t+1$, using Eq. 17, given *current observed* state $\vec{s}_t$ and applying *zero current input* $\vec{u}_t^a$.

Note that $\hat{\vec{s}}^{\,a}_{t+2|\vec{u}_t^a=0} = (A - BK_{t+1})\,\hat{\vec{s}}^{\,a}_{t+1|\vec{u}_t^a=0}$ is the system state *estimated* for time $t+2$, using Eq. 17, conditioned on *current observed* state $\vec{s}_t$ and the application of *zero current input* $\vec{u}_t^a$. Similarly, $\hat{\vec{y}}^{\,a}_{t|\vec{u}_t^a=0} = C\hat{\vec{s}}^{\,a}_{t+1|\vec{u}_t^a=0}$ is the output *estimated* for time $\tau = t$, conditioned on *current observed* state $\vec{s}_t$ and applying *zero current input* $\vec{u}_t^a$. Applying these substitutions, we get

$$\begin{aligned}
\vec{u}_t^a &:= -V_{t+1}\left[A^T W_{t+2}\,(A - BK_{t+1}) + C^T QC\right]\hat{\vec{s}}^{\,a}_{t+1|\vec{u}_t^a=0} \\
&= -V_{t+1}\left[A^T W_{t+2}\,\hat{\vec{s}}^{\,a}_{t+2|\vec{u}_t^a=0} + C^T Q\,\hat{\vec{y}}^{\,a}_{t|\vec{u}_t^a=0}\right] \\
&= -V_{t+1}\left[A^T\left[A^T W_{t+3}\,(A - BK_{t+2}) + C^T QC\right]\hat{\vec{s}}^{\,a}_{t+2|\vec{u}_t^a=0} + C^T Q\,\hat{\vec{y}}^{\,a}_{t|\vec{u}_t^a=0}\right]
\end{aligned}$$

Reapplying the above mentioned substitutions for another time-step, and generalizing:

$$\begin{aligned}
\vec{u}_t^a &:= -V_{t+1}\left[A^T\left[A^T W_{t+3}\,\hat{\vec{s}}^{\,a}_{t+3|\vec{u}_t^a=0} + C^T Q\,\hat{\vec{y}}^{\,a}_{t+1|\vec{u}_t^a=0}\right] + C^T Q\,\hat{\vec{y}}^{\,a}_{t|\vec{u}_t^a=0}\right] \\
&= -V_{t+1}\left[(A^T)^{N-1} W_{t+N}\,\hat{\vec{s}}^{\,a}_{t+N|\vec{u}_t^a=0} + \sum_{k=0}^{N-2}(A^T)^k C^T Q\,\hat{\vec{y}}^{\,a}_{t+k|\vec{u}_t^a=0}\right] && (96) \\
&= -V_{t+1}\left[\sum_{k=0}^{N-1}(A^T)^k C^T Q\,\hat{\vec{y}}^{\,a}_{t+k|\vec{u}_t^a=0}\right] && (97) \\
&= -V_{t+1}\left[\sum_{k=0}^{N-1}(A^T)^k C^T Q\,\varepsilon\hat{\vec{y}}^{\,a}_{t+k|\vec{u}_t^a=0}\right] - V_{t+1}\left[\sum_{k=0}^{N-1}(A^T)^k C^T Q\,\bar{\vec{y}}\right] && (98)
\end{aligned}$$

We get Eq. 97 from Eq. 96 by recognizing the recursion base-case $W_{t+N} = C^T Q_f C$ and applying our choice of $Q_f = Q$. In Eq. 98, $\varepsilon\hat{\vec{y}}^{\,a}_{t+k|\vec{u}_t^a=0} = \hat{\vec{y}}^{\,a}_{t+k|\vec{u}_t^a=0} - \bar{\vec{y}}$ is the *trajectory-tracking error* estimated for time-step $t+k$ assuming a zero input is applied at current time-step t.

Next, let's examine the offset to the input trajectory, $\vec{u}_t^b = \vec{o}_t$, required to offset the state trajectory to $\vec{r}_t = \bar{\vec{y}}$.

$$
\begin{aligned}
\vec{u}_t^{\,b} &= K_t^v \vec{v}_{t+1} = V_{t+1}\vec{v}_{t+1} \\
&= V_{t+1}\left[(A - BK_{t+1})^T \, \vec{v}_{t+2} + C^T Q \, \vec{r}_{t+1}\right] \\
&= V_{t+1}\left[(A - BK_{t+1})^T \left[(A - BK_{t+2})^T \, \vec{v}_{t+3} + C^T Q \, \vec{r}_{t+2}\right] + C^T Q \, \vec{r}_{t+1}\right] \\
&= V_{t+1}\left[\sum_{k=0}^{N-1} \left(\prod_{p=1}^{k} (A - BK_{t+p})^T\right) C^T Q \, \vec{r}_{t+1+k}\right] \\
&\qquad \text{(recognizing recursion base case } \vec{v}_{t+N} = C^T Q_f \, \vec{r}_{t+N}, \text{ and our choice of } Q_f = Q\text{)} \\
&= V_{t+1}\left[\sum_{k=0}^{N-1} \left(\prod_{p=1}^{k} (A - BK_{t+p})^T\right) C^T Q \, \vec{\tilde{y}}\right] \qquad (99) \\
&\qquad \text{(since we choose a constant trajectory: } \vec{r}_t = \vec{\tilde{y}}\text{)}
\end{aligned}
$$

Combining Eq. 98 and Eq. 99:

$$
\begin{aligned}
\vec{u}_t := V_{t+1} \sum_{k=0}^{N-1} &\left[\left(\prod_{p=1}^{k} (A - BK_{t+p})^T\right) C^T Q \, \vec{\tilde{y}} - (A^T)^k C^T Q \, \vec{\tilde{y}}\right] \\
&- V_{t+1} \sum_{k=0}^{N-1} \left[(A^T)^k C^T Q \, \varepsilon \vec{\tilde{y}}^{\,a}_{t+k|\vec{u}_t^{\,a}=0}\right]
\end{aligned}
$$

By defining the following matrices

$$
E_t \triangleq V_{t+1} \sum_{k=0}^{N-1} \left[\left(\prod_{p=1}^{k} (A - BK_{t+p})^T\right) C^T Q - (A^T)^k C^T Q\right] \qquad (100)
$$

$$
D^{(k,t)} \triangleq V_{t+1}(A^T)^k C^T Q, \qquad \text{for } k = 0 \text{ to } N - 1 \qquad (101)
$$

we get a simplified form

$$\vec{u}_t := E_t\,\vec{\tilde{y}} + \sum_{k=0}^{N-1} D^{(k,t)}\,\varepsilon\vec{\hat{y}}^{\,a}_{t+k|\vec{u}_t{}^a=0} \tag{102}$$

$$= E_t\,\vec{\tilde{y}} + \sum_{k=0}^{N-1} \begin{bmatrix} \vec{d}_1^{(k,t)} & \vec{d}_2^{(k,t)} & \cdots & \vec{d}_m^{(k,t)} \end{bmatrix} \begin{bmatrix} (\varepsilon\hat{y}_1^a)_{t+k|\vec{u}_t{}^a=0} \\ (\varepsilon\hat{y}_2^a)_{t+k|\vec{u}_t{}^a=0} \\ \vdots \\ (\varepsilon\hat{y}_m^a)_{t+k|\vec{u}_t{}^a=0} \end{bmatrix} \tag{103}$$

$$= E_t\,\vec{\tilde{y}} + \sum_{k=0}^{N-1}\sum_{i=1}^{m} \left((\varepsilon\hat{y}_i^a)_{t+k|\vec{u}_t{}^a=0}\right) \vec{d}_i^{(k,t)} \tag{104}$$

$$= E_t\,\vec{\tilde{y}} + \sum_{i=1}^{m} \left[\sum_{k=0}^{N-1} \left((\varepsilon\hat{y}_i^a)_{t+k|\vec{u}_t{}^a=0}\right) \vec{d}_i^{(k,t)}\right] \tag{105}$$

$$= E_t\,\vec{\tilde{y}} + \sum_{i=1}^{m} \begin{bmatrix} \vec{d}_i^{(0,t)} & \vec{d}_i^{(1,t)} & \cdots & \vec{d}_i^{(N-1,t)} \end{bmatrix} \begin{bmatrix} (\varepsilon\hat{y}_i^a)_{t+0|\vec{u}_t{}^a=0} \\ (\varepsilon\hat{y}_i^a)_{t+1|\vec{u}_t{}^a=0} \\ \vdots \\ (\varepsilon\hat{y}_i^a)_{t+N-1|\vec{u}_t{}^a=0} \end{bmatrix} \tag{106}$$

$$= E_t\,\vec{\tilde{y}} + \sum_{i=1}^{m} F^{(i,t)}\ \vec{\hat{err}}^a(y_i;\ t, t+N-1) \tag{107}$$

Here, for a specific output objective y_i, $F^{(i,t)}$ is constructed by selecting the i^{th} columns from the $D^{(k,t)}$ matrices. For a specific y_i, $\vec{\hat{err}}^a(y_i;\ t, t+N-1)$ is the sequence of error-values for y_i estimated using the $\vec{u}_\tau^{\,a}$ and $\vec{s}_\tau^{\,a}$ sequences. Note that $\vec{\hat{err}}^a(y_i;\ t, t+N-1)$ is the only mechanism in LQR for incorporating state feedback about observed values of y_i into $\vec{u}_t$. And, this mechanism weighs the (non-state dependent) columns of matrix $F^{(i,t)}$ based on the *projected error-estimates* for y_i. Hence, the magnitude of the error-terms in $\vec{\hat{err}}^a(y_i;\ t, t+N-1)$ *linearly impacts* the control-inputs that the controller applies to the application with regards to correcting y_i.

By showing linearity, we establish the monotonicity property we sought. The application of adaptive-integral tuning to the tracking error can now be expected to drive application inputs in a manner that drives the tracking error closer to zero in the next time step, even when the regulator does not do so by itself due to modeling approximations.

