SEEC: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems

by

Henry Hoffmann

B.S., University of North Carolina at Chapel Hill (1999)
S.M., Massachusetts Institute of Technology (2003)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2013

Author .....
Department of Electrical Engineering and Computer Science
October 15, 2012

Certified by .....
Anant Agarwal
Professor
Thesis Supervisor

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

SEEC: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems

by

Henry Hoffmann

Submitted to the Department of Electrical Engineering and Computer Science
on October 15, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Modern computing systems require applications to balance competing goals, e.g.,high performance and low power or high performance and high precision. Achieving the right balance for a particular application and system places an unrealistic burden on application programmers who must understand the power, performance, and precision implications of a variety of application and system configurations (e.g.,changing algorithms or allocating cores). To address this problem, we propose the Self-aware Computing framework, or SEEC. SEEC automatically and dynamically configures systems and applications to meet goals accurately and efficiently. While other self-aware implementations have been proposed, SEEC is uniquely distinguished by its *decoupled* approach, which allows application and systems programmers to separately specify goals and configurations, each according to their expertise. SEEC's runtime decision engine observes and configures the system automatically, reducing programmer burden. This general and extensible decision engine employs both control theory and machine learning to reason about previously unseen applications and system configurations while automatically adapting to changes in both application and system behavior. This thesis describes the SEEC framework and evaluates it in several case studies.

SEEC is evaluated by implementing its interfaces and runtime system on multiple, modern Linux x86 servers. Applications are then instrumented to emit goals and progress, while system services are instrumented to describe available adaptations. The SEEC runtime decision engine is then evaluated for its ability to meet goals accurately and efficiently. For example, SEEC is shown to meet performance goals with less than 3% average error while bringing average power consumption within 92% of optimal. SEEC is also shown to meet power goals with less than 2% average error while achieving over 96% of optimal performance on average. Additional studies show SEEC reacting to maintain performance in response to unexpected events including fluctuations in application workload and reduction in available resources. These studies demonstrate that SEEC can have a positive impact on real systems by understanding high level goals and adapting to meet those goals online.

Thesis Supervisor: Anant Agarwal
Title: Professor

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor

Acknowledgments

I have to begin by thanking my tremendous, intelligent, loving, and beautiful wife, Michelle Hoffmann, who has been extremely supportive of my circuitous journey through grad school. Since Michelle and I met, I have left graduate school twice and returned twice. I would guess that the corresponding changes in our finances and my schedule were hard to take, but I don't know for sure because Michelle never said a word about it. I know it could be hard when I was working late to finish a paper or worrying that a paper wouldn't get accepted (ever). The work contained in this thesis means a great deal to me, but Michelle's loving support of this work and all the inconveniences she had to endure to make it possible is an order of magnitude more important to me.

I also want to thank my son, Zack, who was born several months after I returned to school for the final time. Zack probably won't remember much about my time in grad school, but I will. I often felt torn between doing a good job as a graduate student and a good job as a father, and many times I couldn't do either as well as I wanted. During my time in grad school Zack taught me some important lessons about how sometimes it is more important to play play-dough than it is to run one more simulation. Zack deserves special thanks because he was supporting me without even knowing it.

My parents, Carl and Kathleen Hoffmann, have also been extremely supportive of my academic pursuits throughout my entire life. This was often not easy for them as it required driving me to summer school classes when I was trying to get ahead and also putting up with my bad moods when I was working hard on problems and feeling frustrated. Through out the whole process (from kindergarten to 32nd grade or whatever I just finished) my parents have worked hard to help me in a number of different ways and I would like to thank them for all that effort.

My sister, Arianna, also deserves congratulations for putting up with me as a brother her whole life and still continuing to speak to me. I know I have not been an ideal older brother and that was often because I was busy working on something.

Next, I have to thank Anant Agarwal. Anant is one of my two thesis supervisors and I have been working with him for over eleven years. Over the years, Anant has provided me with a number of remarkable opportunities, including both exciting academic projects and even a startup company. Anant taught me a number of things, but one of the most influential things he told me was his definition of computer architecture: "Everything that happens below the application." Armed with this definition, I have felt secure pursuing a number of different projects in architecture, programming models, and system software.

Srini Devadas is my other thesis supervisor and I owe him a huge thanks. In addition to supervising this work, Srini put a great deal of effort into helping me put my job application materials together. I am sure he got sick of reading my research statement repeatedly, but he kept providing feedback and helping me make it better. Srini also kindly listen to me gripe every time some piece of work included in this thesis got rejected from one conference or another. Also, Srini had to put up with my maddening inconsistency of hyphenation and italicization in earlier drafts of this thesis. (I tend to misuse affect/effect, as well.) Finally, I want to thank Srini for continuing to support this work even beyond graduation by including me and this work on several grant proposals.

There are a number of other people who have been influential in helping me get to this point in my career. To try to avoid leaving anyone out, I will go through things in chronological order, starting with my undergraduate work.

At UNC, I was lucky enough to run into two professors who really encouraged my pursuit of a research oriented career. Jim Anderson helped me get through a lot of self-doubt surrounding my initial inability to understand finite automata and computability. That semester I was a regular in office hours and if Jim hadn't been so patient with me, I don't think I would have made it through that class. Kevin Jeffay also supported me, by supervising an independent study in real-time systems. This work gave me a first glimpse into what research in computer science was all about and I was hooked.

After undergrad, I couldn't face doing any more problem sets, so I decided to get

some practical experience for a little while, and I ended up a MIT Lincoln Laboratory where I was working on high-performance embedded computing systems. I had some great managers in Group 102, especially Bob Bond, who gave me an enormous amount of responsibility for someone just out of school. Some of the other senior people there proved to be great teachers, giving me a firm background in parallel computing; so thanks go to James Lebak, Jim Daly, Glenn Schrader, and Janice McMahon. I also had some great colleagues there who became great friends outside of work including Eddie Rutledge, Mark Fagnani, Tim Largy, and Michelle Anzalone.

After a couple years at Lincoln I decided to go to grad school and I was lucky enough to be accepted to both MIT and the Lincoln Scholars program, which funded my Masters degree. This research was done on the Raw multicore processor where I worked with a number of outstanding people, including Volker Strumpen, who I thanked profusely and deservedly in my Masters Thesis. As a junior student on Raw, I was privileged to work with and learn from more experienced students including Michael Taylor, Walter Lee, Jason Miller, and Dave Wentzlaff.

After completing my Masters, I returned to Lincoln Laboratory where I worked on a project that automatically assigned parallel computing resources to Lincoln's flavor of parallel Matlab. I worked with Bob Bond again, but also Nadya Travinin and Jeremy Kepner. That was a fun project with great colleagues and I wish I had been able to spend more time on it.

After a year and a half, though, I got a chance to return to grad school and finish my PhD. However, not long after getting settled back in school, I left to join Tilera, a startup that commercialized our earlier work on the Raw processor. Tilera took a large number of Raw veterans including Anant, of course, but also Ken Steele, Dave Wentzlaff, Walter Lee, Patrick Griffin, and Ian Bratt. It was a great learning experience to continue the Raw work with these folks. At Tilera, I met a number of other great engineers who were always willing to explain some facet of the hardware, compiler, or OS. This list includes Mat Hostetter, Chris Metcalf, Bruce Edwards, Matt Mattina, Richard Schooler, John Brown III, and GS Rao. I also want to thank Vijay Aggarwal for giving me the opportunity to get in front of Tilera customers,

which was a great learning experience.

After several years at Tilera, I decided that my heart was really in research and I needed to go back to school and finish my PhD. There I got involved in the work that became this thesis. In addition to Anant and Srini, who directly supervised this work, I was lucky to work with several other great advisors on this project. Martin Rinard had a big influence on the work trading application precision for performance and power benefits. Martin also had (and still has, no doubt) lots of great advice on life, often couched in elaborate animal metaphors. Alberto Leva from Politecnico di Milano also helped formulate some of the control solutions described in this work. As with all my projects at MIT, I was fortunate to work with brilliant and helpful students on this project all of whom certainly had a big impact on the final product in this document. Martina Maggio helped tremendously with the SEEC runtime system. Jonathan Eastep helped define the Application Heartbeat API. Sasa Misailovic and Michael Carbin contributed greatly to the creation of adaptive applications. Sabrina Neuman, Eric Lau, Yildiz Sinangil, and Mahmut Sinangil, helped to develop an architecture designed to support the SEEC model presented in this thesis. I was also fortunate to have the guidance and assistance of more than my fair share of postdocs and research scientists including Marco Santambrogio, Jason Miller, Stelios Sidiroglou, and Jim Holt.

I am grateful beyond words to the community of friends, family, advisors, and colleagues that has made this work possible. This is obvious, but worth saying: I could never have done this by myself. It was only through the support of others that I could make it to this point. I wish I had more to offer than just listing names in this document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

For many computer scientists, optimizing a computing system (including application, system software and hardware) has traditionally referred to maximizing its *performance* (i.e.,increasing speed or reducing execution time). Recently, though, changes in the nature of applications and the physical constraints on the machines that support them have given rise to new metrics which can be as important as application speed. For example, many applications (such as those that process massive amounts of data) have a tradeoff between the *precision* of the result they produce and the power or time that they require to produce that result. In addition, many components (e.g.,memory, processing cores, operating system) of a computer system support tradeoffs between their *power* consumption and performance. Given these additional metrics, optimizing a program is no longer about maximizing performance, but is instead an exercise in positioning the application and system at a particular point in the performance/power/precision tradeoff space.

While configuring an application in a multidimensional tradeoff space is difficult, this problem is often further exacerbated by the dynamic nature of modern computing systems. For example, systems must cope with fluctuations in application workload, variations in available power, and failures of system components. Creating an application and system which can dynamically reconfigure itself in a multidimensional

tradeoff space is a challenge beyond most application programmers, yet one that will become increasingly important as the complexity of computing systems increases.

1.2 Background

Dealing with multiple constraints and dynamic fluctuations requires application developers to be experts in system optimization in addition to the expertise already required in a particular application domain. Thus, a great deal of knowledge must be acquired by one user and it is natural to look for ways to reduce this burden. One promising approach is the adoption of *autonomic*, or *self-adaptive* techniques, which can reconfigure themselves automatically [46, 54]. Ideally, a self-adaptive approach will drive the system to a desired operating point in a way that is both *accurate* (i.e.,the application and system meet a desired precision, performance, or power goal) and *efficient* (i.e.,goals are met while optimizing behavior in unconstrained dimensions).

Prior approaches to building self-adaptive systems that ease programmer burden by addressing combinations of power, performance and precision tradeoffs have achieved accuracy and efficiency using one of two strategies. First, many researchers have adopted *system-specific* solutions that work with only a fixed set of components which are known at design time [11, 21, 23, 25, 79]; if new components become available, or existing ones fail, these systems have to be redesigned and re-implemented. Second, other projects have adopted application-specific approaches that require the application programmer to understand the interaction of all components in the system [18, 30, 40, 58, 72, 95] and do not generalize across applications. It is a challenge to build a general system that achieves accuracy and efficiency when the goals, system, and application are not known ahead of time.

1.3 The SEEC Model

This thesis designs and implements the *SElf-awarE Computing (SEEC) Model*, a novel solution for accurate and efficient management of power, performance, and precision tradeoffs in a computer system. Instead of working with a fixed set of adaptations or a fixed application, SEEC is designed to be general with respect to the components and applications it supports. This generality is achieved through two interfaces and a runtime system which allow the construction of an *observe-decide-act* (ODA) loop as illustrated in Figure 1-1.

The SEEC model supports observation through an interface that allows expression of high-level goals (e.g.,target performance, power consumption, or precision) and the current progress towards those goals. The model supports action through a separate interface that allows specification of components and behaviors that can be changed in a system (e.g.,changing the clock speed). Finally, the model supports decision through a runtime system that observes the goals and available actions then determines how to apply those actions and ensure that goals are met accurately and efficiently even in the face of unexpected changes like workload fluctuations or resource failure.

- **Specifying Goals and Progress:** In SEEC, applications explicitly state their goals and other system components measure whether those goals are being met. SEEC uses the Application Heartbeats application programming interface (API) [37] to specify application goals and progress. The API's key abstraction is a heartbeat; applications use a function to emit heartbeats at important intervals, while additional API calls specify goals in terms of this heartbeat. SEEC currently supports three application specified goals: performance, precision, and power. All three goals are specified through the Heartbeats API.
- **Specifying Actions in the System:** SEEC supports a range of actions specified from the application-level, system software level, and the hardware level. SEEC does so by providing an interface that all system components use to specify available actions. This interface is designed to be general and support actions exposed by different developers working at different levels of the system

stack. Actions are specified by describing the *actuators* that implement them. In SEEC, an actuator is a data object with: a name, a list of allowable settings, a function that changes the setting, a set of metrics which the actuator affects (e.g.,performance and power), and the effects of each setting on each metric.

- **Making Decisions:** SEEC's runtime system automatically selects actions to meet goals while reducing cost. The SEEC decision engine is designed to work without prior knowledge of the applications which it will support. In addition, the runtime system will need to react quickly to changes in application load and fluctuations in available resources. To meet these requirements for handling unknown applications and volatile environments, the SEEC decision engine is designed with multiple layers of adaptation. At the lowest-level, SEEC acts as a classical control system, taking feedback, in the form of heartbeats, and using it to tune actuators to meet goals. The classical control system works well given prior knowledge about the application's behavior. Additional layers of adaptation, including adaptive control and machine learning based techniques, allow the SEEC runtime to allocate resources efficiently without prior knowledge of the application, or when the behavior of the actuator diverges from the predicted behavior.

1.4 Example

Consider the development of a video encoder whose goal is to encode thirty frames per second while minimizing power. Furthermore, these goals must be met even though different videos (and even frames within one video) differ in their compute demands and these demands cannot be predicted *a priori*. In a traditional system, the application developer must understand the power and performance tradeoffs of different system configurations (such as number of cores, clock speed, and memory usage) and optimize the encoder to meet performance with minimal power while adapting to both input and system fluctuations.

In contrast, SEEC's runtime system observes application behavior and optimizes

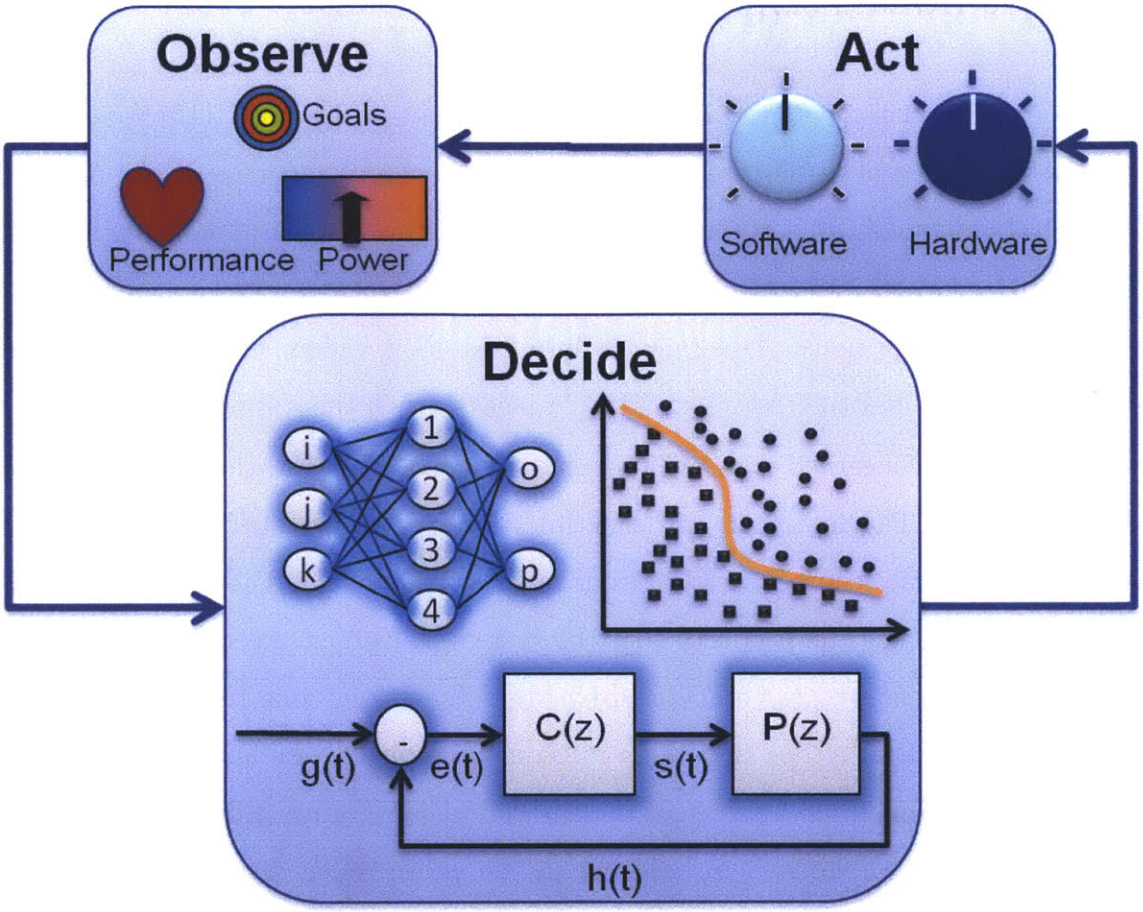


Figure 1-1: The SEEC model. Using SEEC, application developers provide goals and feedback while systems developers describe actions that can be taken in the system. SEEC's runtime decision engine uses a general and extensible mechanism to select actions that meet goals accurately and efficiently.

the system for the application. Using SEEC, the encoder developer indicates the goal of thirty frames per second and the current speed of the encoder. Independently, systems developers specify actions that affect applications (e.g.,allocation of cores, clock-speed, and memory). SEEC's runtime decision engine determines a sequence of actions that achieve thirty frames per second while minimizing power. If an input becomes more difficult, the encoder does not meet its goals and SEEC assigns it additional resources. If an input becomes less difficult, the encoder exceeds its goals, and SEEC will reclaim resources to save power. In addition, SEEC continuously updates its internal models of applications and systems, so it can adapt if new resources become available or if existing resources fail.

1.5 Properties of SEEC

SEEC's runtime decision engine is heavily based in control theoretic techniques. This use of well-founded decision mechanisms allows important properties of the decision system to be demonstrated analytically. Specifically, this analysis demonstrates that SEEC will accurately and efficiently converge to achieve a target goal given some set of initial assumptions. Different assumptions allow different guarantees about convergence time and the trajectory SEEC takes to reach this target.

This thesis is concerned with five properties that describe how the decision engine meets its goals. The first four are sometimes referred to as SASO properties and consist of: stability, accuracy, settling time, and overshoot [36]. The final property is efficiency. Stability is the property that the system converges. Accuracy is achieved when the system converges to the desired target. Settling time refers to the time that the system takes to become stable. Overshoot describes the maximum value by which the system may miss its target on the way to converging. Efficiency describes how close to optimal the system behavior in the uncontrolled dimension; e.g.,when controlling performance, how much is power reduced for that performance goal.

In general, different levels of adaptation in the SEEC runtime system provide different tradeoffs in terms of their flexibility and the guarantees they provide. The classic control system provides full guarantees about system behavior for each of the five desired properties, but these guarantees are based on a fairly rigid set of assumptions. In contrast, machine learning starts with an extremely flexible set of initial assumptions, but provides the fewest guarantees about system behavior. The intermediate layers of adaptive control provide intermediate sets of tradeoffs.

1.6 Systems Case Studies

In addition to an analytical evaluation of SEEC, this thesis presents several cases studies showing how SEEC can be used to build self-adaptive systems. These studies include:

- Creating seven different adaptive systems on a Linux/x86 system. These systems show simple examples of SEEC managing performance, power and precision goals.
- Building a system that manages power/performance tradeoffs accurately and efficiently for the PARSEC benchmarks on two different Linux/x86 machines with different power/performance characteristics.
- Constructing a system that automatically turns statically configured applications into dynamic applications which self-manage precision/peroformance/power tradeoffs.
- Demonstrating SEEC reacting to unforeseen events like errors in SEEC's internal models or the failure of some system components.

These case studies demonstrate how a small set of application changes, combined with the SEEC runtime system, can enable a tremendous shift in system behavior. Specifically, simply by understanding high-level application goals, the runtime can adapt the system configuration to accurately and efficiently meet those goals. Across a range of applications on different machines, SEEC is found to drive the system to within a few percent of a target goal while achieving close to optimal behavior in other dimensions. Additional studies show how systems built with SEEC can automatically maintain goals even in the face of unforeseen events.

1.7 Scope

SEEC navigates power/performance/precision tradeoffs. Specifically, SEEC is meant for systems which have a constraint (or goal) in one of these three dimensions and have some freedom to vary behavior in one or more other dimensions. Such systems are already in use and may become even more important as physical phenomena (e.g.,dark silicon [31, 66]) place hard limits on some dimensions. In contrast, some systems are not constrained, but instead support a "performance at all costs" model and these systems are not a good match for SEEC.

SEEC delivers accuracy and efficiency for a wide range of applications, including

those with high variance in their performance; however, there are some applications which are not a good match for this approach. First, extremely short-lived applications may not provide enough feedback for SEEC to converge, unless users are willing to allow SEEC to collect data from multiple innovations of an application. In addition, other applications may not be divisible into small units which indicate the performance of the larger application. Such applications will not benefit from SEEC's active monitoring approach.

1.8 Contributions

This thesis makes the following contributions:

- It designs the Application Heartbeats API, which allows application developers to express performance, power, and precision goals to the rest of the system. Other system components can then read these goals and the application's progress towards them. This interface is unique in that it allows applications to explicitly indicate design decisions (such as the preference for meeting a target power consumption over maximizing performance) that were previously implicit in the application and unknown to the rest of the system.
- It designs a separate Actuator Interface which allows systems developers to express available actions that change the power/performance/accuracy tradeoffs of a system.
- It designs and develops the SEEC runtime decision engine which translates application specified goals into system specified actions. The runtime decision engine has several novel features that allow it to meet goals accurately and efficiently. The first feature is its unique incorporation of adaptive control to meet the goals of previously unseen applications accurately. The second feature is the use of adaptive actuator selection, which allows SEEC to meet goals efficiently on different machines or with different sets of actuators. Finally, SEEC incorporates machine learning to ensure accuracy and efficiency even when nothing is known about application and system behavior.

- It creates a *decoupled* approach to the design and implementation of adaptive systems. In prior work, adaptive systems were designed monolithically and required one developer to specify all phases of the ODA loop. SEEC enables a unique approach to the design of adaptive systems, one that allows a separation of concerns. Using SEEC, application programmers specify goals and progress, but do not need to be aware of the actions available in the system. Similarly, systems developers specify actions that the system can take, but do not need to understand the applications they support. Finally, no developer needs to understand decision making techniques, but instead application and systems developers rely on the SEEC runtime decision engine to translate observations into actions.
- It analyzes properties of the SEEC runtime decision engine. This analysis describes tradeoffs between different sets of initial assumptions and the guarantees provided by different layers of SEEC's runtime decision engine. This analysis can guide the customization of the SEEC system for different deployments with differing requirements.
- Through numerous case studies, it demonstrates how SEEC can be used to develop real adaptive systems. These case studies implement SEEC on modern Linux x86 servers and test SEEC's ability to handle a range of different application behaviors, including highly regular and highly variant applications. In these studies, SEEC's accuracy and efficiency is compared to other approaches including oracle systems (which cannot be implemented) that represent the best possible results. SEEC is found to be quite accurate, typically within a few percent of the target goal. In addition, SEEC is shown to be efficient, in many cases delivering results close to what could be achieved by the oracle system. For example, when managing performance and power tradeoffs, SEEC meets power goals while providing 96.1% of the maximum performance and meets performance goals while exceeding the minimal power consumption by only 7.2%. Finally, additional studies show how SEEC maintains goals while reacting to unforeseen fluctuations in the environment including changes in ap-

plication workload and changes in available resources.

The remainder of this document is organized as follows. Chapter 2 describes related work and highlights some of the key innovations of SEEC compared to prior approaches. Chapter 3 describes the SEEC approach including both interfaces and the SEEC runtime system. Chapter 4 describes how application and system developers can make use of the SEEC framework. Chapter 5 describes the case studies we use to evaluate SEEC. Chapter 6 evaluates the SEEC decision engine analytically. Chapter 7 wraps up the thesis by discussing lessons learned, future work and conclusions.

Chapter 2

Background and Related Work

This chapter discusses some of the background and prior research on adaptive systems and the relationship between these existing systems and the SEEC approach. The implementation and deployment of adaptive systems has the potential to make the process of programming complicated systems much easier, leading to reduced development time and greater efficiency. One major challenge in the design and implementation of adaptive systems is to make such systems as general as possible with respect to the applications they support and the components (i.e.,actions) that they manage. Generality is important because a more general approach should be more widely applicable and have a greater impact on reducing programmer burden.

Prior approaches to achieving generality in adaptation tend to fall into two categories: those that are *application-specific* and those that are *system-specific*. Application-specific approaches tend to be general with respect to the adaptations that they support but limited in their support for new applications. System-specific approaches tend to be general with respect to the applications they support but are built for specific sets of system hardware and software components and have difficulty incorporating new components without redesign.

Part of the problem with these prior approaches is that they tend to make a single developer responsible for the entire process of specifying the observe-decide-act (ODA) loop characteristic of adaptive computing. Application-specific approaches make the application developer responsible for all phases, when application-level developers re-

ally should focus on specifying how to observe the application and, possibly, how the application can adapt. In contrast, system-specific approaches tend to make the systems developer responsible for all ODA phases, when system developers should focus on how to specify low-level observations and actions that the system can implement.

In contrast to prior approaches, the SEEC model is designed to be general with respect to both applications and systems components. SEEC achieves this by having separate interfaces that can be used by different developers so that each specifies the components of the ODA loop with which they are most familiar.

This chapter describes both how SEEC is influenced by and differs from prior work. The chapter first presents an overview of adaptive systems, then describes existing application- and system- specific approaches, and finally highlights some unique features of SEEC.