APPENDIX B

ALGORITHMS

```
Function FrameTransition
  Input: $b$ : budget allocated for operation of $\mathcal{R}$
  Data:
   $t$                                      : current frame number
   $\mathcal{M}$                            : currently active model
   $\mathcal{C}$                            : currently active feedback controller
   $\mathcal{H}$                            : history of application control inputs and response till frame $t$
                                                $\mathcal{H} = \{(\vec{x}_{t-k}, \vec{y}_{t-k}) \mid k \in [1, |\mathcal{H}|]\}$
   $\vec{\mu}$, $\vec{\sigma}$              : mean and std statistics on $\vec{x}$ samples in $\mathcal{H}$, updated incrementally
   $\vec{\chi}$, $\vec{v}_{min}$, $\vec{v}_{max}$ : max swing statistics on $\vec{x}$ samples in $\mathcal{H}$, updated incrementally
   $\gamma, L_\gamma$                       : current history forget-rate parameter and desired history length
   $\vec{x}_{t-1}$                          : control input applied last frame
   $\vec{y}_{t-1}$                          : application QoS output observed for last frame
   $\vec{u}_t$                              : continuous, unclipped control input generated for this frame
   $\vec{x}_t$                              : discretized, clipped control input to be applied this frame
   $\vec{s}_{t-1}, \vec{s}_t$               : previous and current states of the LDS implied by $\mathcal{M}$
   $\vec{\beta}$                            : adaptive scaling-parameters to compensate for drift between
                                               estimated linear model and observed application characteristics
   lastExp                                  : last frame on which inputs were explored
   prevDuration                             : time taken by previous invocation of $FrameTransition()$
1  startTime ← time()
2  $\vec{y}_{t-1}$ ← ReadFrameOutputs()
3  $\mathcal{H} \leftarrow (\vec{x}_{t-1}, \vec{y}_{t-1}){+}{+}\mathcal{H}$
4  $\vec{\mu}$, $\vec{\sigma}$, $\vec{\chi}$, $\vec{v}_{min}$, $\vec{v}_{max}$ ← AddSample($\vec{x}_{t-1}$, $\vec{\mu}$, $\vec{\sigma}$, $\vec{\chi}$, $\vec{v}_{min}$, $\vec{v}_{max}$, $|\mathcal{H}|-1, \gamma$)
5  coverageMet, $\kappa$ ← CheckHistoryCoverage($\vec{\sigma}$, $\vec{\chi}$, $|\mathcal{H}|$, $\gamma$)
6  $\vec{s}_t, \vec{\beta}$ ← AdaptiveStateTransition($\vec{x}_{t-1}$, $\vec{y}_{t-1}$, $\vec{s}_{t-1}$, $\vec{\beta}$, lastExp)
7  $b_{\text{usable}}$ ← BudgetAllocate($b$, prevDuration)
8  deadline ← startTime + $b_{\text{usable}}$
9  doPFE, remainingLen ←
   ProbForcedExploration(new$_\mathcal{M}$, apply$_\mathcal{C}$, coverageMet, $\vec{\mu}$, $\vec{\sigma}$, $|\mathcal{H}|$, $\gamma$, $L_\gamma$)
10 new$_\mathcal{M}$ ←
   UpdateModel(startTime, deadline, $\vec{\beta}$, coverageMet, $\gamma$, $L_\gamma$, $\vec{x}_{t-1}$, $\vec{y}_{t-1}$, doPFE)
11 new$_\mathcal{C}$ ← UpdateRegulator(deadline, $\vec{s}_t$)
12 if new$_\mathcal{C}$ then  $\vec{\beta} \leftarrow \vec{1}$
13 apply$_\mathcal{C}$ ← ($\mathcal{C} \neq \phi$ ∧ IsStateFullySetup($\vec{s}_t$) ∧ doPFE = false)
14 if apply$_\mathcal{C}$ then
15     $\vec{u}_t \leftarrow \mathcal{C}(\vec{s}_t)$
16 else
17     $\vec{u}_t$ ← InputExplorer( coverageMet, $\vec{\mu}$, $\vec{\sigma}$, $\vec{\chi}$, $\vec{x}_{t-1}$, $|\mathcal{H}|$ )
18     lastExp ← $t$
19 $\gamma, L_\gamma$ ← ResizeHistory($\vec{x}_{t-1}$, $\vec{y}_{t-1}$, new$_\mathcal{M}$, $\kappa$, doPFE, remainingLen)
20 $\vec{x}_t$ ← DiscretizeAndClipInputBoundViolations($\vec{u}_t$)
21 ApplyControlInput( $\vec{x}_t$ )
22 $t \leftarrow t + 1$
23 prevDuration ← time() − startTime
24 Return control to application to execute next frame.
```

```
Function UpdateModel(startTime, deadline, $\vec{\beta}$, coverageMet, $\gamma$, $L_\gamma$, $\vec{x}_{t-1}$, $\vec{y}_{t-1}$, doPFE)
  Data:
    $\lambda, \mathcal{M}, \eta, \rho, e, t_{\mathcal{M}}$              : active model, metrics and estimation timestep
    $\lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'}$        : substitute model, metrics and estimation timestep
    $\text{new}_{\mathcal{M}}, \text{new}_{\mathcal{M}'}$              : indicates if a new $\mathcal{M}$ or $\mathcal{M}'$ is applied this frame
    sameModel                                      : indicates if $\mathcal{M}$ and $\mathcal{M}'$ are the same
    $\text{Adv}_{\mathcal{M}',\mathcal{M}}$                            : prediction accuracy advantage of $\mathcal{M}'$ over $\mathcal{M}$
    $\vec{\mu}', \vec{\sigma}', \vec{\chi}', \vec{v}'_{min}, \vec{v}'_{max}$, count   : statistics on $\vec{x}$ since last $\mathcal{M}$ or $\mathcal{M}'$ estimated/applied
  Initially: $\mathcal{M} \leftarrow \phi$
1 if
  $\mathcal{M} \neq \phi \wedge \big((\texttt{MTE}(\mathcal{H}, \mathcal{M}) \gg e \wedge t - t_{\mathcal{M}} > W) \vee ((\eta, \rho) \textit{ unbalanced } \wedge t - t_{\mathcal{M}} \gg L_\gamma)\big)$
  then $\mathcal{M}, \mathcal{C} \leftarrow \phi, \phi$
2 $\text{new}_{\mathcal{M}}, \text{new}_{\mathcal{M}'} \leftarrow \text{false}, \text{false}$
3 if coverageMet = true then
4     if $\mathcal{M} = \phi$ then
5         $\lambda, \mathcal{M}, \eta, \rho, e, t_{\mathcal{M}} \leftarrow \texttt{LLSEandRefineLambda}(\mathcal{H}, \gamma)$
6         $\lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'} \leftarrow \lambda, \mathcal{M}, \eta, \rho, e, t_{\mathcal{M}}$
7         $\text{new}_{\mathcal{M}}, \text{new}_{\mathcal{M}'}$, sameModel $\leftarrow$ true, true, true
8     while $\texttt{time}() < (\text{startTime} + \text{deadline})/2 \wedge (\eta', \rho')$ unbalanced do
9         $\lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'} \leftarrow \texttt{LLSEandRefineLambda}(\mathcal{H}, \gamma)$
10        $\text{new}_{\mathcal{M}'}$, sameModel $\leftarrow$ true, false
11    if $(\eta, \rho)$ unbalanced $\wedge$ $(\eta', \rho')$ balanced then
12        $\lambda, \mathcal{M}, \eta, \rho, e, t_{\mathcal{M}} \leftarrow \lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'}$
13        $\text{new}_{\mathcal{M}}$, sameModel $\leftarrow$ true, true
14 if $\mathcal{M} \neq \phi$ then
15    if $\text{new}_{\mathcal{M}} = \text{false} \wedge \text{new}_{\mathcal{M}'} = \text{false}$ then
16        $e_{t-1} \leftarrow \frac{1}{dim(\vec{y})}||\mathcal{M}(\mathcal{H}, t-1) - \vec{y}_{t-1}||^2_{(\vec{s},\vec{\delta})}$;  $e'_{t-1} \leftarrow \frac{1}{dim(\vec{y})}||\mathcal{M}'(\mathcal{H}, t-1) - \vec{y}_{t-1}||^2_{(\vec{s},\vec{\delta})}$
17        $\text{Adv}_{\mathcal{M}',\mathcal{M}} \leftarrow \dfrac{e_{t-1} - e'_{t-1}}{\texttt{max}(e_{t-1}, e'_{t-1})} + \gamma\, \text{Adv}_{\mathcal{M}',\mathcal{M}}$
18        count $\leftarrow$ count + 1
19        coverageMet' $\leftarrow$ coverageMet
20        if count $< |\mathcal{H}|$ then
21            $\vec{\mu}', \vec{\sigma}', \vec{\chi}', \vec{v}'_{min}, \vec{v}'_{max} \leftarrow \texttt{AddSample}(\vec{x}_{t-1}, \vec{\mu}', \vec{\sigma}', \vec{\chi}', \vec{v}'_{min}, \vec{v}'_{max}, \text{count}-1, \gamma)$
              coverageMet', $\kappa' \leftarrow \texttt{CheckHistoryCoverage}(\vec{\sigma}', \vec{\chi}', \text{count}, \gamma)$
22        Threshold $\leftarrow \dfrac{1 - \gamma^{\texttt{max}(L_\gamma, \text{count})}}{1 - \gamma} \times 0.10$
23        if $\text{Adv}_{\mathcal{M}',\mathcal{M}} >$ Threshold $\wedge$ coverageMet' = true then
24            $\lambda, \mathcal{M}, \eta, \rho, e, t_{\mathcal{M}} \leftarrow \lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'}$
25            $\text{new}_{\mathcal{M}}$, sameModel $\leftarrow$ true, true
26    if $\texttt{time}() <$ deadline $\wedge$ coverageMet = true $\wedge$ doPFE = false
      $\wedge\, \mathcal{C} \neq \phi \wedge t_{\mathcal{C}} = t - 1 \wedge (t_{\mathcal{M}'} \leq t - L_\gamma \vee$ sameModel = true) then
27        $\lambda', \mathcal{M}', \eta', \rho', e', t_{\mathcal{M}'} \leftarrow \texttt{LLSEandRefineLambda}(\mathcal{H}, \gamma)$
28        $\text{new}_{\mathcal{M}'}$, sameModel $\leftarrow$ true, false
29 (continued on next page)
```

Function UpdateModel(continued)

```
30 if $\text{new}_{\mathcal{M}} \vee \text{new}_{\mathcal{M}'}$ then
31     $\text{Adv}_{\mathcal{M}',\mathcal{M}} \leftarrow 0.0$
32     count $\leftarrow 0$
33 if $\text{new}_{\mathcal{M}} = \text{true}$ then $\mathcal{C} \leftarrow \phi$
34 return $\text{new}_{\mathcal{M}}$
```

Function CheckHistoryCoverage($\vec{\sigma}^h, \vec{\chi}^h, \text{len}^h, \gamma$)

```
Input:
  $\vec{\sigma}^h, \vec{\chi}^h, \text{len}^h$ : standard deviation, max swing and sample length for some history $h$
  $\gamma$                    : forget rate used in history $h$
1 if $\text{len}^h \leq y_{order}$ or $\text{len}^h \leq x_{order}$ or $\text{len}^h \leq W$ then
2     return false, 0
3 numSpanningDims $\leftarrow 0$
4 for $j \in [1, dim(\vec{x})]$ do
5     if $\sigma^h_j \geq 0.5\, N_j \;\wedge\; \chi^h_j \geq \frac{1+\gamma}{2} N_j$ then
6         numSpanningDims $\leftarrow$ numSpanningDims $+ 1$
7 $\kappa \leftarrow \frac{\text{numSpanningDims}}{dims(\vec{x})}$
8 if $\kappa = 1$ then coverageMet = true else coverageMet = false
9 return coverageMet, $\kappa$
```

Function LLSEandRefineLambda(h, γ)

```
Data: $\lambda_{\text{next}}$ : value to use for next LLSE invocation
Initially: $\lambda_{\text{next}} \leftarrow$ default
1 $\lambda_m \leftarrow \lambda_{\text{next}}$
2 $m, \eta_m, \rho_m \leftarrow \texttt{LLSE}(h, \lambda_m, \gamma)$
3 $t_m \leftarrow t$
4 $e_m \leftarrow \texttt{MTE}(h, m)$
5 if $\rho_m > 0$ then
6     $\lambda_{\text{next}} \leftarrow \sqrt{\frac{\eta_m \lambda_m}{\rho_m}}$
7 if $\rho_m = 0 \vee \lambda_{next} = \infty$ then
8     if $\lambda > 1$ then
9         $\lambda \leftarrow \sqrt{\lambda}$
10    else
11        $\lambda \leftarrow \lambda^2$
12 if $\lambda = 0$ then
13     $\lambda \leftarrow 10^{-6}$
14 return $(\lambda_m, m, \eta_m, \rho_m, e_m, t_m)$
```

```
Function MTE($h$,$m$)
    Input:  $h = \{\ (\vec{x}_{t_0-k}, \vec{y}_{t_0-k}) \mid k \in \big[1, |h|\big]\ \}$  :  History till some time $t_0$
            $m$                                                                          :  A linear model
    Data: $\gamma$ : current history sample forget rate
1   Let $m(h,t') = \vec{\tilde{y}}_{t'}$ for $t' \in \big[t_0 - |h| + y_{order},\ t_0 - 1\big]$ denote the *evaluation* of the model
    $m$ to produce a prediction at time $t'$ using observed history data till before $t'$ and the
    applied input at $t'$
2   errorConvergingSum $= \frac{1-\gamma}{dim(\vec{y})} \sum_{k=1}^{|h|-y_{order}} \gamma^{k-1} \sum_{i=1}^{dim(\vec{y})} s_i \frac{(y_{i|t_0-k} - \tilde{y}_{i|t_0-k})^2}{\delta_i^2}$
3   return errorConvergingSum
```

Function UpdateRegulator(deadline, $\vec{s}_t$)

Input:

deadline	:	current frame's deadline for non-critical work
$\vec{s}_t$	:	current state

Result: Construct $\mathcal{C}$ from $\mathcal{M}$ and refine based on projected inputs

Data:

t	:	current frame number
$\mathcal{C},\ t_{\mathcal{C}}$	:	the active controller and its last refinement timestep
$\text{new}_{\mathcal{C}}$	:	indicates if a new controller is created this frame from $\mathcal{M}$
R	:	input-costs diagonal matrix used for the LQR design of $\mathcal{C}$
$R^{\text{p}}, \vec{x}^{\text{p}}$	:	the previous input-costs matrix and input
$R^{\text{r}}, \vec{x}^{\text{r}}$	:	the refined input-costs matrix and input
$\vec{\tilde{y}}_t, TT^{\text{r}}$	:	projected output on input $\vec{x}^{\text{r}}$ and resulting trajectory-tracking error
$\vec{f}^{\text{ bu}}$	:	flag if each input dimension is barely under-constrained
$\vec{f}^{\text{ term}}$	:	flag if each input dimension is ready to terminate refinement
$\vec{c}$	:	counters used for updating corresponding diagonal entries of R

Initially: $\mathcal{C} \leftarrow \phi,\ R \leftarrow I_{dim(\vec{x})\times dim(\vec{x})},\ \vec{f}^{\text{ bu}} \leftarrow \vec{\text{false}},\ \vec{c} \leftarrow \vec{1}$

```
1  new_C ← false
2  if $\mathcal{M} \neq \phi \ \wedge\ (\texttt{time}() < \text{deadline} \vee \mathcal{C} = \phi)$ then
3      if $\mathcal{C} = \phi$ then
4          $\mathcal{C} \leftarrow \texttt{LQR}(\phi, \mathcal{M}, R)$;    $t_{\mathcal{C}} \leftarrow t$
5          $\text{new}_{\mathcal{C}} \leftarrow$ true
6          $\vec{f}^{\text{ bu}} \leftarrow \vec{\text{false}}$
7      if $\texttt{IsStateFullySetup}(\vec{s}_t) = \text{true} \ \wedge\ \texttt{time}() < \text{deadline}$ then
8          $R^{\text{p}},\ \vec{x}^{\text{p}} \leftarrow R,\ \mathcal{C}(\vec{s}_t)$
9          for $1 \leq j \leq n$ do
10             $R_{jj} \leftarrow \texttt{InitialRefinement}(f^{\text{bu}}_j,\ |x^{\text{p}}_j|,\ R^{\text{p}}_{jj},\ c_j)$
11         $\mathcal{C} \leftarrow \texttt{LQR}(\mathcal{C}, \mathcal{M}, R)$;    $t_{\mathcal{C}} \leftarrow t$
12         $R^{\text{r}},\ \vec{x}^{\text{r}} \leftarrow R,\ \mathcal{C}(\vec{s}_t)$
13         $\vec{f}^{\text{ term}} \leftarrow \vec{\text{false}}$
14         repeat
15             $\vec{\tilde{y}}_t \leftarrow \mathcal{M}(\vec{s}_t,\ \vec{x}^{\text{r}})$
16             $TT^{\text{r}} \leftarrow (\vec{\tilde{y}}_t - \vec{\tilde{y}})^T\, Q\, (\vec{\tilde{y}}_t - \vec{\tilde{y}})$
17             for $1 \leq j \leq n$ do
18                 $R_{jj},\ f^{\text{term}}_j,\ f^{\text{bu}}_j,\ c_j \leftarrow$
19                     $\texttt{RefineInputCost}(f^{\text{term}}_j,\ f^{\text{bu}}_j,\ |x^{\text{p}}_j|,\ |x^{\text{r}}_j|,\ R^{\text{p}}_{jj},\ R^{\text{r}}_{jj},\ c_j,\ TT^{\text{r}})$
20             $\mathcal{C} \leftarrow \texttt{LQR}(\mathcal{C}, \mathcal{M}, R)$;    $t_{\mathcal{C}} \leftarrow t$
21             $R^{\text{p}},\ \vec{x}^{\text{p}} \leftarrow R^{\text{r}},\ \vec{x}^{\text{r}}$
22             $R^{\text{r}},\ \vec{x}^{\text{r}} \leftarrow R,\ \mathcal{C}(\vec{s}_t)$
23         until $\forall j\ f^{term}_j = \text{true}$ or $\texttt{time}() \geq \text{deadline}$
24 return $\text{new}_{\mathcal{C}}$
```