2.1 Overview of Self-Adaptive Systems

Self-aware, or autonomic, computing has been proposed as one method to deal with the rising complexity of computer systems [46, 54], and adaptive systems have been implemented in both hardware [4, 11, 23, 25] and software [77]. Some example systems include those that that manage resource allocation in multicore chips [11], schedule asymmetric processing resources [83, 76], optimize for power [53], and manage cache allocation online to avoid resource conflicts [93]. In addition, languages and compilers have been developed to support adapting application implementation for performance [89, 5], power [6, 82], or both [39]. Adaptive techniques have been built to provide performance [9, 58, 71, 78, 95] and reliability [14] in web servers. Real-time schedulers have been augmented with adaptive computing [12, 34, 60]. Operating systems are also a natural fit for self-aware computation [17, 45, 52, 69]. Self-aware techniques are prominent in industry; companies such as IBM [41] (e.g.,IBM Touchpoint Simulator, the K42 Operating System [52]), Oracle (e.g.,Oracle Automatic Workload Repository [70]), and Intel (e.g.,Intel RAS Technologies for Enterprise [43]) have released products with self-aware capabilities.

Table 2.1: Comparison of several self-aware approaches.

	ControlWare [95]	Tunability Interface [18]	METE [79]	Choi & Yeung [23]	Bitirgen et al. [11]	SEEC
Observation	Application	System	System[a]	System	System	Application & System
Decision	Control[b]	Classifier	Adaptive Control	Hill Climbing	Neural Network	Adaptive Control & Machine Learning
Action	Application	Application	Application	System	System	Application & System
Handles unknown applications?	No	No	No	Yes	Yes	Yes
Add actions without redesign?	Yes	Yes	Yes	No	No	Yes

[a]The paper allows a transducer that converts application goals to IPC, but does not support the case where application performance does not translate directly to IPC.

[b]The paper claims that adaptive control is supported, but the tested implementations use classic control.

While the use of self-adaptive techniques has become common, there are many challenges still to overcome, some of which are described in [47, 65, 77]. One challenge facing researchers is the development of general and extensible self-aware implementations. It is important to distinguish the concept of a general *implementation* from a general *technique* and note that many general techniques have been identified. For example, reinforcement learning [88] and control theory [36] are both general techniques which can be used to create a variety of adaptive systems. However, general techniques tend to be customized when deployed, and through that customization they become *problem-specific*; i.e.,they are designed with a single, narrow problem in mind and do not generalize. Generalization of an implementation can be limited in multiple ways: 1) they may be *application-specific*, i.e.,built to manage a specific application (e.g.,a webserver [78]) and 2) they may be *system-specific*; i.e.,general with respect to applications but handling only a fixed and known set of actions (e.g.,managing the hardware for a memory controller [44]).

2.2 Application-Specific Approaches

Researchers have developed several frameworks that can be customized for a specific application. These approaches include: ControlWare [95], Agilos [58], SWiFT [30], the *tunability interface* [18], AutoPilot [72], and Active Harmony [40]. One limitation to the generality of these approaches is their exclusive focus on customization at the application level. For example, ControlWare allows application developers to specify application-level feedback (such as the latency of a request in a web server) as well as application-level adaptations (such as admission control for requests). Unfortunately, these approaches do not allow application-level feedback to be linked to system-level actions performed by the hardware, compiler, or operating system. Furthermore, once these frameworks are customized, they lose their generality. In contrast, SEEC allows applications to specify the feedback to be used for observation, but does not require application developers to make decisions or specify alternative actions (application developers can optionally specify application-level actions, see Section 3.2). Addition-

ally, the SEEC runtime system is designed to handle previously unseen applications and can do so without redesign or re-implementation. Thus, SEEC's decoupled approach allows application programmers to take advantage of underlying system-level adaptations without even knowing they are available.

2.2.1 Limitations of Application-Specific Approaches

This section presents a small experiment illustrating how application-specific approaches can fail to generalize. This experiment is run on a Linux/x86 system. First the facesim benchmark (from PARSEC [10]) is modified so it can adjust its core usage and processor speed to meet a target performance while minimizing power. Separately, the blackscholes benchmark is modified to do the same. Both benchmarks target a performance that is 50% of the maximum achievable, and performance and power are measured for these benchmarks. After running each benchmark, the management systems for each are switched, so that facesim is managed by the system designed for blackscholes and blackscholes is managed by the system designed for facesim.

Figure 2-1 shows the results of this study. The x-axis shows the two benchmarks and the y-axis shows the performance per Watt measured for these two benchmarks and normalized to the best result for that application. There are two bars shown per benchmark. The first shows the results when using a system designed for blackscholes. The second shows the results when using a system designed for facesim.

The results in Figure 2-1 demonstrate some of the issues with application-specific approaches. Both approaches are comparable when controlling facesim; however, when controlling blackscholes, the system designed for facesim does a poor job. In fact, this system achieves just over 60% of the possible performance per Watt.

When looking at the behavior of blackscholes over time, the situation is revealed to be even worse. Figure 2-2 shows the performance of blackscholes as a function of time when controlled by a system designed for blackscholes and a system designed for facesim. In this figure time is shown on the x-axis and performance is shown on the y-axis.

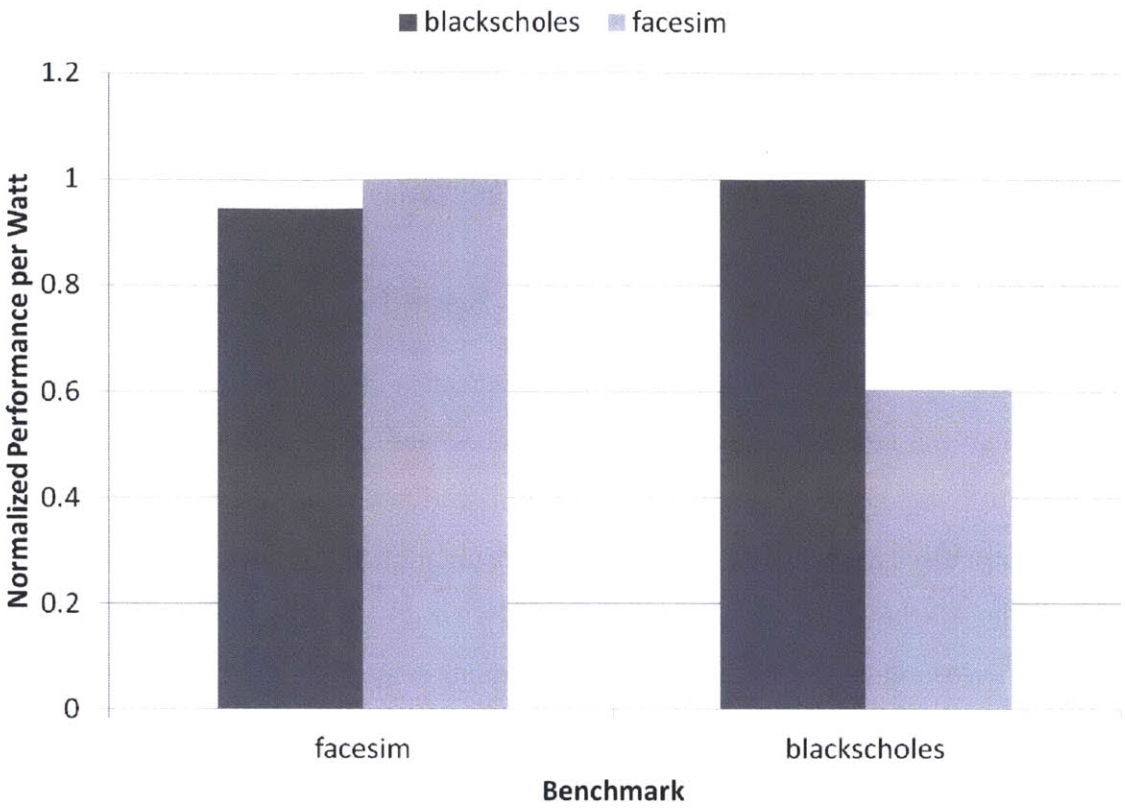


Figure 2-1: Applying a system designed for one application to a new application.

As shown in the figure, when controlled by the system designed for facesim, blackscholes' performance never converges. This represents a lack of accuracy. Indeed, the performance oscillates between two extremes. When controlled by the system designed for blackscholes, however, the performance converges to the desired value after some initial instability.

These results demonstrate how application-specific approaches can fail to provide accuracy and efficiency when generalized to work with new applications. This lack of generality means that additional work has to be done for every application a developer wants to create. This work includes understanding the interaction of the new application and the available adaptations, which can be a large and tedious task. We note, however, that most application-specific approaches (including all listed as reference in this section) provide good support for decision making so that application developers do not have to micro-manage the decision making mechanism.

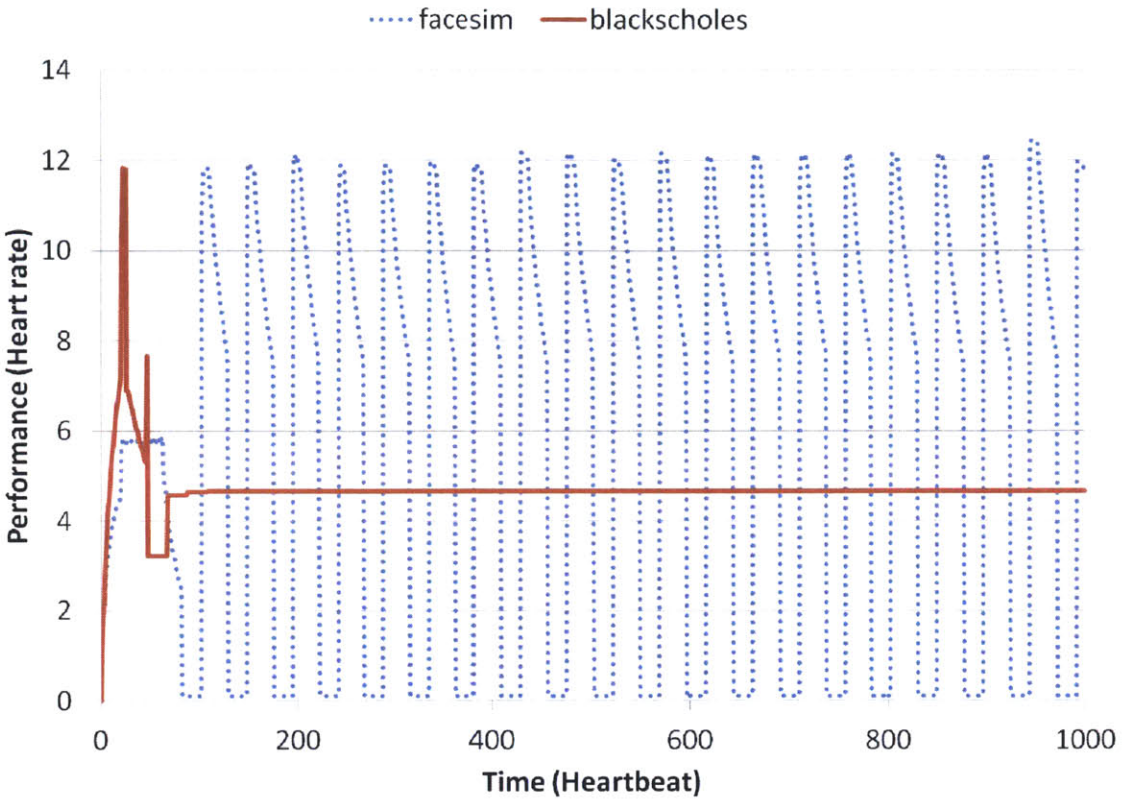


Figure 2-2: Accuracy of application-specific approaches.

One of SEEC's goals is to create a system that will work with new applications without requiring the application developer to understand how the application interacts with the adaptations available in the system. Instead, SEEC requires only that applications emit some high-level goals and a measure of progress towards those goals. Application developers do not need even to know what adaptations are available in the system. This generality has two clear benefits. First, this approach should reduce the amount of knowledge required by the application developer. Second, because applications do not contain any system-specific information, they can be ported to new platforms without rewrite. These benefits illustrate some of the ways the SEEC approach can reduce programmer burden.

2.3 System-Specific Approaches

Many research projects have explored ways to manage a single system component's tradeoffs. Examples include: reducing power consumption through idling [28], limiting cache usage [7], managing dynamic voltage and frequency scaling (DVFS) [91], creating adaptive applications [39], and managing memory controllers [44]. These approaches move the component to a desired operating point accurately and efficiently; however, it is unclear how the system will act when multiple such components are adjusted independently.

Other researchers have developed self-aware approaches to adapt multiple system-level actions (made available in system software or hardware) to handle a variety of previously unseen applications. Such system-level approaches include machine learning hardware for managing a memory controller [44], a neural network approach to managing on-chip resources in multicores [11], a hill-climbing technique for managing resources in a simultaneous multi-threaded architecture [23], techniques for adapting the behavior of super-scalar processors [4], a control system for allocating resources in a multicore [79], and several operating systems with adaptive features [17, 45, 52, 69].

While these approaches allow system-level adaptations to be performed without input from the application programmer, they suffer from other drawbacks. First, application performance must be inferred from either low-level metrics (e.g.,performance counters [4] or instructions-per-clock (IPC) [79]) or high-level metrics (e.g.,total system throughput [11]), and there is no way for the system to tell if a specific application is meeting its goals. In contrast, SEEC allows systems developers to specify available actions independently from the specification of feedback that guides action selection. In addition, these prior systems work with a fixed set of available actions and require redesign and re-implementation if the set of available actions changes. For example, if a new hardware resource becomes available for allocation, the control system used by METE [79] will have to be redesigned and re-implemented. For this reason, we refer to these approaches as *closed* adaptive systems because they are not designed to be general with respect to the set of adaptations they support. In contrast, SEEC

can combine actions specified by different developers and learn models for these new combinations of actions online without a redesign of its control system.

2.3.1 Limitations of System-Specific Approaches

One major drawback of closed adaptive systems is that they cannot be composed. This drawback will become increasingly important as the number of adaptive components deployed on a single system increases.

To illustrate the problems of composing closed adaptive systems, we present the following experiment. Using the Graphite simulator [68], we run the barnes application from the SPLASH2 benchmark suite on a multicore system with two possible adaptations: the total number of cores assigned to it (from 1-64, by powers of 2), and the size of the L2-cache on each core (from 16-256 KB, by powers of 2). For each combination of core allocation and cache size, we measure the performance of the application and the total energy consumed. The results are shown in Figure 2-3, where the x-axis shows energy and the y-axis shows instructions per second. The solid diamond points represent all tested configurations. The squares show configurations that appear optimal for a closed system which only considers cache adaptations. The triangles show possible configurations for a system that only considers core allocations. The best configurations are the ones with highest performance and lowest total energy; i.e.,the Pareto-optimal frontier which is depicted by those diamond points that are connected by a line in the figure. Notice that both triangles and squares appear to the right of the Pareto frontier, and these points represent configurations that closed systems would believe to be optimal, but, in fact, are sub-optimal for the overall system.

These sub-optimal points can prevent the system from ever actually settling into a Pareto-optimal configuration, because optimizing a single adaptation without information from others will cause the overall system to jump between sub-optimal configurations. For example, one system might increase core count, and observe that it overshoots the performance goal. However, unbeknownst to it, the sizes of the caches were also increased at the same time, which was the reason for the overshoot

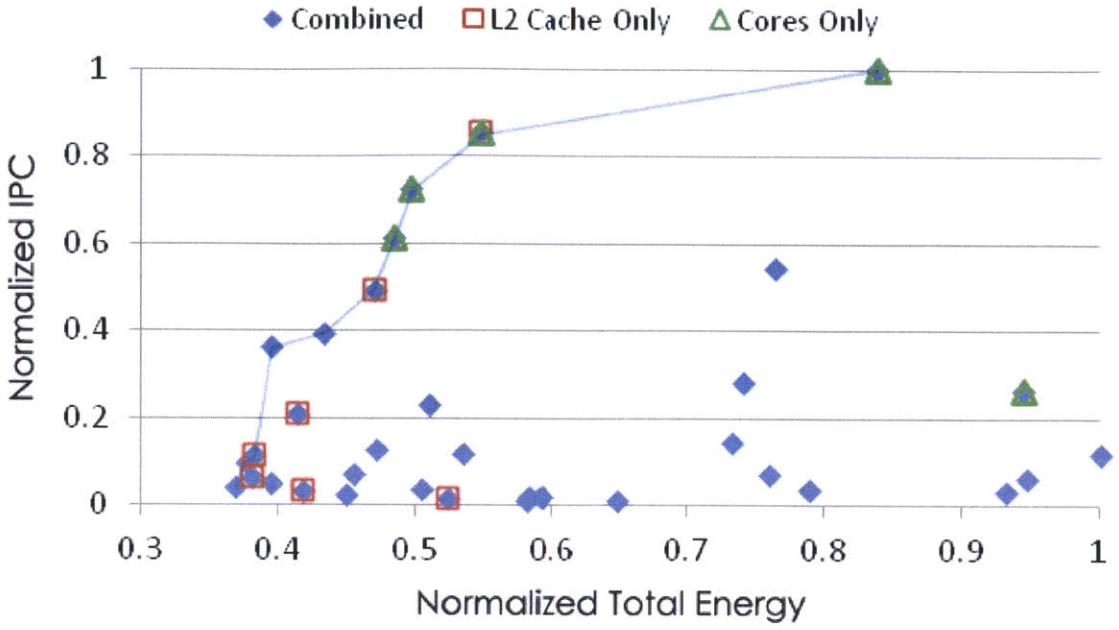


Figure 2-3: Efficiency of closed adaptive systems.

of the performance goal. Since neither system is aware of the other, they both pull back their adaptation and end up undershooting the performance goal. The greater number of sub-optimal configurations that an independent closed system might consider as optimal, the worse this problem becomes.

To avoid sub-optimal configurations, the SEEC model provides a general interface allowing adaptations supported by different system components to be described by their designer and then manipulated by the SEEC runtime decision system. For example, this interface can be used to describe both operating system-level actions (e.g.,allocation of cores to an application [62]) and hardware-level actions (e.g.,reconfiguration of the hardware data cache [7]). Given this information, the SEEC runtime system can coordinate adaptation to keep the system on the Pareto optimal curve shown in Figure 2-3. To support this model, hardware must be explicitly designed to expose adaptations instead of attempting to adapt as a closed system.

2.4 Unique Features of the SEEC Approach

Table 2.1 highlights the differences between SEEC and some representative prior adaptive implementations. The table includes approaches for both application- and system-specific adaptation. For each project, the table shows the level (system or application) at which observation and actions are specified and the methodology used to make decisions. In addition, the table indicates whether the system can handle previously unseen applications and whether the emergence of new actions requires redesign of the decision engine.

As shown in Table 2.1, SEEC is unique in several respects. SEEC is the only system designed to incorporate observations made at both the system and application level. SEEC is also the only system designed to incorporate actions specified at both application and system level. SEEC's novel decision engine is, itself, an adaptive system combining both machine learning and control theory and capable of learning new application and system models online. Finally, SEEC is the only adaptive system that can both handle previously unseen applications and incorporate new actions without redesign of its decision engine.

2.4.1 Importance of Application-Level Feedback

SEEC distinguishes itself from many existing systems by incorporating application-level feedback in the form of heartbeats. This distinction can be critical for applications that execute data dependent code, i.e., where the processing changes based on the input data.

As an example of why this distinction is important, consider the x264 benchmark from PARSEC. This benchmark performs video encoding and the key metric of performance is therefore frames per second. It is easy to indicate this goal by issuing a heartbeat every time a frame is encoded. To show the benefits of using heartbeats, we collect 16 different HD video inputs from xiph.org and measure both the heart rate (or frames per second) and the instructions per second (micro-ops retired per second) on a Linux x86 Xeon server.

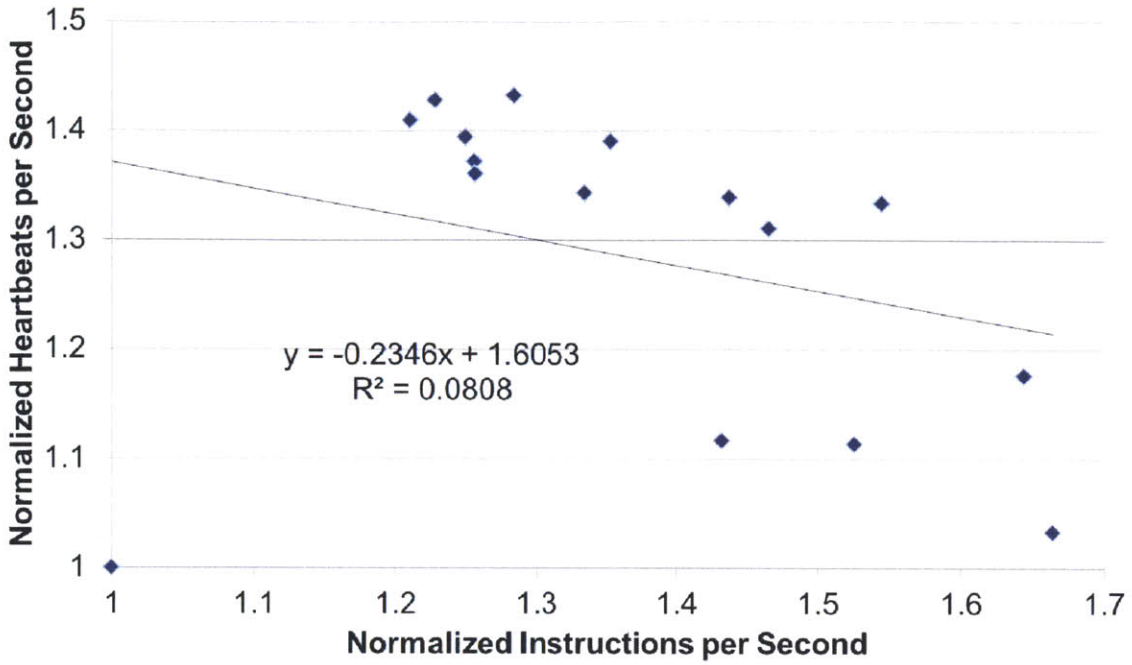


Figure 2-4: Instructions per second vs. heart rate.

Figure 2-4 shows the results with instructions per second on the x-axis and heartbeats (or frames) per second on the y-axis. In addition, the figure shows the trend line and the R^2 value for this data. The results show that instructions per second is a poor predictor of actual application-level performance goals for this benchmark. In fact, there is a slight negative correlation between the two, which means that a system which uses instructions per second to allocate resources to x264 would do the wrong thing and under-allocate resources when they are most needed. This negative correlation is because harder inputs cause x264 to spend more time in *motion estimation*, which is expensive, but implemented with highly efficient code. Overall, these results show the importance of using application-level feedback for applications whose performance is data-dependent; i.e.,whose behavior or progress varies as the data processed varies. Since we cannot know if application progress is data dependent ahead of time, the SEEC framework adopts the stance of using application feedback for all applications.

2.4.2 Comparison with Other Control-Based Approaches

SEEC makes decisions about how to schedule actions using a control theoretic decision engine. Hellerstein et al [36] and Karamanolis et al [45] have both suggested that control systems can be used as "off-the-shelf" solutions for managing the complexity of modern computing systems, especially multi-tiered web-applications. While control theory represents a general *technique*, specific deployments require identification of a feedback mechanism and translation of an existing control model into software. These are difficult concepts to generalize, which leads to solutions that address a specific computing problem (e.g.,managing utilization in a web server) using control theory, but cannot handle other applications or other actuators [59, 71, 84, 86]. In contrast, SEEC provides a general runtime that is not tied to a particular application or set of actuators. The runtime works with a range of applications and system components, and thus overcomes some limitations of prior approaches recently identified by Hellerstein [35].

In comparison with existing control-based approaches, one of the unique contributions of SEEC is its generalized control strategy. SEEC's control system is designed to work with any adaptations that affect power/performance/precision tradeoffs. Whereas prior approaches would control a specific actuator (e.g.,CPU utilization), the SEEC control system computes a generalized control signal that describes how behavior needs to change. The SEEC runtime system then translates this signal into a specific set of actions. By separating the computation of the control from the setting of the actuators, SEEC creates a general solution that can work with different sets of actuators without redesign or re-implementation.

Chapter 3

The SEEC Framework

This chapter designs and develops the SEEC framework for accurate and efficient management of performance, power, and precision tradeoffs. The chapter begins with a brief overview of SEEC and then describes the interfaces and runtime system that comprise the framework.

A key novelty of the SEEC approach is its decoupling of observe-decide-act (ODA) loop implementation. The SEEC framework achieves this decoupling through the use of two interfaces and a runtime system. Thus, there are three distinct roles in the SEEC model: application developer, system developer, and the SEEC runtime decision system. Table 3.1 shows the responsibilities of each of these three entities for the three phases of ODA execution: observation, decision, and action. The application developer is responsible for indicating the application's goals and current progress toward those goals. The systems developer is responsible for indicating a set of actions and a function which implements these actions. The SEEC runtime system is responsible for providing a generalized and extensible decision engine to coordinate actions and meet goals. In practice, roles can overlap: application developers can supply application-level actions and systems developers can provide system-level observations.

One difficulty implementing a decoupled adaptive system is designing a decision engine which can support a wide range of applications and actions. Given that difficulty, the majority of this section focuses on SEEC's decision engine which augments

Table 3.1: Roles and Responsibilities in the SEEC model.

Phase	Applications Developer	Systems Developer	SEEC Runtime
Observation	Specify goals and performance	-	Read goals and performance
Decision	-	-	Determine how to meet goals with minimum cost
Action	-	Specify actions and initial models	Initiate actions and update models

a classical control system with several novel features. SEEC uses *adaptive* control to tailor its response to previously unseen applications and react swiftly to changes within an application. Second, SEEC implements *adaptive actuator selection* to make efficient use of available components. Finally, SEEC incorporates a *machine learning engine* used to determine the true costs and benefits of each action online. This hierarchy of adaptation in the SEEC system is illustrated in the block diagram of Figure 3-1.

This chapter first presents the interface used for specifying goals and progress in Section 3.1. Next, the interface for specifying available actions is described in Section 3.2. Section 3.3 describes the runtime decision engine. We note that observation and action require developer input, but SEEC's runtime handles decisions without requiring additional programmer involvement.