Function InitialRefinement($f_j^{\mathrm{bu}}, |x_j^{\mathrm{p}}|, R_{jj}^{\mathrm{p}}, c_j$)

```
1 if $f_j^{bu}$ = false then
2     $c_j \leftarrow 1$
3 if $|x_j^p| > N_j$ then
4     $R_{jj} \leftarrow$ IncreaseInputCost($R_{jj}^{\mathrm{p}}, c_j$)
5 else
6     $R_{jj} \leftarrow$ DecreaseInputCost($R_{jj}^{\mathrm{p}}, c_j$)
7 return $R_{jj}$
```

Function RefineInputCost($f_j^{\mathrm{term}}, f_j^{\mathrm{bu}}, |x_j^{\mathrm{p}}|, |x_j^{\mathrm{r}}|, R_{jj}^{\mathrm{p}}, R_{jj}^{\mathrm{r}}, c_j, TT^{\mathrm{r}}$)

```
1  if $f_j^{bu} = \text{false}$ then  $c_j \leftarrow 1$
2  if $f_j^{term} = \text{true}$ then
3      assert($f_j^{\mathrm{bu}} = \text{false}$)
4      $R_{jj}, f_j^{\mathrm{term}} \leftarrow$ EvaluateTerminatedRefinement($f_j^{\mathrm{term}}$, $|x_j^{\mathrm{p}}|$, $|x_j^{\mathrm{r}}|$, $R_{jj}^{\mathrm{p}}$, $R_{jj}^{\mathrm{r}}$, $c_j$)
5      return $R_{jj}, f_j^{term}, f_j^{bu}, c_j$
6  assert($f_j^{\mathrm{term}} = \text{false and } R_{jj}^{\mathrm{p}} \neq R_{jj}^{\mathrm{r}}$)
7  if $|x_j^p| \leq N_j \;\wedge\; |x_j^r| \leq N_j$ then
8      if PracticallyEqual($|x_j^p|$, $|x_j^r|$) then
9          if PracticallyEqual($|x_j^r|$, 0) $\wedge$ $(0.5)^2 * R_{jj}^r > 10 * TT^r$ then
10             $R_{jj} \leftarrow$ DecreaseInputCost($R_{jj}^{\mathrm{r}}, c_j$)
11         else
12             $R_{jj} \leftarrow R_{jj}^{\mathrm{p}}$ ; $f_j^{\mathrm{bu}} \leftarrow$ false ; $f_j^{\mathrm{term}} \leftarrow$ true
13     else if $(R_{jj}^p - R_{jj}^r) * (|x_j^p| - |x_j^r|) > 0$ then
14         $R_{jj} \leftarrow R_{jj}^{\mathrm{p}}$ ; $f_j^{\mathrm{bu}} \leftarrow$ false
15     else if $(R_{jj}^p - R_{jj}^r) * (|x_j^p| - |x_j^r|) < 0$ then
16         $R_{jj} \leftarrow$ DecreaseInputCost($R_{jj}^{\mathrm{r}}, c_j$)
17 if $|x_j^p| \leq N_j \;\wedge\; |x_j^r| > N_j$ then
18     if $R_{jj}^p > R_{jj}^r$ then
19         $f_j^{\mathrm{bu}} \leftarrow$ true ; $c_j \leftarrow c_j + 1$ ; $R_{jj} \leftarrow$ IncreaseInputCost($R_{jj}^{\mathrm{r}}, c_j$)
20     if $R_{jj}^p < R_{jj}^r$ then
21         $R_{jj} \leftarrow R_{jj}^{\mathrm{p}}$ ; $f_j^{\mathrm{bu}} \leftarrow$ false
22 if $|x_j^p| > N_j \;\wedge\; |x_j^r| \leq N_j$ then
23     if $R_{jj}^p < R_{jj}^r$ then
24         $f_j^{\mathrm{bu}} \leftarrow$ true ; $c_j \leftarrow c_j + 1$ ; $R_{jj} \leftarrow$ DecreaseInputCost($R_{jj}^{\mathrm{r}}, c_j$)
25     if $R_{jj}^p > R_{jj}^r$ then
26         $R_{jj} \leftarrow R_{jj}^{\mathrm{p}}$ ; $f_j^{\mathrm{bu}} \leftarrow$ false
27 if $|x_j^p| > N_j \;\wedge\; |x_j^r| > N_j$ then
28     if $(R_{jj}^p - R_{jj}^r) * (|x_j^p| - |x_j^r|) \leq 0$ then
29         $R_{jj} \leftarrow$ IncreaseInputCost($R_{jj}^{\mathrm{r}}, c_j$)
30     else
31         $R_{jj} \leftarrow R_{jj}^{\mathrm{p}}$ ; $f_j^{\mathrm{bu}} \leftarrow$ false
32 return $R_{jj}, f_j^{term}, f_j^{bu}, c_j$
```

Function EvaluateTerminatedRefinement($f_j^{\text{term}}, |x_j^{\text{p}}|, |x_j^{\text{r}}|, R_{jj}^{\text{p}}, R_{jj}^{\text{r}}, c_j$)

```
1 if $R_{jj}^{p} \neq R_{jj}^{r}$ then
2     $R_{jj} \leftarrow R_{jj}^{\text{r}}$
3 else
4     if PracticallyEqual($|x_j^p|,\ |x_j^r|$) $=$ true then
5         $R_{jj} \leftarrow R_{jj}^{\text{r}}$
6     else
7         $R_{jj} \leftarrow$ DecreaseInputCost($R_{jj}^{\text{r}}, c_j$)
8         $f^{\text{term}} \leftarrow$ false
9 return $R_{jj}, f_j^{term}$
```

Function IncreaseInputCost(R_{jj}^{r}, c_j)

```
1 $R_{jj} \leftarrow R_{jj}^{\text{r}} * \left(1 + \frac{1.0}{c_j}\right)$
2 return $R_{jj}$
```

Function DecreaseInputCost(R_{jj}^{r}, c_j)

```
1 $R_{jj} \leftarrow R_{jj}^{\text{r}} / \left(1 + \frac{1.0}{c_j}\right)$
2 return $R_{jj}$
```

Function PracticallyEqual($x_j^{\text{p}}, x_j^{\text{r}}$)

```
1 if $|x_j^p - x^r| * (10 * W) < 0.5$ then
2     return true
3 else
4     return false
```

```
Function ProbForcedExploration(prevNew_$\mathcal{M}$, prevApplied_$\mathcal{C}$, coverageMet, $\vec{\mu}$, $\vec{\sigma}$,
histLength, $\gamma$, $L_\gamma$)
  Data:
    $q,\ d$                         : probability to start a new PFE this frame, cluster length
    remainingLen                    : length of PFE cluster remaining (if previously started)
    $\theta$                        : fraction of application frames given to PFE
    $\tau_{t-1}$                    : the QoS performance of the controller in the last frame
    $l_{\max}$                      : Number of performance levels to maintain SR statistics
    $\forall k \in \{(1.50)^l \mid l \in \{0,\ \ldots,\ l_{\max}\}\}$ :
      $\hat{\mathtt{SR}}_{\mathcal{M}}(k),\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}}(k)$     : estimated full and current SR of $\mathcal{M}$ on $\tau \le k$
      $\hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}}(k),\ \hat{\mathtt{SR}}_{\text{Ach}}(k)$  : Set of $\hat{\mathtt{SR}}_{\mathcal{M}}(k)$ observed, computed achievability
      $Q(k)$                        : queue of $\tau \le k$ tests for the $10 * W$ most recent frames
    $k_{\text{tr}}, l_{\text{tr}}$  : the current SR performance tracking level, and its index
    $\text{nf}_{\mathcal{M}}$       : number of frames where $\mathcal{M}$ drove inputs, after initial $W$
    numClusters, clIdx              : num-clusters to split $L_{\text{PFE}}$ into, cluster index in group
    $\tau_{\max}$                   : the maximum $\tau_{t-1}$ encountered so far
  Initially: $l_{\max} \leftarrow -1$; $\hat{\mathtt{SR}}_{\mathcal{M}},\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}},\ \hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}},\ \hat{\mathtt{SR}}_{\text{Ach}},\ Q \leftarrow \phi,\ \phi,\ \phi,\ \phi,\ \phi$; $\text{nf}_{\mathcal{M}} \leftarrow 1$;
             remainingLen $\leftarrow 0$; numClusters $\leftarrow 1$; clIdx $\leftarrow 0$; $\tau_{\max} \leftarrow 1.0$
1  $\tau_{\max} \leftarrow \mathtt{max}\{\tau_{\max},\ \tau_{t-1}\}$; $l'_{\max} \leftarrow \left\lceil \frac{\log \tau_{\max}}{\log 1.50} \right\rceil$
2  for $l \in \{l_{max}+1, \ldots, l'_{max}\}$ do
3      $k \leftarrow (1.50)^l$
4      $\hat{\mathtt{SR}}_{\mathcal{M}}(k),\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}}(k),\ \hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}}(k),\ \hat{\mathtt{SR}}_{\text{Ach}}(k),\ Q(k) \leftarrow 1,\ 1,\ \{1\},\ 1,\ [1]$
5  $l_{\max} \leftarrow l'_{\max}$
6  if prevNew_$\mathcal{M}$ = true $\wedge$ $\text{nf}_{\mathcal{M}} > 1$ then
7      if $\text{nf}_{\mathcal{M}}$ not divisible by $10 * W$ then
8          $\hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}},\ \hat{\mathtt{SR}}_{\text{Ach}} \leftarrow \mathtt{UpdateAchievability}(\hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}},\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}},\ l_{\max})$
9      for $k \in \{(1.50)^l \mid l \in \{0,\ \ldots,\ l_{max}\}\}$ do
10         $\hat{\mathtt{SR}}_{\mathcal{M}}(k),\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}}(k),\ Q(k) \leftarrow 1,\ 1,\ [1]$
11     $\text{nf}_{\mathcal{M}} \leftarrow 1$
12 if prevApplied_$\mathcal{C}$ = true then
13     if $t - t_{\mathcal{M}} \ge W$ then
14         for $k \in \{(1.50)^l \mid l \in \{0,\ \ldots,\ l_{max}\}\}$ do
15             $\hat{\mathtt{SR}}_{\mathcal{M}}(k) \leftarrow \frac{\hat{\mathtt{SR}}_{\mathcal{M}}(k) * \text{nf}_{\mathcal{M}} + I\{\tau_{t-1} \le k\}}{\text{nf}_{\mathcal{M}} + 1}$; $Q(k)$.append($I\{\tau_{t-1} \le k\}$);
               $\hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}}(k) \leftarrow$ Average($Q(k)$);
16         $\text{nf}_{\mathcal{M}} \leftarrow \text{nf}_{\mathcal{M}} + 1$;
17     if $\text{nf}_{\mathcal{M}}$ divisible by $10 * W$ then
18         $\hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}},\ \hat{\mathtt{SR}}_{\text{Ach}} \leftarrow \mathtt{UpdateAchievability}(\hat{\mathtt{SR}}^{\text{set}}_{\text{Ach}},\ \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}},\ l_{\max})$
19     $l_{\text{tr}} \leftarrow \underset{0 \le l < l_{\max}}{\text{argmax}} \frac{\hat{\mathtt{SR}}_{\text{Ach}}((1.50)^{l+1}) - \hat{\mathtt{SR}}_{\text{Ach}}((1.50)^l)}{l+1}$;   $k_{\text{tr}} \leftarrow (1.50)^{l_{\text{tr}}}$
20     $\theta \leftarrow \frac{1}{2}\,(1 - \hat{\mathtt{SR}}^{\text{cur}}_{\mathcal{M}}(k_{\text{tr}}))\ \mathtt{max}\left(\frac{\hat{\mathtt{SR}}_{\text{Ach}}(k_{\text{tr}}) - \hat{\mathtt{SR}}_{\mathcal{M}}(k_{\text{tr}})}{\hat{\mathtt{SR}}_{\text{Ach}}(k_{\text{tr}}) + 0.01},\ 0\right) + 0.01$
21 (continued on next page)
```