3.1 Observe

As described in Section 2.4.1, low-level hardware metrics of progress (e.g.,instructions per second) do not necessarily correlate with application-level metrics of progress (e.g.,frames per second). Therefore, SEEC exposes an interface that application programmers can use to set goals and indicate progress; the SEEC runtime system can then observe these goals and determine whether or not they are being met. This observation interface is based on the Application Heartbeats API [37]. The API's key abstraction is a heartbeat; applications use a function to emit heartbeats at im-

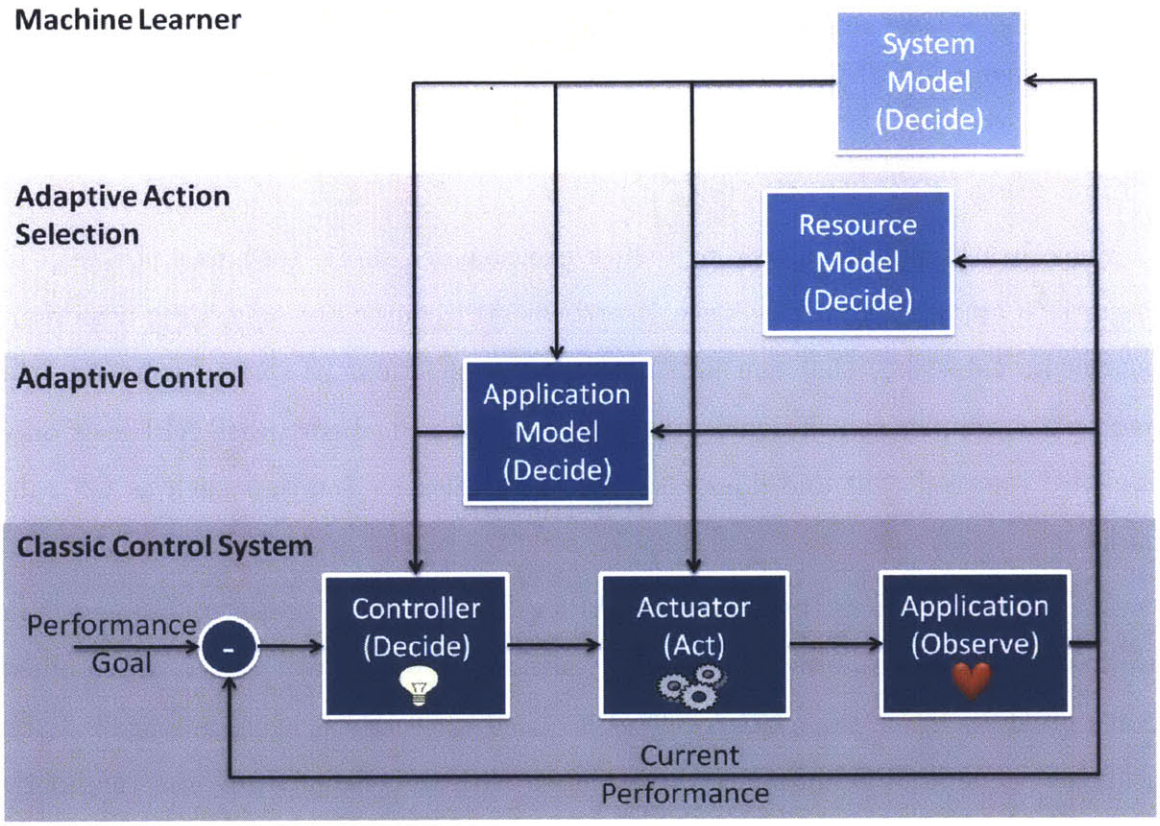


Figure 3-1: SEEC block diagram. SEEC's decision engine is built with multiple layers of adaptation. At the lowest level, it is based on classical control theory. Additional adaption incorporates adaptive control to adjust to application-specific characteristics, adaptive actuator selection to adjust to different resource usage characteristics, and reinforcement learning to adjust to unknown or changing costs and benefits.

portant intervals, while additional API calls specify goals in terms of this heartbeat. SEEC currently supports three application specified goals: performance, precision, and power. Performance is specified as a target heart rate or a target latency between specially tagged heartbeats. Precision goals are measured as a *distortion*, or linear distance from an application defined nominal value [39], measured over some set of heartbeats. Power and energy goals can be specified as target average power for a given heartrate or as a target energy between tagged heartbeats. This interface is not exclusive to SEEC; heartbeat data can be read by any other process in the system. In fact, several other projects have built adaptive systems based on heartbeats without using SEEC [27, 26, 81].

Since heartbeats are meant to reduce programmer effort, they must be easy to insert into applications. The basic Heartbeat API consists of only a few functions (shown in Table 3.2) that can be called from applications or system software. To maintain a simple, conventional programming style, the Heartbeats API uses only standard function calls and does not rely on complex mechanisms such as OS callbacks.

The key function in the Heartbeat API is *HB_heartbeat*. Calls to *HB_heartbeat* are inserted into the application code at significant points to register the application's progress. Each time *HB_heartbeat* is called, a heartbeat event is logged. Each heartbeat generated is automatically stamped with the current time and thread ID of the caller. In addition, the user may specify a tag that can be used to provide additional information. For example, a server application may use one tag to denote the arrival of a request and another tag to signal its completion. Tags can also be used as sequence numbers in situations where some heartbeats may be dropped or reordered. Finally, tags can be used to determine the latency between events, for example, the time between a request arriving at an application and the completion of the task. An optional argument to this function can be used to indicate the current precision. For applications with power or energy goals, the system will independently read available power information and add this to the record of heartbeat data.

Many applications will be concerned with performance in terms of throughput,

Table 3.2: Heartbeat API functions

Function Name	Arguments	Description
HB_initialize	window[int], buffer_size[int]	Initialize the Heartbeat runtime system and specify how many heartbeats will be used to calculate the default average heart rate and how many heartbeats to buffer
HB_heartbeat	tag[int], precision[float]	Generate a heartbeat to indicate progress, precision is an optional argument indicating current precision level
HB_current_rate		Returns the average heart rate calculated from the last *window* heartbeats
HB_get_current_heartbeat		Returns the tag, time-stamp, current heart rate, power, and accuracy measured the last time a heartbeat was generated
HB_set_target_rate	min[float], max[float]	Called by the application to indicate to an external observer the average heart rate it wants to maintain
HB_get_target_min_rate		Called by the application or an external observer to retrieve the minimum target heart rate set by HB_set_target_rate
HB_get_target_max_rate		Called by the application or an external observer to retrieve the maximum target heart rate set by HB_set_target_rate
HB_set_target_latency	min[float], max[float], tag1[int], tag2[int]	Called by the application to indicate to an external observer the average latency it wants to achieve between two heartbeats with the given tags
HB_get_target_min_latency	tag1[int], tag2[int]	Called by the application or an external observer to retrieve the minimum target latency set by HB_set_target_latency
HB_get_target_max_latency	tag1[int], tag2[int]	Called by the application or an external observer to retrieve the maximum target latency set by HB_set_target_latency
HB_set_power_goal	min [int], max [int]	Sets desired power
HB_get_power_goal		Called by observer to retrieve power information
HB_set_energy_goal	min [int], max [int], tag1 [int], tag2 [int]	Sets desired energy between tags
HB_get_energy_goal	tag1 [int], tag2 [int]	Called by observer to retrieve power information
HB_set_precision_goal	min [int], max [int]	Sets desired precision
HB_get_precision_goal		Called by observer to retrieve desired precision
HB_set_goal_priority	goal [int]	Set priority goal to performance (goal=0), power (goal=1), or precision (goal=2)
HB_get_goal_priority		get priority goal
HB_get_history	n[int]	Returns the time-stamp, tag, and thread ID for the last n heartbeats

or the rate at which heartbeats are generated. For example, a video encoder may generate a heartbeat for every frame of video. For these applications, it is likely that the key metric will be the average frequency of heartbeats or *heart rate*. The *HB_current_rate* function returns the average heart rate. It is important to note that SEEC (and Application Heartbeats) do *not* assume that the emission of heartbeats is regular. Continuing the video encoder example, it is possible that different frames of video will take different times to encode, depending on their complexity. It is up to the system (SEEC or another heartbeat reader) to handle variance in heartbeat data. It is not a requirement that the application issue heartbeats at regular time intervals. Rather, applications should issue heartbeats at a place that is meaningful to the application. As will be discussed in Section 3.3 much work has been put into the decision engine to address the possibility of irregular heartbeats.

Different applications and observers may be concerned with either long- or short-term trends. Therefore, it is possible to specify the number of heartbeats (or *window*) used to calculate the average heart rate, power, and precision. There may be some tension between the application registering the heartbeats and the system service reading the heartbeats. We assume that the application knows which window size is most appropriate for the computation it is performing, so the API allows the application to set the window size and this size is the default used whenever an external system requests the current heart rate. A system service that wants to calculate a windowed average using a different window size can make use of the *HB_get_history* function discussed in greater detail below.

Applications with real-time deadlines or performance goals will generally have a target heart rate that they wish to maintain. For example, if a heartbeat is produced at the completion of a task, then this corresponds to completing a certain number of tasks per second. Some applications will observe their own heartbeats and take corrective action if they are not meeting their goals. However, the value in the Heartbeats approach lies in communicating the performance and goals of an application to external systems which can also adapt their behavior and increase performance. To enable this communication, the API provides the *HB_set_target_rate* function which

allows the application to specify a target heart rate range. If a system service sees that an application is not meeting its goals, it can adapt its behavior to assist the application. Alternatively, if an application is achieving greater performance than it requires, services can reclaim some resources from that application.

Some applications may be more interested in reducing latency than increasing throughput. For example, a server might want to minimize the latency of processing a given type of request rather than strictly maximizing the number of requests serviced. In this case, applications can specify a target latency desired between two heartbeat tags using the function *HB_set_target_latency*. In the server example, one tag value would be used to indicate receiving a request while a distinct value indicates request completion. The *HB_set_target_latency* function allows the server to specify the desired latency between these two events. Other functions allow external observers to determine the desired latency and then make their own decisions as to how to help the application meet this goal.

In addition to performance goals, the Heartbeats API allows applications to set power/energy and precision goals. These functions and there usage is essentially the same as those that establish performance goals. One difference is that applications must explicitly state their current precision, as this is a quantity that can only be derived from the application. In addition, applications are not responsible for reporting their power consumption. Instead it is assumed that any system which is controlling power has a way to measure this value[1]. An API function allows applications to specify the priority goal in the system. For example, some systems might want to guarantee performance and minimize power or maximize precision, while other systems might want to maintain a set power consumption and maximize performance.

When more in-depth analysis of heartbeats are required, the *HB_get_history* function can be used to get a complete log of recent heartbeats. It returns an array of the last n heartbeats in the order that they were produced. This allows the user to examine intervals between individual heartbeats or filter heartbeats according to

[1]The studies in this thesis use a WattsUp? power meter, which reports full system power consumption at one second intervals.

their tags. The maximum value of n is determined by the *buffer_depth* parameter passed to the heartbeat initialization function.

While not necessary for SEEC, some systems may contain hardware that can automatically adapt using heartbeat information. For example, hardware could automatically adjust its own supply voltage to maintain a desired heart rate in the application. Therefore, it must be possible for hardware to directly read from the heartbeat buffers. In this case the hardware must be designed to manipulate the buffers' data structures just as software would. To facilitate this, an additional standard must be established specifying the components and layout of the heartbeat data structures in memory. We leave the establishment of this standard and the design of hardware that uses it to future work.

3.2 Act

The SEEC model provides a separate, system programmers interface for specifying actions that can be taken in the system, which is summarized in Table 3.3. The key abstraction in the SPI is a *control panel* populated with *actuators*. The SEEC runtime exports a control panel and systems developers use the SPI to register new actuators. The actuator data structure includes: a name, a list of allowable settings, a function which changes the setting, and the benefits and costs of each setting. These costs and benefits are listed as multipliers over a nominal setting, whose costs and benefits are unity. In addition, each actuator specifies the axis for its costs and benefits as one of PERFORMANCE, POWER, or PRECISION. Each actuator specifies a *delay*, or the time between when it is set and when its effects can be observed. Finally, each actuator specifies whether it works on only the application that registered it or if it works on all applications. This last feature allows applications to register application-specific actuators with the control panel.

A systems developer writes a program to register an actuator. This program first calls ACT_attach_control_panel to connect to the SEEC control panel. It then calls the ACT_register_actuator function providing both the name of a text file and

Table 3.3: SEEC System Programmer Interface Listing

Function Name	Arguments	Description
ACT_attach_control_panel		Gets a handle to the system control panel
ACT_detach_control_panel		Releases handle to the control panel
ACT_register_actuator	name [string], file [string]	Registers new actuator with properties specified in the file
ACT_delete_actuator	name [string]	Removes the named actuator from the control panel
ACT_get_nactuators		Returns the number of actuators registered to the control panel
ACT_get_actuators		Returns an array with all actuators registered to the control panel

a name for the actuator. The text file simply has an enumeration of the attributes of the actuator (settings, costs and benefits, delay). Specifying these values in a text file aids portability as the same program can be used on different systems by changing the file. For example, the same program can register a DVFS actuator on machines with different clock speeds by simply changing the file. If the systems developer wants to disable an actuator, the ACT_delete_actuator function will remove it from the control panel. Two query functions are used (primarily by the SEEC runtime) to get information about the number of actuators and the different actuators available. These functions allow multiple systems developers to register actuators independently. The costs and benefits for actuators only serve as initial estimates and the SEEC runtime uses adaptive actuator selection to recover if there are errors in the values specified by the systems developer. However, SEEC allows these models to be specified to provide maximum responsiveness in the case where the models are accurate.

By convention, the actuator setting with identifier 0 is considered to be the one with a benefit of 1 and a cost of 1; the benefits and costs of additional actions are specified as multipliers. Additionally, the systems developer specifies whether

an action can affect all applications or a single application; in the case of a single application, the developer indicates the process identifier of the affected application. Finally, for each action the systems developer indicates a list (possibly empty) of conflicting actions. Conflicting actions represent subsets of actions which cannot be taken at the same time; e.g.,allocation of both five and four cores in an 8-core system.