Function ProbForcedExploration(continued)

```
22 if remainingLen $> 0$ then
       if clIdx $<$ numClusters $\vee$ remainingLen $> 1$ $\vee$ coverageMet $=$ true then
           remainingLen $\leftarrow$ remainingLen $- 1$
       if clIdx $=$ numClusters $\wedge$ remainingLen $= 0$ then
           clIdx $\leftarrow 0$
       return true, remainingLen
28 if coverageMet $=$ true then
       return false, 0
30 $L_{\text{PFE}} \leftarrow$ EstimateShortestPFEClusterLength($\vec{\mu}$, $\vec{\sigma}$, histLength, $\gamma$, $L_\gamma$)
31 $q' \leftarrow \phi$
32 if numClusters $> L_{PFE}$ then
       numClusters $\leftarrow L_{\text{PFE}}$
34 while true do
       $d_{\text{peak}} \leftarrow \left\lfloor \frac{L_{\text{PFE}}}{\text{numClusters}} \right\rfloor$
       $Ed, d \leftarrow$ AdjustClusterLengthDistributionAndSample($d_{\text{peak}}$, $L_\gamma$)
       $q \leftarrow \frac{\theta}{Ed\,(1-\theta)+\theta}$
       if $q' \neq \phi \wedge q$ and $q'$ straddle $\frac{\theta}{5}$ then
           break
       if $q < \frac{\theta}{5}$ then
           if numClusters $= L_{PFE}$ then
               break
           numClusters $\leftarrow$ numClusters $+ 1$
       else
           if numClusters $= 1$ then
               break
           numClusters $\leftarrow$ numClusters $- 1$
       $q' \leftarrow q$
49 if $q \leq$ random(0.0, 1.0) then
       return false, 0
51 remainingLen $\leftarrow d - 1$
52 clIdx $\leftarrow$ min(clIdx $+ 1$, numClusters)
53 return true, remainingLen
```

Function UpdateAchievability($\hat{\texttt{SR}}^{\text{set}}_{\text{Ach}}$, $\hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}$, $l_{\max}$)

```
1 for $k \in \{(1.50)^l \mid l \in \{0, \ldots, l_{max}\}\}$ do
      $\hat{\texttt{SR}}^{\text{set}}_{\text{Ach}}(k) \leftarrow \hat{\texttt{SR}}^{\text{set}}_{\text{Ach}}(k) \uplus \{\hat{\texttt{SR}}^{\text{cur}}_{\mathcal{M}}(k)\}$
      $\hat{\texttt{SR}}_{\text{Ach}}(k) \leftarrow$ Average of largest 25% in $\hat{\texttt{SR}}^{\text{set}}_{\text{Ach}}(k)$
4 return $\hat{SR}^{\text{set}}_{Ach}$, $\hat{SR}_{Ach}$
```

Function EstimateShortestPFEClusterLength($\vec{\mu}$, $\vec{\sigma}$, histLength, γ, L_γ)

```
1  $j_o \leftarrow \operatorname*{argmin}_j \frac{\sigma_j}{N_j}$
2  $\mu_o, \sigma_o, N_o \leftarrow \mu_{j_o}, \sigma_{j_o}, N_{j_o}$
3  lower, upper $\leftarrow 1, \left\lceil \frac{L_\gamma}{2} \right\rceil$
4  while lower < upper do
5      $L \leftarrow \left\lfloor \frac{\text{lower} + \text{upper}}{2} \right\rfloor$
6      if $L + \text{histLength} \leq L_\gamma$ then
7          $R \leftarrow \text{histLength}$
8      else
9          $R \leftarrow L_\gamma - L$
10     $\mu_e \leftarrow \frac{1}{1-\gamma^{R+L}} \left[ \frac{1-\gamma^L}{1+\gamma}(1-\gamma)N_o + \gamma^L(1-\gamma^R)\mu_o \right]$
11     ${\sigma_e}^2 \leftarrow \frac{1}{1-\gamma^{R+L}} \left[ \frac{1-\gamma^L}{1+\gamma} \left[ (N_o - \mu_e)^2 + \gamma(N_o + \mu_e)^2 \right] + \gamma^L(1-\gamma^R) \left[ {\sigma_o}^2 + \gamma^L(\mu_o - \mu_e)^2 \right] \right]$
12     if $\sigma_e \geq 0.5 N_o$ then
13         upper $\leftarrow L$
14     else if lower + 1 = upper then
15         lower $\leftarrow$ upper
16     else
17         lower $\leftarrow L$
18 $L_{\text{PFE}} \leftarrow$ upper
19 return $L_{PFE}$
```

Function ResizeHistory($\vec{x}_{t-1}$, $\vec{y}_{t-1}$, new$_{\mathcal{M}}$, κ, doPFE, remainingLen)

Data:

$\mathcal{M}$	:	currently active model at frame t
History at t	:	$\mathcal{H} = \left[\ (\vec{x}_{t-k}, \vec{y}_{t-k}) \mid k \in [1, \lvert\mathcal{H}\rvert] \ \right]$
$\vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}$	:	statistics on $\vec{x}$ samples in $\mathcal{H}$
γ	:	history sample forget rate
L_γ, L''	:	length of significant history allowed by γ, candidate L_γ
L_{min}	:	minimal length for history required by user
L_s, L_c	:	length recommendations for stability and coverage
prevFrameEndedPFE	:	was previous frame last frame of a PFE cluster?

Initially: $\gamma \ \leftarrow \ (0.9)^{1/W}$, $L_\gamma \ \leftarrow \ \dfrac{\log 0.1}{\log \gamma}$, $L'' \ \leftarrow \ L_\gamma$, prevFrameEndedPFE $\leftarrow$ false,
$L_{min} \ \leftarrow \ \texttt{max}\ \{ \ W, \ 2\ dim(\vec{x}) \ + \ 1 \ \}$ (if not specified by user)

```
1  if $L_\gamma < L_{min} \ \vee \ \mathcal{M} = \phi$ then
2      $L_\gamma \leftarrow \texttt{max}\{L_{min}, (\mathcal{M} = \phi \, ? \, |\mathcal{H}| \ : \ 0)\};\ L'' \leftarrow L_\gamma;\ \gamma \leftarrow (0.1)^{1/L_\gamma}$
3  $\bar{D}$, $L_s$, $t_{\text{bcp}} \leftarrow$ UpdateStabilityLength($\vec{x}_{t-1}$, $\vec{y}_{t-1}$, new$_{\mathcal{M}}$, $t - |\mathcal{H}|$)
4  $L_c \leftarrow$ UpdateCoverageLength(prevFrameEndedPFE, $\kappa$, $|\mathcal{H}|$, $t_{\text{bcp}}$)
5  if $L_s \neq \phi$ then
6      if $L_s \geq L_c$ then  $L' \leftarrow L_s$
7      else
8          coverageStability $\leftarrow$ InterpolatedStability($L_c$, $\bar{D}$, $L_s^{\text{set}}$)
9          if coverageStability = *stable* then
10             $L' \leftarrow L_c$
11         else if coverageStability = *unknown* then
12             $L_s^{\text{set}} \leftarrow$ AddStabilityCandidate($L_c$, $L_s^{\text{set}}$)
13             $L' \leftarrow$ max($L_s$, $L''$)
14         else if coverageStability = *unstable* then
15             $L' \leftarrow \dfrac{L_s + L_c}{2}$
16         else if coverageStability = *highly unstable* then
17             $L' \leftarrow L_s$
18 else  $L' \leftarrow L_c$
19 $L'' \leftarrow \dfrac{L' + L''}{2}$
20 if $|L_\gamma - L''| \geq W \ \wedge \ L'' \geq L_{min}$ then
21     $L_\gamma \leftarrow L''$
22     $\gamma \leftarrow (0.1)^{1/L_\gamma}$
23 if $\mathcal{M} \neq \phi \ \wedge \ (|\mathcal{H}| > L_\gamma \ \vee \ t_{\text{bcp}} \neq \phi)$ then
24     $L_{\text{bcp}} \leftarrow |\mathcal{H}|$
25     if $t_{\text{bcp}} \neq \phi$ then  $L_{\text{bcp}} \leftarrow t - t_{\text{bcp}}$
26     $L_{reduced} \leftarrow$ min{$L_\gamma$, $L_{\text{bcp}}$}
27     for $k = |\mathcal{H}|$ *downto* $L_{reduced}$+1 do
28         $\vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max} \leftarrow$ RemoveSample($\mathcal{H}$, $\vec{\mu}$, $\vec{\sigma}$, $\vec{\chi}$, $\vec{v}_{min}$, $\vec{v}_{max}$, $k$, $\gamma$)
29     $\mathcal{H} \leftarrow \mathcal{H}[1 \,..\, L_{reduced}]$
30 prevFrameEndedPFE $\leftarrow$ (doPFE = true $\wedge$ remainingLen = 0)
31 return $\gamma, L_\gamma$
```

Function UpdateStabilityLength($\vec{x}_{t-1}$, $\vec{y}_{t-1}$, new$_{\mathcal{M}}$, $t_{\mathcal{H}}$)

```
Data:
  $\mathcal{M}$ : Currently active model at frame $t$
  $\mathcal{T}_{xy} = \left[\ (\vec{x}_{t-k}, \vec{y}_{t-k}) \mid k \in [1, |\mathcal{T}_{xy}|]\ \right]$ : Stability Samples History
  $\mathcal{T}_e = \left[\ e_{t-k} \mid k \in [1, |\mathcal{T}_e|]\ \right]$ : Prediction Error History for $\mathcal{M}$
  $L_s^{\text{set}}$ : Candidate lengths for stability, persists across models
  $e_{min}$, $e_{max}$ : The min and max values of $e_t$ seen so far
  reconstruct : Captures the condition that all current histogram data
                has become invalid and must be subsequently re-constructed

Initially: $\mathcal{T}_{xy} \leftarrow []$, $\mathcal{T}_e \leftarrow []$, $L_s^{\text{set}} \leftarrow \{W\}$, $e_{min} \leftarrow \phi$, $e_{max} \leftarrow \phi$, reconstruct $\leftarrow$ true

1  $\mathcal{T}_{xy} \leftarrow (\vec{x}_{t-1}, \vec{y}_{t-1}) \mathbin{++} \mathcal{T}_{xy}$
2  if new$_{\mathcal{M}}$ = true $\vee$ $\mathcal{M} = \phi$ then
3      $\mathcal{T}_e \leftarrow []$
4      reconstruct $\leftarrow$ true
5  if $\mathcal{M} = \phi$ then return $\phi$, $\phi$, $\phi$
6  if $|\mathcal{T}_{xy}| \geq \texttt{max}\{x_{order}, y_{order}\}$ then
7      prev_e$_{min}$, prev_e$_{max}$ $\leftarrow$ $e_{min}$, $e_{max}$
8      numRemaining $\leftarrow |\mathcal{T}_{xy}| - |\mathcal{T}_e| - \texttt{max}\{x_{order}, y_{order}\}$
9      for $k$ = numRemaining downto 1 do
10         $e_{t-k} \leftarrow \frac{1}{dim(\vec{y})} ||\mathcal{M}(\mathcal{T}_{xy}, t-k) - \vec{y}_{t-k}||^2$
11         $\mathcal{T}_e \leftarrow (\, e_{t-k}\,) \mathbin{++} \mathcal{T}_e$
12         $e_{min} \leftarrow \texttt{min}\{e_{min}, e_{t-k}\}$;  $e_{max} \leftarrow \texttt{max}\{e_{max}, e_{t-k}\}$
13     if $(e_{max} - e_{min}) > 1.10\,(\text{prev\_e}_{max} - \text{prev\_e}_{min})$ then
14         reconstruct $\leftarrow$ true
15 repeat
16     $\bar{D}, L_s, t_{\text{bcp}} \leftarrow \texttt{UpdateAveragedKSDistances}(\text{reconstruct}, \mathcal{T}_e, L_s^{\text{set}}, e_{min}, e_{max}, t_{\mathcal{H}})$
17     reconstruct $\leftarrow$ false
18     removeSet, addSet $\leftarrow$ {}, {};   $L_1$, $L_2 \leftarrow \phi$, $\phi$;   $L_s^{\text{max}} \leftarrow \texttt{max}\{L_s^{\text{set}}\}$
19     for $L_3$ in ascending order of $L_s^{set}$ do
20         if $L_1 \neq \phi \wedge L_2 \neq \phi \wedge \texttt{Stability}(\bar{D}(L_1)) = \texttt{Stability}(\bar{D}(L_2)) =$
           $\texttt{Stability}(\bar{D}(L_3)) \neq$ unknown then
21             removeSet $\leftarrow$ removeSet $\cup \{L_2\}$
22         if $L_2 \neq \phi \wedge \texttt{Stability}(\bar{D}(L_2)) \neq$ unknown $\wedge$ $\texttt{Stability}(\bar{D}(L_3)) \neq$
           unknown $\wedge$ $\texttt{Stability}(\bar{D}(L_2)) \neq \texttt{Stability}(\bar{D}(L_3))$ then
23             $L \leftarrow \frac{L_2 + L_3}{2}$ rounded to closest multiple of $W$
24             if $L \neq L_2 \wedge L \neq L_3$ then
25                 addSet $\leftarrow$ addSet $\cup \{L\}$
26         $L_1$, $L_2 \leftarrow L_2$, $L_3$
27     if $L_s = \phi \wedge |\mathcal{T}_e| > 2L_s^{max}$ then
28         addSet $\leftarrow$ addSet $\cup \{2L_s^{\text{max}}\}$
29     $L_s^{\text{set}} \leftarrow (L_s^{\text{set}} - \text{removeSet}) \cup \text{addSet}$
30 until removeSet = {} $\wedge$ addSet = {}
31 if $|\mathcal{T}_e| > 2L_s^{max}$ then drop $(|\mathcal{T}_e| - 2L_s^{\text{max}})$ oldest samples from $\mathcal{T}_{xy}$ and $\mathcal{T}_e$
32 return $\bar{D}$, $L_s$, $t_{\text{bcp}}$
```

Function UpdateAveragedKSDistances(reconstruct, $\mathcal{T}_e$, L_s^{set}, e_{min}, e_{max}, $t_{\mathcal{H}}$)

Data:

histogram(L)	:	Histogram of last completed segment of length $L \in L_s^{\text{set}}$ in $\mathcal{T}_e$
timestamp(L) $= t'$	:	Segment $[t'-L+1,\, t']$ of $\mathcal{T}_e$ over which histogram(L) was computed
$D(L)$	:	Sequence $\big[D(L)[i] \,\big|\, i \in [1,\, \lvert D(L)\rvert]\big]$ of the most recent K-S distances for $L \in L_s^{\text{set}}$ for current $\mathcal{M}$
$t_D(L)$	:	Sequence $\big[t_D(L)[i] \,\big|\, i \in [1,\, \lvert D(L)\rvert]\big]$ of corr. timestamps
$\bar{D}(L)$	:	Averaged K-S metrics for $L \in L_s^{\text{set}}$ for current $\mathcal{M}$
$t_{\text{bcp}}^{\text{prev}}$	:	Previous t_{bcp} found. $= \phi$ initially

```
1  if reconstruct = true then
2      histogram ← $\phi$; timestamp ← $\phi$; $D \leftarrow \phi$; $t_D \leftarrow \phi$; $\bar{D} \leftarrow \phi$; $t_{\text{bcp}}^{\text{prev}} \leftarrow \phi$
3  for $L \in L_s^{set}$ do
4      if $|\mathcal{T}_e| \geq L$ then
5          if timestamp($L$) $= \phi \ \vee$ timestamp($L$) $< t - |\mathcal{T}_e| - 1$ then
6              histTimestep ← $t - |\mathcal{T}_e| + L - 1$
7          else
8              histTimestep ← timestamp($L$) $+ L$
9          numAppends ← 0
10         for $t' \leftarrow$ histTimestep; $t' \leq t - 1$; $t' \leftarrow t' + L$ do
11             if $|D(L)| = 0$ then resetBins ← true else resetBins ← false
12             newhist ← ConstructHistogram($\mathcal{T}_e$, $t'$, $L$, resetBins, $e_{min}$, $e_{max}$)
13             if timestamp($L$) $\neq \phi$ then
14                 newD ← KolmogorovSmirnovD(newhist, histogram($L$), $L$)
15                 $D(L) \leftarrow$ ( newD ) ++ $D(L)$
16                 $t_D(L) \leftarrow$ ( timestamp($L$) ) ++ $t_D(L)$
17                 numAppends ← numAppends + 1
18             histogram($L$) ← newhist
19             timestamp($L$) ← $t'$
20         if numAppends > 0 then
21             if $|D(L)| > 20$ then Drop any oldest samples in $D(L)$ and $t_D(L)$
                on indices $i$ s.t. $t_D(L)[i] < t_{\mathcal{H}}$, but keeping $|D(L)| \geq 20$
22             if $|D(L)| = 1$ then $w \leftarrow 1$ else $w \leftarrow (0.10)^{\frac{1}{|D(L)|-1}}$
23             $\bar{D}(L) \leftarrow \left( \sum_{i=1}^{|D(L)|} w^{i-1}\, D(L)[i] \right) / \left( \sum_{i=1}^{|D(L)|} w^{i-1} \right)$
24 $L_s \leftarrow \phi$
25 if $\exists$ smallest $L \in L_s^{set}$ s.t. Stability($\bar{D}(L)$) = stable then
26     $L_s \leftarrow L$
27 $t_{\text{bcp}} \leftarrow \phi$
28 if $L_s \neq \phi \ \wedge \ \exists$ least $i \in [1, |D(L_s)|]$ s.t. Stability($D(L_s)[i]$) =
   highlyunstable $\wedge\ t_D(L_s)[i] \geq t_{\mathcal{H}} \ \wedge\ t_D(L_s)[i] > t_{\text{bcp}}^{prev}$ then
29     $t_{\text{bcp}} \leftarrow t_D(L_s)[i]$; $t_{\text{bcp}}^{\text{prev}} \leftarrow t_{\text{bcp}}$
30 return $\bar{D}$, $L_s$, $t_{\text{bcp}}$
```

Function Stability(Dmetric)

```
1 if Dmetric = $\phi$ then return unknown
2 if Dmetric $\leq 0.10$ then return stable
3 if Dmetric $\leq 0.50$ then return unstable
4 return highlyunstable
```

Function InterpolatedStability(L, $\bar{D}$, L_s^{set})

```
1 $L' \leftarrow L$ rounded to closest multiple of $W$
2 if $L' \in L_s^{set} \wedge$ Stability($\bar{D}(L')$) = stable then
3     return stable
4 $L_1 \leftarrow \phi$
5 for $L_2 \in L_s^{set}$ in ascending order do
6     if $L_1 \neq \phi \wedge L_1 < L' < L_2 \wedge$ Stability($\bar{D}(L_1)$) = Stability($\bar{D}(L_2)$) then
7         return Stability($\bar{D}(L_1)$)
8     if $L_2 \neq L'$ then
9         $L_1 \leftarrow L_2$
10 return unknown
```

Function AddStabilityCandidate(L, L_s^{set})

```
1 $L' \leftarrow L$ rounded to closest multiple of $W$
2 if $L' \notin L_s^{set}$ then
3     $L_s^{\text{set}} \leftarrow L_s^{\text{set}} \cup \{L'\}$
4 return $L_s^{set}$
```

```
Function ConstructHistogram($\mathcal{T}_e$, $t'$, $L$, resetBins, $e_{min}$, $e_{max}$)
    Data:
     range$_{min}(L)$, range$_{max}(L)$  :  The min-max bounds registered for length $L$
    Initially: range$_{min}$, range$_{max}$ $\leftarrow \phi, \phi$
1   if range$_{min}(L) = \phi \lor$ resetBins = true then
2       range$_{min}(L) \leftarrow e_{min}$
3       range$_{max}(L) \leftarrow e_{max}$
4   numBins $\leftarrow 10$
5   bins[1 .. numBins] $\leftarrow$ [0 .. 0]
6   for $t'' \in [t'-L+1, t']$ do
7       Get $e_{t''}$ from $\mathcal{T}_e$
8       bin $\leftarrow \left\lfloor \frac{e_{t''} - \text{range}_{min}(L)}{\text{range}_{max}(L) - \text{range}_{min}(L)} * \text{numBins} \right\rfloor + 1$
9       if bin < 1 then  bin $\leftarrow 1$
10      if bin > numBins then  bin $\leftarrow$ numBins
11      bins[bin] $\leftarrow$ bins[bin] + 1
12  return bins
```

```
Function KolmogorovSmirnovD(bins$_1$, bins$_2$, $L$)
    Input: bins$_1$[1 .. numBins], bins$_2$[1 .. numBins] :  histograms representing pdfs
1   maxdiff, cdf$_1$, cdf$_2$ $\leftarrow 0, 0, 0$
2   for (bin $\leftarrow 1$; bin $\leq$ numBins; bin++) do
3       cdf$_1 \leftarrow$ cdf$_1$ + bins$_1$[bin]
4       cdf$_2 \leftarrow$ cdf$_2$ + bins$_2$[bin]
5       maxdiff $\leftarrow$ max{maxdiff, |cdf$_1$ − cdf$_2$|}
6   Dmetric $\leftarrow \frac{\text{maxdiff}}{L}$
7   return Dmetric
```

Function UpdateCoverageLength(prevFrameEndedPFE, κ, $|\mathcal{H}|$, t_{bcp})

```
Data:  $L_c^{\mathrm{seq}}$ : A sorted sequence of tuples of coverage-lengths and timestamps,
                              kept sorted by coverage-length
       $t$                : Current timestep
       $L_c$              : Last updated coverage-length recommendation
Initially: $L_c^{\mathrm{seq}} \leftarrow \{(W,\, t)\}$ , $L_c \leftarrow W$
1  if $t_{\mathrm{bcp}} \neq \phi$ then
2      Drop samples from $L_c^{\mathrm{seq}}$ with timestamps $\leq t_{\mathrm{bcp}}$
3      if $|L_c^{seq}| = 0$ then $L_c^{\mathrm{seq}} \leftarrow \{(L_c,\, t)\}$
4  if prevFrameEndedPFE = true then
5      if $\kappa < 1$ then
6          $L \leftarrow |\mathcal{H}| + (1-\kappa) \times 2dim(\vec{x})$
7      else
8          $L \leftarrow \mathtt{max}\{|\mathcal{H}| - 2, W\}$
9      Add sample $(L, t)$ to $L_c^{\mathrm{seq}}$
10 Drop samples with oldest timestamps until $|L_c^{\mathrm{seq}}| \leq 10$
11 if prevFrameEndedPFE = true $\vee$ $t_{\mathrm{bcp}} \neq \phi$ then
12     $L_c \leftarrow$ Average of the 50% to 75% median coverage lengths in $L_c^{\mathrm{seq}}$
13 return $L_c$
```

Function AddSample($\vec{x}, \vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}, k, \gamma$)

```
Input: $\vec{x}$                                                  : most recent sample to be added to some history $h$
       $\vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}$ : statistics of history $h$ prior to adding sample
       $k$                                                        : length of history $h$ prior to adding sample
       $\gamma$                                                   : forget rate for past samples in history
Result: $\vec{\mu}^{\,\mathrm{new}}, \vec{\sigma}^{\,\mathrm{new}}, \vec{\chi}^{\,\mathrm{new}}, \vec{v}_{min}^{\,\mathrm{new}}, \vec{v}_{max}^{\,\mathrm{new}}$ : updated statistics after adding sample
1  if $k = 0$ then
2      $\vec{\mu}^{\,\mathrm{new}}, \vec{\sigma}^{\,\mathrm{new}}, \vec{\chi}^{\,\mathrm{new}}, \vec{v}_{min}^{\,\mathrm{new}}, \vec{v}_{max}^{\,\mathrm{new}} \leftarrow \vec{x}, 0, 0, \vec{x}, \vec{x}$
3      return $\vec{\mu}^{\,new}, \vec{\sigma}^{\,new}, \vec{\chi}^{\,new}, \vec{v}_{min}^{\,new}, \vec{v}_{max}^{\,new}$
4  $\vec{\mu}^{\,\mathrm{new}} \leftarrow \dfrac{(1-\gamma)\vec{x} + \gamma(1-\gamma^k)\vec{\mu}}{1-\gamma^{k+1}}$
5  $\vec{\sigma^2}^{\,\mathrm{new}} \leftarrow \dfrac{1-\gamma}{1-\gamma^{k+1}} \left( \gamma \dfrac{1-\gamma^k}{1-\gamma} \vec{\sigma}^2 + (\vec{x} - \vec{\mu}) \circ (\vec{x} - \vec{\mu}^{\,\mathrm{new}}) \right)$
6  $\vec{\chi}^{\,\mathrm{new}} \leftarrow \mathtt{max}\,(\gamma\, \vec{\chi},\, |\vec{x} - \gamma\, \vec{v}_{min}|\,,\, |\vec{x} - \gamma\, \vec{v}_{max}|)$
7  $\vec{v}_{min}^{\,\mathrm{new}} \leftarrow \mathtt{min}\,(\vec{x},\, \gamma\, \vec{v}_{min})$
8  $\vec{v}_{max}^{\,\mathrm{new}} \leftarrow \mathtt{max}\,(\vec{x},\, \gamma\, \vec{v}_{max})$
9  return $\vec{\mu}^{\,new}, \vec{\sigma}^{\,new}, \vec{\chi}^{\,new}, \vec{v}_{min}^{\,new}, \vec{v}_{max}^{\,new}$
```