For example, to specify the allocation of cores to a process in an 8-core system, the developer indicates an actuator with eight settings ($i \in \{0, ..., 7\}$) and provides a function that takes a process identifier and action identifier i binding the process to $i+1$ cores. The systems developer provides an estimate of the increase in performance and power consumption associated with each i. For the core allocator, the speedup of action i might be $i+1$, i.e.,linear speedup, while the increase in power consumption will be found by profiling the target architecture. For each action i, the list of conflicting actions includes all j such that $j + 1 + i + 1 > 8$. Finally, the core allocator will indicate that it can affect any application. In contrast, application-level adaptations indicate that they only affect the given application.

SEEC combines n sets of actuator settings $A^0, ..., A^{n-1}$ defined by (possibly) different developers using the following procedure. First, SEEC creates a new actuator where each setting is defined by the n-tuple $< a_i^0, a_j^1, ..., a_k^{n-1} >$, and corresponds to taking the ith setting from set A^0, the jth setting from set A^1, etc. The benefit of each new set is computed as $s_{<a_i^0,...,a_k^{n-1}>} = s_{a_i^0} \times ... \times s_{a_k^{n-1}}$ and the cost is computed similarly. SEEC may need to combine some actions that affect a single application with others that can affect all applications. If so, SEEC computes and maintains a separate set of actions for each application.

The models only serve as initial estimates and the SEEC runtime system can adapt to even large errors in the values specified by the systems developer. However, SEEC allows these models to be specified to provide maximum responsiveness in the case where the models are accurate. SEEC's runtime adjustment to errors in the models is handled by the different adaptation levels and is described in greater detail in the next section.

Table 3.4: Adaptation in SEEC Decision Engine.

Adaptation Level	Benefits	Drawbacks
Classical Control System	Commonly used, relatively simple	Does not generalize to unseen applications and unreliable system models
Adaptive Control System	Tailors decisions to specific application and input	Assumes reasonable system model
Adaptive Actuator Selection	Supports both race-to-idle and proportional allocation	May over-provision resources if system models are inaccurate
Machine Learning	Learns system models online	Requires time to learn, guarantees performance only in limit

3.3 Decide

The SEEC runtime automatically and dynamically sets actuators to meet goals accurately and efficiently. The SEEC decision engine is designed to handle general-purpose environments and the SEEC runtime system will often have to make decisions about actions and applications with which it has no prior experience. In addition, the runtime system will need to react quickly to changes in application load and fluctuations in available resources. To meet these requirements for handling general and volatile environments, the SEEC decision engine is designed with multiple layers of adaptation, each of which is discussed below.

SEEC uses one set of equations to control performance and another to control power; however, these approaches are extremely similar. Therefore, Sections 3.3.2–3.3.3 provide an in-depth explanation of the system which controls performance. Section 3.3.5 and Section 3.3.6 succinctly explain how to modify performance control for a power or precision goal.

3.3.1 Classical Control System

In its most basic form, the SEEC runtime system implements a basic, model-based feedback control system [36], which complements and generalizes the control system

described in [62]. The controller reads the performance goal g_i for application i, collects the heart rate $h_i(t)$ of application i at time t, computes a speedup $s_i(t)$ to apply to application i at time t, and then translates that speedup into a set of actions based on the model provided by the systems programmer. SEEC uses a generic second order control system which can be customized for a specific system by fine-tuning the tradeoff between responsiveness and rejection of noise.

SEEC's controller observes the heartbeat data of all applications and assumes the heart rate $h_i(t)$ of application i at time t is

$$h_i(t) = \frac{s_i(t-1)}{w_i} + \delta h_i \tag{3.1}$$

Where $w_i(t)$ is the *workload* of application i. Workload is defined as the expected time between two subsequent heartbeats when the system is in the state that provides the lowest speedup, i.e.,when the system takes action 0. In the classical control formulation, SEEC assumes that the workload is not time variant and any noise or variation in the system is modeled with the term δh_i, representing an exogenous disturbance in the measurement of the heartbeat data for application i.

SEEC's goal is to eliminate the *error* $e_i(t)$ between the heart rate goal g_i and the observed heart rate $h_i(t)$ where:

$$e_i(t) = g_i - h_i(t) \tag{3.2}$$

SEEC reduces $e_i(t)$ by controlling the speedup $s_i(t)$ applied to application i at time t. SEEC employs a generic second order transfer function so users can customize the transient behavior of the closed loop system shown in Figure 3-1. Since SEEC employs a discrete time system, we follow standard practice [57, p17] and analyze its transient behavior in the Z-domain:

$$F_i(z) = \frac{(1-p_1)(1-p_2)}{1-z_1} \frac{z-z_1}{(z-p_1)(z-p_2)} \tag{3.3}$$

where $F_i(z)$ is the Z-transform of the closed-loop transfer function for application i and $\{z_1, p_1, p_2\}$ is a set of customizable parameters which alter the transient behavior of the system. The gain of this function is 1, so $e_i(t)$ is guaranteed to reach 0 for all applications. From Equation 3.3, the generic SEEC controller is synthesized following a classical control procedure [57, p281] and SEEC calculates $s_i(t)$ as:

$$\begin{aligned} s_i(t) &= F \cdot [\, A\, s_i(t-1) + B\, s_i(t-2) + \\ &\qquad C\, e_i(t)\, w_i + D\, e_i(t-1)\, w_i\,] \\ A &= p_1 z_1 + p_2 z_1 - p_1 p_2 - 1 \\ B &= -p_2 z_1 - p_1 z_1 + z_1 + p_1 p_2 \\ C &= p_2 - p_1 p_2 + p_1 - 1 \\ D &= (p_1 p_2 - p_2 - p_1 + 1) \cdot z_1 \\ F &= (z_1 - 1)^{-1} \end{aligned} \tag{3.4}$$

To customize the generic controller for specific behavior, the values $\{z_1, p_1, p_2\}$ must be fixed. For stability, SEEC requires $|p_1|, |p_2| < 1$. Setting $z_1 = z_2 = p_1 = 0$ produces a *pure delay* controller which eliminates transient behavior[2] allowing the system to reach $e_i(t) = 0$ as quickly as possible; however, this formulation is sensitive to noise and changes in δh_i will result in commensurate changes in the applied speedup (see Equation 3.1). If $p_1 \leq z_1 \leq p_2$, the controller becomes a *slow convergence* controller which increases the time the system takes to reach g_i. As z_1 approaches p_1, the system will converge more slowly, but will reject larger disturbances in the δh_i term; i.e.,in noisier systems z_1 should be closer to p_1. SEEC can also support an *oscillating* controller. To achieve this, suppose without loss of generality, $p_2 \geq p_1$. If at least one of these values is negative, the system will oscillate around $\bar{r}$. If $p_1 \leq z_1 \leq p_2$, the system will slowly converge to $\bar{r}$. The closer z_1 is to p_2, the faster the system will reach $\bar{r}$. If $z_1 \geq p_1$ the system is subject to overshoot $\bar{r}$ and if $z_1 \geq 1$ the system is subject to undershoot. $p_1 = -\varepsilon$, $p_2 = z_1 = 0$ produces oscillating behavior that allows the system to reach the steady state quickly, while if $p_1 = -1+\varepsilon$ the oscillating

[2] In a control-theoretic sense transient behavior cannot be fully eliminated, but this formulation makes the transient period as small as possible.

behavior slowly converges to the desired value.

3.3.2 Adaptive Control

Unlike the classical control system, the adaptive control system estimates application workload online turning the constant w from Equation 3.4 into a per-application, time varying value. This change allows SEEC to rapidly respond to previously unseen applications and sudden changes in application performance. The true workload cannot be measured online as it requires running the application with all possible actions set to provide a speedup of 1, which will likely fail to meet the application's goals. Therefore, SEEC views the true workload as a hidden state and estimates it using a one dimensional Kalman filter [92].

SEEC represents the true workload for application i at time t as $w_i(t) \in \mathbb{R}$ and models this workload as:

$$\begin{aligned} w_i(t) &= w_i(t-1) + \delta w_i(t) \\ h_i(t) &= \frac{s_i(t-1)}{w_i(t-1)} + \delta h_i(t) \end{aligned} \tag{3.5}$$

where $\delta w_i(t)$ and $\delta h_i(t)$ represent time varying noise in the true workload and heart rate measurement, respectively. SEEC recursively estimates the workload for application i at time t as $\hat{w}_i(t)$ using the following Kalman filter formulation:

$$\begin{aligned} \hat{x}_i^-(t) &= \hat{x}_i(t-1) \\ p_i^-(t) &= p_i(t-1) + q_i(t) \\ k_i(t) &= \frac{p_i^-(t)s_i(t-1)}{[s_i(t)]^2 p_i^-(t) + o_i} \\ \hat{x}_i(t) &= \hat{x}_i^-(t) + k_i(t)[h_i(t) - s_i(t-1)\hat{x}_i^-(t)] \\ p_i(t) &= [1 - k_i(t)s_i(t-1)]p_i^-(t) \\ \hat{w}_i(t) &= \frac{1}{\hat{x}_i(t)} \end{aligned} \tag{3.6}$$

Where $q_i(t)$ and o_i represent the application variance and measurement variance,

respectively. The application variance $q_i(t)$ is the variance in the heart rate signal since the last filter update. SEEC assumes that o_i is a small fixed value as heartbeats have been shown to be a low-noise measurement technique [37]. $h_i(t)$ is the measured heart rate for application i at time t and $s_i(t)$ is the applied speedup (according to Equation 3.4). $\hat{x}_i(t)$ and $\hat{x}_i(t)^-$ represent the *a posteriori*and *a priori*estimate of the inverse of application i's workload at time t. $p_i(t)$ and $p_i^-(t)$ represent the *a posteriori*and *a priori*estimate error variance, respectively. $k_i(t)$ is the *Kalman gain* for the application i at time t.

SEEC's runtime improves on the classical control formulation by replacing the fixed value of w from Equations 3.1 and 3.4 with the estimated value of $\hat{w}_i(t)$. By automatically adapting workload on the fly, SEEC can control different applications without having to profile and model the applications ahead of time. Additionally, this flexibility allows SEEC to rapidly respond to changes in application behavior. In contrast, the classic control model presented in the previous section must use a single value of w for all controlled applications which greatly limits its efficacy in a general computing environment.

Note, $w_i(t)$ and $s_i(t)$ have an inverse relationship in Equation 3.1. Therefore, Equation 3.6 allows SEEC to respond to changes in both application behavior and system resources, as any error in the speedup models will be perceived (using just the adaptive controller) as an error in workload and compensated accordingly. For example, suppose the actions available to SEEC include allocation of cores. Further, suppose an application i is meeting its goals with four cores until the clock speed of these cores is lowered. The change in compute power will change the heart rate $h_i(t)$ at time t, which will, in turn, immediately affect the workload estimate $\hat{w}_i(t)$ (Equation 3.6) and cause a corresponding change in the applied speedup $s_i(t)$ (Equation 3.4). This change in speedup will result in the allocation of additional cores. If the temperature cools and all cores are restored, the workload estimator will return lower values of $w_i(t)$ and SEEC will reduce the allocated cores.

3.3.3 Adaptive Actuator Selection

SEEC's adaptive control system produces a continuous speedup signal $s_i(t)$ which the runtime must translate into a set of actions. SEEC does this by scheduling actions over a time window of τ heartbeats. Given a set $A = \{a\}$ of actions with speedups s_a and costs c_a, SEEC would like to schedule each action for $\tau_a \leq \tau$ time units in such a way that the desired speedup is met and the total cost of all actions is minimized. In other words, SEEC tries to solve the following optimization problem:

$$\begin{aligned} \text{minimize}(\tau_{idle} c_{idle} + \tfrac{1}{\tau}\textstyle\sum_{a\in A}(\tau_a c_a)) \quad \text{s. t.} & \\ \tfrac{1}{\tau}\textstyle\sum_{a\in A}\tau_a s_a &= s_i(t) \\ \tau_{idle} + \textstyle\sum_{a\in A}\tau_a &= \tau \\ \tau_a, \tau_{idle} &\geq 0, \ \forall a \end{aligned} \tag{3.7}$$

Note the *idle* action, which idles the system paying a cost of c_{idle} and achieving no speedup. It is impractical to solve this system online, so SEEC instead considers three candidate solutions: race-to-idle, proportional allocation, and a hybrid approach.

First, SEEC considers *race-to-idle*, i.e.,taking the action that achieves maximum speedup for a short duration hoping to idle the system for as long as possible. Assuming that $max \in A$ such that $s_{max} \geq s_a \forall a \in A$, then racing to idle is equivalent to setting $\tau_{max} = \frac{s_i(t)\cdot\tau}{s_{max}}$ and $\tau_{idle} = \tau - \tau_{max}$. The cost of doing so is then equivalent to $c_{race} = \tau_{max} \cdot c_{max} + \tau_{idle} \cdot c_{idle}$.

SEEC then considers *proportional* scheduling. SEEC selects from actions which are Pareto-optimal in terms of speedup and cost to find an action j with the smallest speedup s_j such that $s_j \geq s_i(t)$ and an action k such that $s_k < s_j$. The focus on Pareto-optimal actions ensures j is the lowest cost action whose speedup exceeds the target. Given these two actions, SEEC takes action j for τ_j time units and k for τ_k time units where $s_i(t) = \tau_j \cdot s_j + \tau_k \cdot s_k$ and $\tau = \tau_j + \tau_k$. The cost of this solution is $c_{prop} = \tau_j \cdot c_j + \tau_k \cdot c_k$.