Function RemoveSample($h, \vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}, k, \gamma$)

```
Input:
  $h$                                                      : history whose oldest sample ($k^{\text{th}}$ sample) is to be dropped
  $\vec{\mu}, \vec{\sigma}, \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}$ : statistics of history $h$ prior to dropping oldest sample
  $k$                                                      : length of history $h$ prior to dropping oldest sample
  $\gamma$                                                 : forget rate for past samples in history
Result:
  $\vec{\mu}^{\text{ new}}, \vec{\sigma}^{\text{ new}}, \vec{\chi}^{\text{ new}}, \vec{v}_{min}^{\text{ new}}, \vec{v}_{max}^{\text{ new}}$ : updated statistics after dropping oldest sample
1  if $k > 1$ then
2      $\vec{x} \leftarrow h[k]$
3      $\vec{\mu}^{\text{ new}} \leftarrow \dfrac{(1-\gamma^k)\vec{\mu} - (1-\gamma)\gamma^{k-1}\vec{x}}{1-\gamma^{k-1}}$
4      $\vec{\sigma^2}^{\text{ new}} \leftarrow \dfrac{1-\gamma}{1-\gamma^{k-1}} \left( \dfrac{1-\gamma^k}{1-\gamma} \left\{ \vec{\sigma}^2 + (\vec{\mu} - \vec{\mu}^{\text{ new}})^2 \right\} - \gamma^{k-1}(\vec{x} - \vec{\mu}^{\text{ new}})^2 \right)$
5      $\vec{\chi}^{\text{ new}}, \vec{v}_{min}^{\text{ new}}, \vec{v}_{max}^{\text{ new}} \leftarrow \vec{\chi}, \vec{v}_{min}, \vec{v}_{max}$
6      for $1 \le j \le n$ do
7          if $v_{min_j} \approx \gamma^{k-1} x_j$ then
8              $v_{min_j}^{\text{new}} \leftarrow \min_{1 \le u < k} \gamma^{u-1} h[u]$
9      for $1 \le j \le n$ do
10         if $v_{max_j} \approx \gamma^{k-1} x_j$ then
11             $v_{max_j}^{\text{new}} \leftarrow \max_{1 \le u < k} \gamma^{u-1} h[u]$
12     for $1 \le j \le n$ do
13         if $\chi_j \approx \max\left( \left| v_{min_j}^{new} - \gamma^{k-1} x_j \right|, \left| v_{max_j}^{new} - \gamma^{k-1} x_j \right| \right)$ then
14             $\chi_j^{\text{new}} \leftarrow \max_{1 \le u < v \le k} \left| \gamma^{u-1} h[u] - \gamma^{v-1} h[v] \right|$
15 else
16     $\vec{\mu}^{\text{ new}}, \vec{\sigma}^{\text{ new}}, \vec{\chi}^{\text{ new}}, \vec{v}_{min}^{\text{ new}}, \vec{v}_{max}^{\text{ new}} \leftarrow 0, 0, 0, 0, 0$
17 return $\vec{\mu}^{\ new}, \vec{\sigma}^{\ new}, \vec{\chi}^{\ new}, \vec{v}_{min}^{\ new}, \vec{v}_{max}^{\ new}$
```

Function BudgetAllocate(b, prevDuration)

```
Input:
    $b$           : per-frame budget allocated by programmer to execute $\mathcal{R}$
    prevDuration  : time take by previous invocation of FrameTransition
Data: debt : accumulated budget deficit over all previous frames
1 debt $\leftarrow$ debt + (prevDuration $-$ $b$)
2 $b_{\text{usable}} \leftarrow$ min($b$, $b$ $-$ debt)
3 return $b_{\text{usable}}$
```

Function InputExplorer(coverageMet, $\vec{\mu}$, $\vec{\sigma}$, $\vec{\chi}$, $\vec{x}_{t-1}$, $\text{len}^{\mathcal{H}}$)

Input:	$\vec{\mu}$, $\vec{\sigma}$	:	weighted mean and std of $\vec{x}$ samples in $\mathcal{H}$
	$\vec{x}_{t-1}$	:	the control input applied the previous frame
	$\text{len}^{\mathcal{H}}$	:	length of history $\mathcal{H}$
Result:	$\vec{x}\,'$	:	input to be applied next

```
1  if $\text{len}^{\mathcal{H}} = 0$ then
2      $\vec{x}\,' \leftarrow [0, 0, \ldots, 0]^{\mathrm{T}}$
3  else if coverageMet = true then
4      $\vec{x}\,' \leftarrow \vec{x}_{t-1}$
5      $j \leftarrow$ random(1, $dim(\vec{x})$)
6      $x'_j \leftarrow$ random($-N_j$, $N_j$)
7  else
8      for $j \in [1, dim(\vec{x})]$ do
9          if $\sigma_j \geq 1.20 \times \frac{1}{2} N_j \ \wedge \ \chi_j \geq 1.20 \times \frac{1+\gamma}{2} N_j$ then
10             $x'_j \leftarrow x_{j|t-1}$
11         else
12             if $\mu_j \approx 0$ then
13                 $x'_j \leftarrow$ random($-N_j$, $N_j$)
14             if $\mu_j > 0$ then
15                 $x'_j \leftarrow$ random($-N_j$, $\mu_j$)
16             else
17                 $x'_j \leftarrow$ random($\mu_j$, $N_j$)
18 return $\vec{x}\,'$
```

APPENDIX C

ZERO-ORDER SYSTEMS

For some applications, the functional design of the application may make it simple for the programmer to infer that the execution properties of each frame are completely (or pre-dominantly) independent of the prior frames. For such applications, choosing a model-order = 0 (i.e., $x_{order} = 0, y_{order} = 0$) would be the most accurate characterization of the application. While the LQR scheme is essentially a highly optimized special-case of an LLSE problem, the LQR case is intended for feedback-control and does not address the zero-order model situation. However, the zero-order model can be efficiently inverted directly using LLSE (since the complexity of the control problem grows significantly with model-order and control horizon N).

The zero-order model $\mathcal{M}$ is determined by estimating matrix L in

$$\vec{\tilde{y}}_{t_0} = L\,\vec{x}_{t_0}, \tag{108}$$

a special-case of Eq 26.

At time-step t, we want to solve for $\Delta\vec{x}_t$ in

$$\vec{\beta} \circ \epsilon\vec{y}_{t-1} = L\,\Delta\vec{x}_t, \tag{109}$$

while minimizing fit-error $||\vec{\beta} \circ \epsilon\vec{y}_{t-1} - L\,\Delta\vec{x}_t||^2$.

Here $\epsilon\vec{y}_{t-1} = \vec{y}_{t-1} - \vec{\tilde{y}}$ is the observed tracking-error for the previous time-step. Rather than simply inverting $\vec{\tilde{y}} = L\,\vec{x}_t$ in order to determine the best input $\vec{x}_t$ to produce objective $\vec{\tilde{y}}$ at time-step t, we want to incorporate feedback control to compensate for model mismatch/drift between $\mathcal{M}$ and the application. We arrive at Eq 109 as follows:

- Use the *difference form* of Eq 108:

$$\Delta\vec{y}_t = L\,\Delta\vec{x}_t, \tag{110}$$

to incorporate feedback of observations.

- Use the adaptive strategy elaborated in Section 6.5.2 to dynamically adjust feedback sensitivity to deviations from the objective $\vec{\tilde{y}}$. Use of the sensitivity gain $\vec{\beta}$ requires the monotonicity property (Kumar et al. [31]), which is also justified by Section 6.5.2.

Note that the difference-form of the model in Eq 110 can stabilize on an input when $\Delta\vec{y}_{t-1} \to 0$, without requiring that $\vec{y}_{t-1} \to \vec{\tilde{y}}$. However, the adaptive tracking-error form in Eq 109 will continue to adjust $\vec{x}_t$ until the tracking error $\epsilon\vec{y}_{t-1} \to 0$, even in the presence of model-drift from $\mathcal{M}$, and varying sensitivities of $\vec{y}_t$ to $\vec{x}_t$ over their possibly non-linear response curve (rather, higher-dimensional response surface).

We can compute the least-squares-error input-increment in Eq 109 as follows: $\Delta\vec{x}_t \leftarrow L^{\dagger}\,(\vec{\beta} \circ \epsilon\vec{y}_{t-1})$. However, L may not be full-rank, and therefore, may not have a pseudo-inverse $L^{\dagger}$. As discussed in Section 5.7.2, this limitation is overcome by *regularization*, yielding the following procedure:

$$\Delta\vec{x}_t \leftarrow \begin{bmatrix} L \\ \sqrt{\delta}\,I \end{bmatrix}^{\dagger} \begin{bmatrix} \vec{\beta} \circ \epsilon\vec{y}_{t-1} \\ 0 \end{bmatrix}. \tag{111}$$

Typically, the pseudo-inverse is not explicitly computed, as that requires the inversion and multiplication of very large matrices. Instead, the LLSE procedure is used for efficiency. However, here we are dealing with a small $m \times n$ matrix L, and the pseudo-inverse of its regularized form $L' = \begin{bmatrix} L \\ \sqrt{\delta}\,I \end{bmatrix}$ is quite efficient to compute explicitly: $L'^{\dagger} \leftarrow (L'^{T}L')^{-1}L'^{T}$.

When a zero-order model-structure is required (i.e., $x_{order} = y_{order} = 0$), the previously described algorithms need to be modified as follows:

- The following UpdateRegulatorZO() procedure replaces the previously described UpdateRegulator() procedure.

- The state representation needs to be modified to retain the previous frame's input and output (despite zero-order for both), as follows: $\vec{s}_t = \begin{bmatrix} \vec{y}_{t-1}^{\,\text{sc}} \\ \vec{u}_{t-1} \end{bmatrix}$.
- Further note that, as signified by the use of $\vec{u}_{t-1}$, the retained past input is *continuous* instead of the discretized $\vec{x}_{t-1}$ (but is still clipped to input-bounds).
- Correspondingly, the FrameTransition() procedure must be modified to call the AdaptiveStateTransition() procedure with $\vec{u}_{t-1}$ as parameter instead of $\vec{x}_{t-1}$.
- Therefore, the FrameTransition() procedure needs to retain $\vec{u}_{t-1}$ as past input in addition to $\vec{x}_{t-1}$.

```
Function UpdateRegulatorZO(deadline)
  Data:
      $\delta$            : regularization parameter to use for next LLSE for constructing $\mathcal{C}$
      $\eta_\delta, \rho_\delta$   : metrics produced by the last invocation of $\mathcal{C}$
  Initially: $\delta \leftarrow$ default, $\eta_\delta \leftarrow \phi$, $\rho_\delta \leftarrow \phi$
1 if $\mathcal{M} \neq \phi$ then
2     $L =$ matrix form of $\mathcal{M}$
3     if time() $<$ deadline $\wedge$ $(\mathcal{C} = \phi \vee \textit{unbalanced}(\eta_\delta, \delta\,\rho_\delta))$ then
4         if $\eta_\delta \neq \phi \wedge \eta_\delta > 0 \wedge \rho_\delta > 0$ then
5             $\delta \leftarrow \sqrt{\frac{\eta_\delta\,\delta}{\rho_\delta}}$
6         $L' \leftarrow \begin{bmatrix} L \\ \sqrt{\delta}\,I \end{bmatrix}$
7         $L_{inv} \leftarrow (L'^T L')^{-1} L'^T$
8         $\mathcal{C}(\vec{s}_{t_0}) \leftarrow Function\{$
                  $\Delta\vec{u}_{t_0} \leftarrow L_{inv}\,(\vec{y}^{\,\text{sc}}_{t_0-1} - \vec{\tilde{y}})$,
                        where $\vec{s}_{t_0} = \begin{bmatrix} \vec{y}^{\,\text{sc}}_{t_0-1} \\ \vec{u}_{t_0-1} \end{bmatrix}$, $\vec{y}^{\,\text{sc}}_{t_0-1} = \vec{\tilde{y}} + \vec{\beta} \circ \vec{y}_{t_0-1}$
                  $\eta_\delta \leftarrow ||L\,\Delta\vec{u}_{t_0} - (\vec{y}^{\,\text{sc}}_{t_0-1} - \vec{\tilde{y}})||^2$
                  $\rho_\delta \leftarrow ||\Delta\vec{u}_{t_0}||^2$
                  $\vec{u}_{t_0} \leftarrow \vec{u}_{t_0-1} + \Delta\vec{u}_{t_0}$
                  $\vec{u}_{t_0} \leftarrow$ ClipInputBoundViolations($\vec{u}_{t_0}$)
                  return $\vec{u}_{t_0}$
              }
```

APPENDIX D

MERGING VCG PATTERNS

Section 9.6 covers the overall criteria and algorithms for the merger of patterns. Here we expand on some implementation-level details of the merge algorithms and capture some basic math. Reading this appendix is not needed for gaining a full understanding of the techniques introduced in Chapter 9.

A pattern P_i refers to the *root node* of the pattern, from which the rest of the pattern can be traversed. Two patterns P_1 and P_2 are merged using a three pass algorithm. The **first pass**, shown as recursive Function MergePatternTrees (Algorithm 1), produces a *merge-candidate* VCG tree structure P_M, assuming no incompatibility is found between the tree structures of P_1 and P_2. Each node in P_M has a pointer to the corresponding node in P_1 and/or P_2. The corresponding node may possibly be absent in at most one of P_1 or P_2. Therefore, the tree structure produced in P_M is an *overlap* of the tree structures of P_1 and P_2.

The first stage of Function MergePatternTrees (Algorithm 1), lines 1 - 2, checks if P_1 and P_2 match on the name and types of their root nodes. The second stage, lines 4 - 23, performs a **call-context compatibility check** over the call-contexts from nodes P_1 and P_2 to their children. Note that this is the mechanism that determines which child of P_1 corresponds to which child of P_2 (if any). The orderings $P_1.children[0..m_1]$ and $P_2.children[0..m_2]$ *do not establish a correspondence* between the children of P_1 and P_2. The call-context compatibility check verifies that if a newly created child node *merge_child* of P_M has a call-context cc (i.e., $cc \in merge\_child.ccs$), and cc is in the call-context of some child, *child*, of P_1, then this must imply that *i) child* under P_1 becomes the corresponding node for *merge_child* under P_M, and *ii)* every other call-context $cc_1 \in child.ccs$ must necessarily also be mapped to *merge_child* (and similarly for an appropriate child of P_2). The second requirement may fail to be satisfied if some $cc_1 \in child.ccs$ has already been assigned to a different merge-child

Algorithm 1: MergePatternTrees(P_1, P_2)

```
1  if $P_1 \neq \bot \wedge P_2 \neq \bot$ then
2      if $P_1.fname \neq P_2.fname$
       $\vee\ P_1.node\_type \neq P_2.node\_type$ then
       return $\bot$
3  end
4  $all\_P_1\_ccs \leftarrow \bigcup_i P_1.children[i].ccs$ if
   $P_1 \neq \bot$, else $\phi$
5  $all\_P_2\_ccs \leftarrow \bigcup_i P_2.children[i].ccs$ if
   $P_2 \neq \bot$, else $\phi$
6  $all\_ccs \leftarrow all\_P_1\_ccs \cup all\_P_2\_ccs$
7  Initialize $Assigned[cc] \leftarrow \bot,\ \forall\, cc \in all\_ccs$
8  $mcount \leftarrow 0$
9  foreach $cc \in all\_ccs$ do
10     if $Assigned[cc] = \bot$ then
11         $Assigned[cc] \leftarrow mcount$
12         $mcount \leftarrow mcount + 1$
13         if $P_1 \neq \bot \wedge \exists\, i$ s.t. $cc \in$
           $P_1.children[i].ccs$ then
14             if $\exists\, cc_1 \in P_1.children[i].ccs$ s.t.
               $Assigned[cc_1] \neq \bot\ \wedge$
               $Assigned[cc_1] \neq Assigned[cc]$
               then
15                 return $\bot$
16             end
17             $Assigned[cc_1] \leftarrow Assigned[cc]$,
               $\forall\, cc_1 \in P_1.children[i].ccs$
18         end
19         if $P_2 \neq \bot \wedge \exists\, i$ s.t. $cc \in$
           $P_2.children[i].ccs$ then
20             if $\exists\, cc_2 \in P_2.children[i].ccs$ s.t.
               $Assigned[cc_2] \neq \bot\ \wedge$
               $Assigned[cc_2] \neq Assigned[cc]$
               then
21                 return $\bot$
22             end
23             $Assigned[cc_2] \leftarrow Assigned[cc]$,
               $\forall\, cc_2 \in P_2.children[i].ccs$
24         end
25     end
26 end
27
28 $P_M \leftarrow$ Create VCG node with the
   matching $fname$ and $node\_type$
   attributes from $P_1$ and/or $P_2$
   (whichever $\neq \bot$).
29
30 $P_M.CorrNodes \leftarrow [P_1,\ P_2]$
31
32 for $0 \leq m < mcount$ do
33     $combined\_child\_ccs \leftarrow$
       $\{cc : Assigned[cc] = m\}$
34     $child\_P_1 \leftarrow \bot$
35
36     if $P_1 \neq \bot \wedge \exists\, i$ s.t. $Assigned[cc] = m$
       where $cc \in P_1.children[i].ccs$ then
37         $child\_P_1 \leftarrow P_1.children[i]$
38     end
39
40     $child\_P_2 \leftarrow \bot$
41     if $P_2 \neq \bot \wedge \exists\, i$ s.t. $Assigned[cc] = m$
       where $cc \in P_2.children[i].ccs$ then
42         $child\_P_2 \leftarrow P_2.children[i]$
43     end
44     $merge\_child \leftarrow$
       MergePatternTrees($child\_P_1$, $child\_P_2$)
45     if $merge\_child = \bot$ then
46         return $\bot$
47     end
48
49     $merge\_child.ccs \leftarrow combined\_child\_ccs$
50     $merge\_child.parent \leftarrow P_M$
51     $P_M.children[m] \leftarrow merge\_child$
52 end
53 return $P_M$
```

under P_M. Violation of the call-context compatibility check will cause the merge process to fail. The intent here is that the call-contexts of *child* must not get split over multiple children of P_M. Therefore, *child.ccs* $\subseteq$ *merge_child.ccs*, as *merge_child.ccs* is being created

Algorithm 2: MergeContributionStructure(P_M)

```
   input  : VCG pattern $P_M$, with merge-nodes pointing to corresponding nodes in $P_1$
            and $P_2$
   output: $P_M$ with contribution structure merged if compatible, else $\perp$
1  foreach $mnode$ in pre-order traversal of $P_M$ tree do
2      if $\exists\, cnode_i, cnode_j \in mnode.CorrNodes,\ cnode_i \neq cnode_j \wedge cnode_i, cnode_j \neq \perp$ s.t. $\exists\, LS_i \in cnode_i.list\_ols$ with no corresponding $LS_j \in cnode_j.list\_ols$, and, $cnode_j$ has a child node corresponding to the second node in the $LS_i$ sequence
       then return $\perp$
3      if $\exists\, cnode_i, cnode_j \in mnode.CorrNodes,\ cnode_i \neq cnode_j \wedge cnode_i, cnode_j \neq \perp$ s.t. $cnode_i.cls \neq \perp$ then
4          if $cnode_j.cls = \perp \vee$ Node sequences $cnode_i.cls$ and $cnode_j.cls$
           are not in correspondence via $P_M$ then return $\perp$
5      end
6      foreach $cnode_i \in mnode.CorrNodes$ do
7          foreach $LS_i \in cnode.list\_ols$ do
8              $merged\_LS \leftarrow$ Construct a sequence of nodes in $P_M$ that corresponds to
               $LS_i$
9              if $merged\_LS \notin mnode.list\_ols$ then
10                 Append $merged\_LS$ to $mnode.list\_ols$
11                 foreach $mn$ in $merged\_LS$ node sequence except first node (i.e.,
                   $mnode$) do $mn.cls \leftarrow merged\_LS$
12             end
13         end
14     end
15 end
16 return $P_M$
```

to accommodate the call-chains of corresponding nodes $child_1$ and $child_2$ from P_1 and P_2, and $child_1.ccs$ may not necessarily be identical to $child_2.ccs$ (say, due to internal merging of subtrees within a pattern). The last stage, lines 28 - 51, *recursively* invokes Function MergePatternTrees on pairs of children drawn over P_1 and P_2 that have been found to be in correspondence (based on call-contexts). This creates *mcount* merged children subtrees under P_M.

The **second pass**, shown as Function MergeContributionStructure (Algorithm 2), takes the merged tree structure P_M (with its intrinsic pointers to corresponding nodes within P_1 and P_2) and returns it with the corresponding contribution structures also merged, provided no incompatibility exists between the contribution structures of P_1 and P_2.

Line 2 is the **originating-linear-segment compatibility check**. Compatibility is violated if a node $cnode_i$ in, say P_1, is originating some linear segment, LS, which does not have a corresponding linear segment originating at $cnode_j$, the node in P_2 that corresponds to $cnode_i$ in P_1 via P_M. However, if $cnode_j$ is missing the child subtree that could have carried the linear segment corresponding to LS then no conflict in interpretation is introduced by allowing the contribution structure to merge. Lines 5 - 4 are the **linear-segment internal-compatibility check**. This check verifies if a node $cnode_i$, say in P_1, is internal to a linear segment LS, and the corresponding node $cnode_j$ in P_2 is internal to a linear segment *identical* to LS. However, if corresponding node $cnode_j$ does not exist (i.e., $cnode_j = \perp$), then no conflict in interpretation of the merged structure is introduced at *mnode*.

The **third pass** updates the metrics on the merged nodes as described in Section D.1. Then it computes the **Kolmogorov-Smirnov difference measure** $0.0 \leq D \leq 1.0$ between the original metrics m_1 and m_2 coming from P_1 and P_2, respectively. We treat the m_1 and m_2 metrics as Gaussians for the purpose of computing D. If the node or edge corresponding to, say the m_2 metric, does not exist in P_2 (since we allow the merging of dissimilar patterns), an appropriately zeroed out value is used for m_2 in computing D. This would make a merger of dissimilar structures automatically more expensive since a zero-distribution will have a relatively high D difference from non-trivial distributions. The algorithm then computes a weighted average of the D values over the merged tree. The W_i measures (total execution time) of merged tree nodes are used to weigh the associated D in the average. Since we want a strong dissimilarity at the root node ($D > \tau_S$) to *by itself* rule the merger to be prohibitively costly, we choose to use a breadth-first-traversal to *progressively* add nodes (and corresponding D terms) to the weighted average. If the progressive average exceeds the summarization pressure τ_S at any point, we prohibit the merger. Otherwise, a scalar merge-cost d_{P_1,P_2} is produced. Therefore, the programmer can choose τ_S in the range 0.0 to 1.0, where a low τ_S only allows highly similar structures and behaviors to merge thereby providing *precision*, whereas a larger τ_S would *force the summarization* of the results into much fewer, more compressed patterns that possibly *generalize better* to future runs of the application.

If a merge-cost $d_{P1,P2}$ is produced, then the tuple $((P_1, P_2), d_{P1,P2}, P_M)$ provides a *pair of points* for the agglomerative-clustering algorithm to consider. Agglomerative clustering uses $d_{P1,P2}$ as the corresponding distance measure, and P_M as the result of combining P_1 and P_2.

D.1 Merging Statistics

Consider corresponding nodes F_1 and F_2 from P_1 and P_2 respectively, that get merged into F_M under merged pattern tree P_M. The root nodes P_1 and P_2 (since a pattern is represented by the root to the pattern tree) have had their call-contexts from `main` combined under P_M, i.e., $P_M.ccs = P_1.ccs \cup P_2.ccs$.

Under P_i there is a path of VCG nodes to F_i, with call-context-sets at each level on this path. Now, $F_1.fname = F_2.fname = F_M.fname$ since the merger was successful. Let us refer to that common function name as $fname$ for brevity here. Therefore, every *full call-chain* (all the way from `main`) that results in those invocations of $fname$ that got counted in N^{F_1}, can be *enumerated* as some concatenation of the call-chain-segments taken from the call-context-sets in P_1 along the VCG path to F_1. Note that the converse is not necessarily true: every full call-chain that can possibly be constructed by taking segments from the call-context-sets in a VCG path, may not have actually occurred during the profiling execution of the application. But we don't need the converse property to hold.

Now, once the patterns P_1 and P_2 are merged, we would like the same property to hold for F_M under P_M. Clearly, this **enumerable-path property** still holds as the call-context-set at each merged node is a superset of the corresponding nodes in P_1 and P_2. With regards to annotated statistics, we still need to combine the N^{F_1} invocations of $fname$ for F_1 and the N^{F_2} invocations for F_2 into N^{F_M} invocations for F_M. The programmer should be able to interpret the statistics uniformly for P_1, P_2 and P_M without needing to know if some of these patterns are the result of merging one or more layers of lower-level patterns. This can indeed be done as follows:

$$\begin{aligned}
N^{F_M} &\leftarrow N^{F_1} + N^{F_2}, \\
\bar{X}^{F_M} &\leftarrow \frac{\bar{X}^{F_1} * N^{F_1} + \bar{X}^{F_2} * N^{F_2}}{N^{F_M}}, \\
\sigma^2_{X^{F_M}} &\leftarrow \frac{N^{F_1} * (\sigma^2_{X^{F_1}} + (\bar{X}^{F_1} - \bar{X}^{F_M})^2) + N^{F_2} * (\sigma^2_{X^{F_2}} + (\bar{X}^{F_2} - \bar{X}^{F_M})^2)}{N^{F_M}}.
\end{aligned}$$

These can be verified as being *mathematical equivalences*. That is, even though we no longer have available the N^{F_1} separate observations of r.v. X^{F_1} and the N^{F_2} separate observations of r.v. X^{F_2}, we can still exactly compute the resulting metrics as if we had re-computed the mean and variance directly from the combined observations. The secondary metrics, CoV^{F_M} and W^{F_M}, can always be recomputed from the primary metrics of mean, variance and count.

For a Task node H with incoming back-edge b_i corresponding to the i^{th} linear segment originating at H, i.e., $H.list\_ols[i]$, the *variance-contribution* statistics are recorded as a tuple in $H.ols\_stats[i] = T_i$. Let descendant W of H be the source of b_i. Then, $T_i = (\bar{Y}^{H|W}, \sigma^2_{Y^{H|W}}, N^H)$ is the statistics tuple for b_i. T_i that can be updated during mergers of structurally-corresponding back-edges b_i from P_1 and P_2 just like the merger of node statistics above. The *contribution fraction* along b_i is a secondary metric $cf_i \leftarrow \mathbf{C}^{H|W}_{W,W}/\sigma^2_H$. The numerator term is updated on merger like the variance terms above.

REFERENCES

[1] R. Cledat, T. Kumar, J. Sreeram, and S. Pande, "Opportunistic computing: a new paradigm for scalable realism on many-cores," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, (Berkeley, CA, USA), pp. 5–5, USENIX Association, 2009.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "Lustre: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, (New York, NY, USA), pp. 178–188, ACM, 1987.

[3] C. André and F. Mallet, "Specification and verification of time requirements with ccsl and esterel," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '09, (New York, NY, USA), pp. 167–176, ACM, 2009.

[4] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction* (R. Horspool, ed.), vol. 2304 of *Lecture Notes in Computer Science*, pp. 179–196, Springer Berlin Heidelberg, 2002.

[5] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "QoS control strategies for high-quality video processing," *Real-Time Syst.*, vol. 30, pp. 7–29, May 2005.

[6] B. C. Song and K. W. Chun, "A virtual frame rate control algorithm for efficient mpeg-2 video encoding," *Consumer Electronics, IEEE Transactions on*, vol. 49, pp. 460–465, May 2003.

[7] P. Tudor, "Mpeg-2 video compression," *Electronics Communication Engineering Journal*, vol. 7, pp. 257–264, Dec 1995.

[8] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, pp. 100–107, July 1968.

[9] J. Yao, C. Lin, X. Xie, A. Wang, and C.-C. Hung, "Path planning for virtual human motion using improved a* star algorithm," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pp. 1154–1158, April 2010.