The third solution SEEC considers is a *hybrid*, where SEEC finds an action j as in the proportional approach. Again, s_j is the smallest speedup such that $s_j \geq s_i(t)$;

however, SEEC considers only action j and the idle action, so $s_i(t) = \tau_j \cdot s_j + \tau_{idle} \cdot s_{idle}$, $\tau = \tau_j + \tau_{idle}$, and $c_{hybrid} = \tau_j \cdot c_j + \tau_{idle} \cdot c_{idle}$.

In practice, the SEEC runtime system solves Equation 3.7 by finding the minimum of c_{race}, c_{prop}, and c_{hybrid} and using the set of actions corresponding to this minimum cost.

3.3.4 Reinforcement Learning

The use of adaptive control and adaptive actuator selection augments a classical control system with the capability to adjust its behavior dynamically and control even previously unseen applications. Even with this flexibility, however, the control system can behave sub-optimally if the costs and benefits of the actions as supplied by the application programmer are incorrect or inconsistent across applications. For example, suppose a systems programmer specifies a set of actions which change processor frequency. Furthermore, the systems programmer specifies that the applications speedup linearly with a linear increase in frequency. This model will work well for compute-bound applications, but the control solutions described so far may allocate too much frequency for I/O bound applications.

To overcome this limitation, SEEC augments its adaptive control system with machine learning. At each time-step, SEEC computes a speedup according to Equation 3.4 using the workload estimate from Equation 3.6 and uses reinforcement learning (RL) to determine an action that will achieve this speedup with lowest cost. Specifically, SEEC uses temporal difference learning to determine the expected utility $Q_a, a \in A$ of the available actions[3]. Q_a is initialized to be s_a/c_a; if the developer's estimates are accurate, the learner will converge more quickly.

Each time the learner selects an action, it receives a reward $r(t) = h(t)/cost(t)$ where $h(t)$ is the measured heart rate and $cost(t)$ is the measured cost of taking the action a for τ_a time units and idling for the remaining $\tau_{idle} = \tau - \tau_a$ time units. Given

[3]SEEC learns the Q functions on a per application basis, but to enhance readability in this section we drop the i subscript denoting application i.

the reward signal, SEEC updates its estimate of the utility function Q_a by calculating:

$$\hat{Q}_a(t) = \hat{Q}_a(t-1) + \alpha(r(t) - \hat{Q}_a(t-1)) \tag{3.8}$$

Where α is the *learning rate* and $0 < \alpha \leq 1$[4]. In addition, SEEC keeps estimates of s_a and c_a calculated as

$$\begin{aligned} \hat{h}_a(t) &= \hat{h}_a(t-1) + \alpha(h(t) - \hat{h}_a(t-1)) \\ \hat{s}_a(t) &= \frac{\hat{h}_a(t)}{\hat{h}_0(t)} \\ \hat{c}_a(t) &= \hat{c}_a(t-1) + \alpha(cost(t) - \hat{c}_a(t-1)) \end{aligned} \tag{3.9}$$

Given a desired speedup $s(t)$ and the current estimate of utility $\hat{Q}_a(t) \forall a \in A$, SEEC updates its estimates of speedups and costs according to Equation 3.8 and then selects an action using Algorithm 1. The selection algorithm uses Value-Difference Based Exploration (VDBE) [90] to balance exploration and exploitation. As shown in the algorithm listing, SEEC keeps track of a parameter, ϵ (where $0 \leq \epsilon \leq 1$) that is used to balance the tradeoff between exploration and exploitation. When selecting an action to meet the desired speedup, a random number r (where $0 \leq r < 1$) is generated. If $r < \epsilon$, the algorithm randomly selects an action. Otherwise, the algorithm selects the lowest cost action that meets the desired speedup. The value of ϵ is updated every time the algorithm is called. A large difference between the reward $r(t)$ and the utility estimate $\hat{Q}_a(t)$ results in a large ϵ, while a small difference makes ϵ small. Thus, when SEEC's estimates of the true speedups and costs are wrong, the algorithm explores available actions. As the estimates converge to the true values, the algorithm exploits the best solution found so far. In other words, as the model converges, the ML system behaves like the adaptive actuator selector from section Section 3.3.3.

Having selected and action a', SEEC executes that action and waits until τ heartbeats have been completed (and SEEC idles itself during this time). If the heartbeats complete sooner than desired for the given value of s, then $\hat{s}_{a'}$ was larger than nec-

[4]In our system the learning rate is set to 0.85 for all experiments.

Algorithm 1 Select an action to meet the desired speedup.

Inputs:

s - a desired speedup

$\hat{Q}$ - estimated utility for available actions

$r(t)$ - the reward at time t

α - the learning rate parameter

A - the set of available actions

$a \in A$ - the last action selected

ϵ - parameter that governs exploitation vs. exploration

Outputs:

$next$ - the next action to be taken

ϵ - an updated value

$x = e^{\frac{-|\alpha(r(t)-\hat{Q}_a(t)|}{\sigma}}$

$f = \frac{1-x}{1+x}$

$\delta = \frac{1}{|A|}$

$\epsilon = \delta \cdot f + (1 - \delta) \cdot \epsilon$

$r =$ a random number drawn with uniform distribution from 0 to 1

if $r < \epsilon$ **then**

randomly select a' from A using a uniform distribution

else

find $A' = \{b | b \in A, \quad \hat{s}_b \geq s\}$

select $a' \in A'$ s.t. $\hat{Q}_{a'}(t) \geq \hat{Q}_b, \forall b \in A'$

end if

return a' and ϵ

essary, so SEEC idles the system for the extra time. If $\hat{s}_{a'}$ was too small, then SEEC does not idle. In either case, the cost and reward are immediately computed using Equation 3.8 and a new action is selected using Algorithm 1. We note that by idling the system if the selected action was too large, SEEC can learn to correctly race-to-idle even when the system models are incorrect.

3.3.5 Managing Power

We modify the equations from Sections 3.3.1–3.3.4 to control power while providing efficient performance. SEEC models the power consumption as:

$$power(t+1) = b \cdot c(t) \tag{3.10}$$

Where $power(t)$ is the power consumption at time t, b is the *base* power consumption (defined as power consumption when the system is active, but using minimal resources), and $c(t)$ is a coefficient representing the cost of additional power consumption at time t.

Given Equation 3.10, SEEC eliminates the error $e_{pow}(t) = g_{pow} - p(t)$ between the power consumption goal g_{pow} and $p(t)$, the measured power at time t. When controlling power, error is reduced by modifying the current cost $c(t)$, and we again analyze transient behavior in the Z-domain:

$$G_i(z) = z \tag{3.11}$$

where $G_i(z)$ is the Z-transform of the closed-loop transfer function that controls power consumption. As is the case for Equation 3.3, we see that Equation 3.11 has a gain of unity when $z = 1$ so the system is accurate when controlling power. From Equation 3.11, the power controller is synthesized as:

$$c(t) = c(t-1) + \frac{e_{pow}(t)}{b} \tag{3.12}$$

where $e(t)$ is the error at time t and $c(t)$ is the cost (in additional power consumption)

to incur at time t.

The coefficient b is analogous to w_i in Equation 3.4 in that both are constants which affect the gain of the controller and thus have tremendous effect on the accuracy and efficiency of control. So, just as we estimate w_i using Equation 3.6, we denote our estimate of the true value of b as $\hat{b}$ and calculate it using a Kalman Filter formulation:

$$\begin{aligned}
\hat{b}^-(t) &= \hat{b}(t-1) \\
p_b^-(t) &= p_b(t-1) + q_b(t) \\
k_b(t) &= \frac{p_b^-(t)c(t-1)}{[c(t)]^2 p_b^-(t) + o_b} \\
\hat{b}(t) &= \hat{b}^-(t) + k_b(t)[power(t) - c(t-1)\hat{b}^-(t)] \\
p_b(t) &= [1 - k_b(t)c(t-1)]p_b^-(t)
\end{aligned} \tag{3.13}$$

Where $q_b(t)$ and o_b represent the system power variance and power measurement variance, respectively. Similar to Equation 3.6, $q_b(t)$ is the variance in the power signal since the last filter update. In this case, o_b represents the noise in the power measurement device, in this case a WattsUp power meter. We set this value to be 2, as we never measured a standard deviation in power of more than 1.4 for a system running a constant workload. $power(t)$ is the measured power consumption at time t (Equation 3.10 and $c(t)$ is the additional cost (Equation 3.4) applied at time t. $\hat{b}(t)$ and $\hat{b}(t)^-$ represent the *a posteriori*and *a priori*estimate of the base power consumption. Again, $p_b(t)$ and $p_b^-(t)$ represent the *a posteriori*and *a priori*estimate error variance. $k_b(t)$ is the *Kalman gain* for the system power consumption at time t.

When controlling power, SEEC first updates its estimate of the base power using Equation 3.13 and then substitutes this new value of $\hat{b}(t)$ in place of the fixed value b in Equation 3.12. While base power does not vary as much as workload (which is entirely application dependent), estimation is beneficial because different applications have different power characteristics based on their instruction mixes, cache uses, etc.

As was the case for controlling performance, the continuous control signal $c(t)$ needs to be translated into actuator settings. In this case, SEEC meets the power goal while trying to maximizing the speedup provided to the application, so the

relevant optimization problem is:

$$\begin{aligned} \text{maximize}(\tau_{idle} c_{idle} + \tfrac{1}{\tau}\textstyle\sum_{a\in A}(\tau_a s_a)) \quad & \text{s. t.} \\ \tfrac{1}{\tau}\textstyle\sum_{a\in A}\tau_a c_a \leq \quad & c(t) \\ \tau_{idle} + \textstyle\sum_{a\in A}\tau_a = \quad & \tau \\ \tau_a, \tau_{idle} \geq \quad & 0, \qquad \forall a \end{aligned} \tag{3.14}$$

As before, SEEC considers the solution at three points and, in this case, takes the one that provides the maximum speedup. As was the case when controlling performance, SEEC continuously updates estimates of key metrics using Equation 3.9.

3.3.6 Managing Precision

Managing precision is done in entirely the same manner as power from the previous section. In fact, the same equations are used, it is only the interpretation of the equations that changes. Now, the system is controlling precision, so $c(t)$ is interpreted as the additional precision to apply at time t. Using classical control, it is again assumed that b is time invariant, and, in this case, represents the base precision.

As precision is entirely application dependent, it is very important to use adaptive control and estimate the value of b as $\hat{b}(t)$. Once again the same Kalman filter formulation can be used. To make it work for precision, the only change required is to measure the noise in the precision metric.

When controlling precision, SEEC can maximize performance or minimize power consumption. The same strategies used to solve Equations 3.7 and 3.14 apply in this case.

3.3.7 Changing goals dynamically

SEEC allows applications to change from performance to power to precision goals dynamically. This may be useful if, for example, a user switches from wall power to battery power and the primary concern switches from performance to power. Switching goals will cause a momentary loss of accuracy as SEEC switches from one goal to

another. To minimize this period of instability, all state is maintained for all three control systems regardless of whether the system is currently controlling power, precision, or performance. Specifically, the system always maintains a current value of $s_i(t)$ and $c(t)$ and always updates its estimates of workload (Equation 3.6) and base power (Equation 3.13) regardless of whether it is controlling performance or power. Maintaining this state allows SEEC to rapidly transition from managing performance to power (or power to performance) and still compute the correct value for $s_i(t+1)$, which depends on $s_i(t)$ (or $c(t+1)$, which depends on $c(t)$).

3.3.8 Multiple Applications

When working with multiple applications, the control system may request speedups whose realization results in resource conflicts (e.g.,in an 8-core system, the assignment of 5 cores to one application and 4 to another). SEEC resolves conflicting actions using a priority scheme. Higher priority applications get first choice amongst any set of actions which govern finite resources. Once actions are scheduled for a high priority application, those actions are removed from consideration for lower priority applications. In the example, the higher priority application would be assigned 5 cores with the other forced to use three and find speedup from an additional source if available.

If applications have the same priority, SEEC resolves conflicts using a *centroid technique* [61]. Suppose the total amount of a resource is n and this resource must be split between m applications. SEEC defines an m-dimensional space and considers the sub-space whose convex hull is defined by the combination of the origin and the m points $(n, 0, \ldots, 0)$, $(0, n, \ldots, 0)$, $\ldots$, $(0, 0, \ldots, n)$. The desired resource allocation is represented by the point $p = (n_1, n_2, \ldots, n_m)$, where the ith component of p is the amount of resource needed by application i. If p is outside the convex hull, SEEC then identifies the *centroid* point $\frac{1}{m}(1, 1, \ldots, 1)$ and the line l intersecting both the centroid and p. SEEC computes the point p' where l intersects with the convex hull and then allocates resource such that the ith component of point p is the amount of resource allocated to application i. This method balances the needs of multiple

applications when resources are oversubscribed.

3.4 Discussion

SEEC's decoupled approach has several benefits. First, application programmers focus on application-level goals and feedback without having to understand the system-level adaptations available. Similarly, systems developers can specify available adaptations without knowing how to monitor an application. Both application and systems developers can rely on SEEC's general and extensible decision engine to coordinate the adaptations specified by the application developer and one or more systems developers. Thus, the decoupled approach makes it easier to develop adaptive systems.