[10] J. Arnold and M. Cavenor, "A practical course in digital video communications based on matlab," *Education, IEEE Transactions on*, vol. 39, pp. 127–136, May 1996.

[11] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Commun. ACM*, vol. 55, pp. 61–69, June 2012.

[12] G. Bradski, "Opencv: Examples of use and new applications in stereo, recognition and tracking," in *Proc. Intern. Conf. on Vision Interface (VI2002)*, p. 347, 2002.

[13] J. Gregory, *Game engine architecture*. CRC Press, 2009.

[14] A. Boeing and T. Bräunl, "Evaluation of real-time physics simulation systems," in *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, (New York, NY, USA), pp. 281–288, ACM, 2007.

[15] K. Ogata, *Modern Control Engineering (4th Edition)*. Prentice Hall, November 2001.

[16] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *trans. ASME*, vol. 64, no. 11, 1942.

[17] C. C. Hang, K. J. Åström, and W. K. Ho, "Refinements of the ziegler–nichols tuning formula," in *IEE Proceedings D (Control Theory and Applications)*, vol. 138, pp. 111–118, IET, 1991.

[18] K. Åström and T. Hägglund, "Revisiting the ziegler–nichols step response method for pid control," *Journal of process control*, vol. 14, no. 6, pp. 635–650, 2004.

[19] K. J. Astrom and B. Wittenmark, *Adaptive Control.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 1994.

[20] M. Krstic, "Lyapunov stability of linear predictor feedback for time-varying input delay," *Automatic Control, IEEE Transactions on*, vol. 55, no. 2, pp. 554–559, 2010.

[21] K. M. Passino, A. N. Michel, and P. J. Antsaklis, "Lyapunov stability of a class of discrete event systems," *Automatic Control, IEEE Transactions on*, vol. 39, no. 2, pp. 269–279, 1994.

[22] S. Park, J. Deyst, and J. P. How, "Performance and lyapunov stability of a nonlinear path following guidance method," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 6, pp. 1718–1728, 2007.

[23] K. Gu and S.-I. Niculescu, "Survey on recent results in the stability and control of time-delay systems*," *Journal of Dynamic Systems, Measurement, and Control*, vol. 125, no. 2, pp. 158–165, 2003.

[24] K. H. Ang, G. Chong, and Y. Li, "Pid control system analysis, design, and technology," *Control Systems Technology, IEEE Transactions on*, vol. 13, no. 4, pp. 559–576, 2005.

[25] R. Monopoli, "Model reference adaptive control with an augmented error signal," *Automatic Control, IEEE Transactions on*, vol. 19, pp. 474–484, Oct 1974.

[26] E. Lavretsky and N. Hovakimyan, "Positive /spl mu/-modification for stable adaptation in the presence of input constraints," in *American Control Conference, 2004. Proceedings of the 2004*, vol. 3, pp. 2545–2550 vol.3, June 2004.

[27] K. J. Åström, "Theory and applications of self-tuning regulators," in *Control Theory, Numerical Methods and Computer Systems Modelling*, pp. 669–680, Springer, 1975.

[28] L. Guo and H.-F. Chen, "The åström-wittenmark self-tuning regulator revisited and els-based adaptive trackers," *Automatic Control, IEEE Transactions on*, vol. 36, pp. 802–812, Jul 1991.

[29] K. Wong and M. Bayoumi, "Multivariable self-tuning regulators," in *Decision and Control including the Symposium on Adaptive Processes, 1981 20th IEEE Conference on*, pp. 978–983, Dec 1981.

[30] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao, "An adaptive control framework for QoS guarantees and its application to differentiated caching," in *IWQoS '02*, pp. 23 – 32.

[31] T. Kumar, R. E. Cledat, and S. Pande, "Dynamic tuning of feature set in highly variant interactive applications," in *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, (New York, NY, USA), pp. 289–298, ACM, 2010.

[32] J. Shamma and M. Athans, "Gain scheduling: potential hazards and possible remedies," *Control Systems, IEEE*, vol. 12, pp. 101–107, June 1992.

[33] J. Shamma and M. Athans, "Guaranteed properties for nonlinear gain scheduled control systems," in *Decision and Control, 1988., Proceedings of the 27th IEEE Conference on*, pp. 2202–2208 vol.3, Dec 1988.

[34] D. J. Leith and W. E. Leithead, "Survey of gain-scheduling analysis and design," *International journal of control*, vol. 73, no. 11, pp. 1001–1025, 2000.

[35] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*, pp. 48–70, Springer, 2009.

[36] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th international conference on Autonomic computing*, pp. 79–88, ACM, 2010.

[37] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ACM SIGPLAN Notices*, vol. 46, pp. 199–212, ACM, 2011.

[38] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 299–310, ACM, 2014.

[39] P. Mejía-Alvarez, R. Melhem, and D. Mossé, "An incremental approach to scheduling during overloads in real-time systems," in *Proceedings of the 21st IEEE conference on Real-time systems symposium*, RTSS'10, (Washington, DC, USA), pp. 283–293, IEEE Computer Society, 2000.

[40] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a pc game engine," *Computer Graphics and Applications, IEEE*, vol. 18, pp. 46–53, Jan 1998.

[41] J. Orkin, "Symbolic representation of game world state: Toward real-time planning in games," in *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, vol. 5, 2004.

[42] H. Lu, W. Yijin, and Y. Hu, "Design and implementation of three-dimensional game engine," in *World Automation Congress (WAC), 2012*, pp. 1–4, June 2012.

[43] B. Leibe, N. Cornelis, K. Cornelis, and L. Van Gool, "Dynamic 3d scene analysis from a moving vehicle," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pp. 1–8, June 2007.

[44] C. P. Marzario, T. Cucinotta, L. Palopoli, L. Marzario, and G. Lipari, "Adaptive reservations in a linux environment," in *RTAS '04*, pp. 238–245, 2004.

[45] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint, "An adaptive framework for multiprocessor real-time system," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, (Washington, DC, USA), pp. 23–33, IEEE Computer Society, 2008.

[46] D. d. Niz, K. Lakshmanan, and R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, (Washington, DC, USA), pp. 291–300, IEEE Computer Society, 2009.

[47] M. Roitzsch and M. Pohlack, "Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, (Washington, DC, USA), pp. 271–280, IEEE Computer Society, 2006.

[48] Y. Huang, A. V. Tran, and Y. Wang, "A workload prediction model for decoding mpeg video and its application to workload-scalable transcoding," in *Proceedings of the 15th international conference on Multimedia*, MULTIMEDIA '07, (New York, NY, USA), pp. 952–961, ACM, 2007.

[49] G. Sullivan and T. Wiegand, "Rate-distortion optimization for video compression," *Signal Processing Magazine, IEEE*, vol. 15, pp. 74–90, Nov 1998.

[50] E. hui Yang and X. Yu, "Rate distortion optimization for h.264 interframe coding: A general framework and algorithms," *Image Processing, IEEE Transactions on*, vol. 16, pp. 1774–1784, July 2007.

[51] T. Kailath, A. H. Sayed, and B. Hassibi, *Linear Estimation.* Prentice Hall, 1 ed., 4 2000.

[52] S. Haykin, *Neural Networks: A Comprehensive Foundation.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1994.

[53] J. Moody and C. J. Darken, "Fast learning in networks of locally-tuned processing units," *Neural Comput.*, vol. 1, pp. 281–294, June 1989.

[54] T. Poggio and F. Girosi, "Networks for approximation and learning," *Proceedings of the IEEE*, vol. 78, pp. 1481 –1497, Sept. 1990.

[55] T. M. Cover, "Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition," *Electronic Computers, IEEE Transactions on*, vol. EC-14, no. 3, pp. 326 –334, 1965.

[56] F. L. Lewis and V. L. Syrmos, *Optimal Control.* Wiley-Interscience, 2nd ed., 1995.

[57] J. Spooner and K. Passino, "Decentralized adaptive control of nonlinear systems using radial basis neural networks," *Automatic Control, IEEE Transactions on*, vol. 44, pp. 2050–2057, Nov 1999.

[58] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: less is more," *SIGPLAN Not.*, vol. 35, pp. 202–211, November 2000.

[59] R. J. Hall, "Call path profiling," in *Proceedings of the 14th international conference on Software engineering*, ICSE '92, (New York, NY, USA), pp. 296–306, ACM, 1992.

[60] M. Arnold, M. Hind, and B. G. Ryder, "Online feedback-directed optimization of Java," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, (New York, NY, USA), pp. 111–129, ACM, 2002.

[61] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 45–57, ACM, 2002.

[62] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (Washington, DC, USA), pp. 135–146, IEEE Computer Society, 2006.

[63] Y. Liu, A. Maxiaguine, S. Chakraborty, and W. T. Ooi, "Processor frequency selection for soc platforms for multimedia applications," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pp. 336–345, Dec 2004.

[64] A. Maxiaguine, Y. Zhu, S. Chakraborty, and W.-F. Wong, "Tuning soc platforms for multimedia processing: identifying limits and tradeoffs," in *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pp. 128–133, Sept 2004.

[65] Y. Liu, S. Chakraborty, and W. T. Ooi, "Approximate VCCs: a new characterization of multimedia workloads for system-level MpSoC design," in *Proceedings of the 42nd annual Design Automation Conference*, DAC '05, (New York, NY, USA), pp. 248–253, ACM, 2005.

[66] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack reclaiming," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, (Washington, DC, USA), pp. 410–421, IEEE Computer Society, 2005.

[67] E. Wandele and L. Thiele, "Abstracting functionality for modular performance analysis of hard real-time systems," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, (New York, NY, USA), pp. 697–702, ACM, 2005.

[68] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, (Washington, DC, USA), pp. 57–66, IEEE Computer Society, 2006.

[69] "AbsInt Angewandte Informatik GmbH." `http://www.absint.com`.

[70] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf, "MediaBench II video: Expediting the next generation of video systems research," *Microprocess. Microsyst.*, vol. 33, pp. 301–318, June 2009.

[71] "Torque game engine." `http://www.torquepowered.com`.

[72] P. C. Hansen and D. P. O'Leary, "The use of the l-curve in the regularization of discrete ill-posed problems," *SIAM Journal on Scientific Computing*, vol. 14, no. 6, pp. 1487–1503, 1993.

[73] D. Calvetti, S. Morigi, L. Reichel, and F. Sgallari, "Tikhonov regularization and the l-curve for large discrete ill-posed problems," *Journal of computational and applied mathematics*, vol. 123, no. 1, pp. 423–446, 2000.

[74] F. S. V. Bazán, "Fixed-point iterations in determining the tikhonov regularization parameter," *Inverse Problems*, vol. 24, no. 3, p. 035001, 2008.

[75] M. Ozuysal, M. Calonder, V. Lepetit, and P. Fua, "Fast keypoint recognition using random ferns," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 32, pp. 448–461, March 2010.

[76] "Real-time face tracking and recognition." `http://rtftr.sourceforge.net`.

[77] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[78] H. Schwarz, D. Marpe, and T. Wiegand, "Overview of the scalable video coding extension of the h.264/avc standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, pp. 1103–1120, Sept 2007.

[79] J. Deutscher, A. Blake, and I. Reid, "Articulated body motion capture by annealed particle filtering," in *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, vol. 2, pp. 126–133 vol.2, 2000.

[80] J. Schmidt, J. Fritsch, and B. Kwolek, "Kernel particle filter for real-time 3d body tracking in monocular color images," in *Automatic Face and Gesture Recognition, 2006. FGR 2006. 7th International Conference on*, pp. 567–572, April 2006.

[81] T. Kumar, J. Sreeram, R. Cledat, and S. Pande, "A profile-driven statistical analysis framework for the design optimization of soft real-time applications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, (New York, NY, USA), pp. 529–532, ACM, 2007.

[82] T. Kumar, R. Cledat, J. Sreeram, and S. Pande, "Statistically analyzing execution variance for soft real-time applications," in *Languages and Compilers for Parallel Computing (LCPC'08)*, pp. 124–140, 2008.

[83] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, (New York, NY, USA), pp. 85–96, ACM, 1997.

[84] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[85] J. E. Freund and R. E. Walpole, *Mathematical Statistics (4th ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[86] "Mimas computer vision toolkit." `http://freshmeat.net/projects/mimas`.

[87] B. Girod, E. Steinbach, and N. Färber, "Performance of the H.263 video compression standard," *J. VLSI Signal Process. Syst.*, vol. 17, pp. 101–111, November 1997.

[88] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," in *ASPLOS'04*, pp. 72–84, 2004.

[89] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[90] P. fei Chuang, H. Chen, G. F. Hoflehner, D. M. Lavery, and W. chung Hsu, "Dynamic profile driven code version selection," *In Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, 2007.

[91] W. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *Journal of Classification*, vol. 1, pp. 7–24, December 1984.

[92] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *IISWC'09*, pp. 55–64, IEEE Computer Society, 2009.

[93] "Facetracker. http://facetracker.net."

[94] "Epfl data set. http://cvlab.epfl.ch/data/pom."

[95] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan, "Variability in the execution of multimedia applications and implications for architecture," in *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, (New York, NY, USA), pp. 254–265, ACM, 2001.

[96] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek, "Mantis: Automatic performance prediction for smartphone applications," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 297–308, USENIX, 2013.

[97] E. Perelman, T. Chilimbi, and B. Calder, "Variational path profiling," in *PACT '05*, pp. 7–16, 2005.