Each of the adaptation levels in SEEC's runtime decision mechanism (Figure 3-1) builds on adaptations from the previous level and each has its tradeoffs, summarized in Table 3.4. In practice, we find that it is best to run SEEC using either adaptive action selection or machine learning and each has different uses. When the systems developer is very confident that the systems models (costs and benefits of actions) are accurate and they do not contain local minima or maxima, then SEEC will work best using adaptive actuator selection. In this case, SEEC can adapt to differing applications quickly and adjust the resource allocation appropriately without machine learning. In contrast, if the system will be running a mix of applications with varying responses to actions, then it is unlikely that the models provided by the developer will be accurate for all applications. In this case, SEEC's machine learning engine can keep the control system from over-provisioning resources for little added gain. Experiments demonstrating these tradeoffs are described in Chapter 5.

SEEC can support two types of application goals. If an application requests a performance that is less than the maximum achievable on a system (e.g.,a video encoder working with live video), SEEC will minimize the cost of achieving that goal. When an application simply wants maximum performance, it can set its goal to be a huge number. SEEC will attempt to meet that number, but it will do so while minimizing costs. For example, if an application is memory bound and requests

infinite performance, SEEC can allocate maximum memory resources, but also learn not to allocate too many compute resources.

SEEC is designed to be general and extensible and SEEC can work with applications that it has not previously encountered and for which its provided models are wrong. To support this generality, SEEC has mechanisms allowing it to learn both application and systems models online. One limitation of this approach is that SEEC needs enough feedback from the application to have time to adapt. Thus, SEEC is appropriate for supporting either relatively long-lived applications or short-lived applications that will be repeatedly exercised. In our test scenarios, all applications emitted between 200 and 60000 heartbeats. SEEC is not designed to support short lived applications that are executed only a small number of times.

Chapter 4

Using SEEC

This section describes the applications and system components used to evaluate the SEEC model and runtime.

4.1 Benchmarks

We use the PARSEC benchmarks [10] to test SEEC's ability to manage a variety of applications. These benchmarks represent a mix of important, emerging multicore workloads and we modify them to emit heartbeats as described in [37]. In general, these benchmarks have some outer loop (in freqmine control is governed by a recursive function call) and this is where the heartbeats are inserted. Table 4.1 shows where the heartbeat is inserted in terms of the application's processing and the average heart rate that the benchmark achieved over the course of its execution running the "native" input data set on the eight-core x86 test platform.

Use of this suite tests SEEC's ability to handle a wide range of applications with different heart rate characteristics. To illustrate this range, the variance in heart rate for each of the PARSEC benchmarks is shown in Table 4.2. This data is gathered by running each benchmark on an eight core processor, measuring the reported heart rate at each heartbeat, and computing the variance in this heart rate signal. Benchmarks with regular performance have low variance, while benchmarks with irregular performance (some iterations much harder/easier than others) will have high vari-

Table 4.1: Heartbeats in PARSEC Benchmarks

Benchmark	Heartbeat Location	Heart Rate (beat/s)
`blackscholes`	Every 25000 options	561.03
`bodytrack`	Every frame	4.31
`canneal`	Every 1875 moves	1043.76
`dedup`	Every "chunk"	264.30
`facesim`	Every frame	0.72
`ferret`	Every query	40.78
`fluidanimate`	Every frame	41.25
`freqmine`	Every recursive function call	7.18
`raytrace`	Every frame	3.49
`streamcluster`	Every 200000 points	0.02
`swaptions`	Every "swaption"	2.27
`vips`	Every task completion	34.37
`x264`	Every frame	11.32

Table 4.2: Variance in heart rate for PARSEC.

Benchmark	Variance	Benchmark	Variance
blackscholes	1.90E-01	raytrace	9.55E-02
bodytrack	2.32E-01	streamcluster	7.41E-03
canneal	2.40E+09	**swaptions**	9.23E+07
dedup	1.10E+10	**vips**	4.93E+09
facesim	3.51E-03	x264	4.94E+02
ferret	2.27E+07	STREAM	1.93E-01
fluidanimate	1.29E-01	dijkstra	2.50E+01
freqmine	1.17E+09		

ance. We note that 6 of the 13 PARSECs (bold in the table) have high variance. To manage these benchmarks, SEEC will adapt its internal models to the characteristics of each application including phases and variance within a single application.

Adding heartbeats to the PARSEC benchmark suite is easy, even when unfamiliar with the benchmark implementations. The PARSEC documentation describes the inputs for each benchmark. With that information it is simple to find the key loops over the input data set and insert the call to register a heartbeat in this loop. The amount of code required to add heartbeats to each of the benchmarks is under a dozen lines for each application. The extra code is simply the inclusion of the header file and declaration of a Heartbeat data structure, calls to initialize and finalize the Heartbeats run-time system, and the call to register each heartbeat.

Unlike the PARSEC benchmarks, the STREAM benchmark does not scale well

with increasing compute resources [64]. STREAM is designed to exercise a processor's memory hierarchy and it is a classic example of a memory-bound benchmark; however, it only becomes memory bound once it has enough compute resources to saturate the memory controllers. To control STREAM optimally, SEEC will have to find the balance between compute and memory resources. STREAM has an outer loop which executes a number of smaller loops that operate on arrays too large to fit in cache. We instrument STREAM to emit a heartbeat every outer loop. STREAM tests SEEC's ability to adjust its models online and learn how to manage a memory-bound benchmark.

The dijkstra benchmark was developed for this thesis specifically to test the SEEC system. dijkstra is a parallel implementation of Dijkstra's single source shortest paths algorithm processing a large, dense graph. The benchmark demonstrates limited scalability, achieving modest speedup with small numbers of processors, but reduced performance with large numbers of processors. The scaling for this benchmark is limited by communication overhead as each iteration of the algorithm must select from and update a priority queue. We instrument this application to emit a heartbeat every time a new vertex is selected from the queue. dijkstra tests SEEC's ability to adjust its models online and learn not to over-provision resources for a benchmark that cannot make use of them.

Using the Heartbeats interface can provide additional insight into the performance of these benchmarks beyond that provided by just measuring execution time. For example, Figure 4-1 shows a moving average of heart rate for the `x264` benchmark using a 20 beat window (a heartbeat is registered as each frame is processed). The chart shows that `x264` has several distinct regions of performance when run on the PARSEC native input. The first occurs in the first 100 frames where the heart rate tends to the range of 12-14 beats per second. Then, between frames 100 and 330 the heart rate jumps to the range of 23-29 beats per second. Finally, the heart rate settles back down to its original range of 12-14 beats per second. In this example, the use of Heartbeats shows distinct regions of performance for the `x264` benchmark with the native input size. This information can be useful for understanding the performance

of certain benchmarks and optimizing these benchmarks on a given architecture. Such regions would be especially important to detect in an adaptive system.

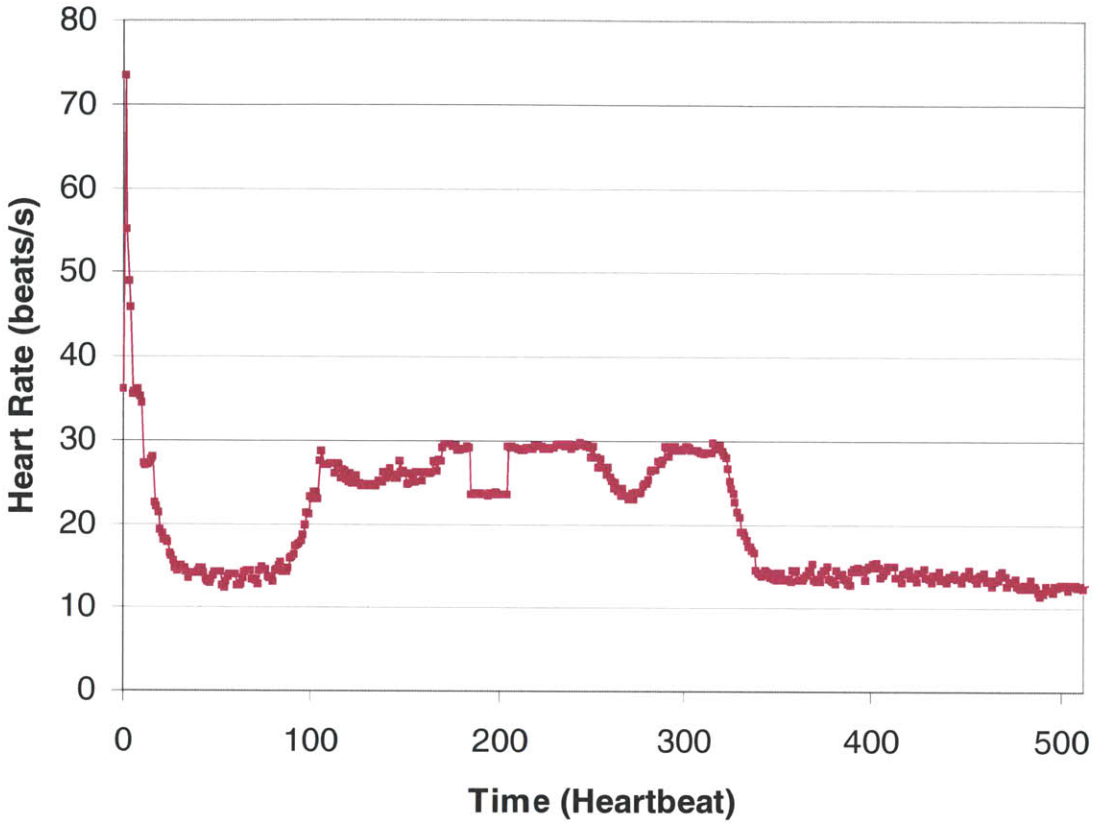


Figure 4-1: Heart rate of the x264 PARSEC benchmark executing native input on an 8-core x86 server.

In summary, the Heartbeats framework is easy to insert into a broad array of applications and our reference implementations are low-overhead for the variety of different computations represented by the PARSEC benchmarks. The next section provides an example of using the Heartbeats framework to develop an adaptive application.

4.2 Adaptations

SEEC can be used to manage both system- and application-level adaptations. This section describes how both have been implemented using SEEC.

Table 4.3: Hardware platforms used in evaluation.

Name	Proc.	No. Cores	No. Mem. Conts.	Speeds (GHz)	No. Speeds	Turbo-Boost	Max Pow. (Watts)	Idle Pow. (Watts)
Machine 1	Intel Xeon X5460	8	1	2.000–3.160	4	no	329	200
Machine 2	Intel Xeon E5520	8	2	1.596–2.395	8	yes	220	90

4.2.1 System-Level Adaptations

To demonstrate the generality of SEEC, we test it on two different machines whose characteristics are summarized in Table 4.3. Both are equipped with Watts Up? power meters [2], which measure average power consumption over an interval, the smallest supported being 1 second. We use these devices to measure power consumption on a per heartbeat basis. If heartbeat signals come less than a second apart, we interpolate power, otherwise the power measurement is returned directly.

Both machines have similar compute capacities but there are distinct sets of components available on each. Machine 1 supports three actuators which allow SEEC to 1) idle an application (by descheduling it), 2) assign cores to an application (through affinity), and 3) change the clock speed of the cores assigned to an application (using `cpufrequtils`). Machine 2 supports the first three actuators and two additional actuators which change the assignment of memory controllers to an application (through numa mappings) and turn the hardware's ability to use TurboBoost on and off. In addition, machine 2 has four extra clock speeds available compared to machine 1. The available actuators are summarized in Table 4.4. Although, these machines look similar, SEEC uses different strategies to manage their performance/power tradeoffs as we will see in the next section.

Table 4.4: Summary of Actuators

Machine	No. of Actuators	Actuators
Machine 1	3	Idle time, Core Allocation, Clock Speed
Machine 2	5	Idle time, Core Allocation, Clock Speed, Memory Controllers, TurboBoost

4.2.2 Application-Level Actuators

SEEC can be used to control actions specified at either the application or system level. The previous section discussed some of the system-level actions that SEEC can manage. This section describes a compiler framework that can turn statically configured applications into applications with *dynamic knobs*, i.e.,applications whose dynamic behavior is exposed by SEEC's actuator interface and controlled by the SEEC runtime system [39].

This approach is designed for applications that 1) have static configuration parameters controlling performance versus precision tradeoffs and 2) use the Application Heartbeats API (the compiler can automatically insert the required API calls). These applications typically exhibit the following general computational pattern:

- **Initialization:** During initialization the application parses and processes the configuration parameters, then computes and stores the resulting values in one or more control variables in the address space of the running application.
- **Main Control Loop:** The application executes multiple iterations of a main control loop. At each iteration it emits a heartbeat, reads the next unit of input, processes this unit, produces the corresponding output, then executes the next iteration of the loop. As it processes each input unit, it reads the control variables to determine which algorithm to use.

With this computational pattern, the point in the performance versus precision tradeoff space at which the application executes is determined by the configuration parameters when the application starts and does not change during its execution. Using SEEC, the compiler can augment the application with the ability to dynamically change the point in the tradeoff space at which it is operating. At a high level, this goal is accomplished as follows:

- **Parameter Identification:** The user of the program identifies a set of configuration parameters and a range of settings for each such parameter. Each combination of parameter settings corresponds to a different point in the performance versus precision tradeoff space.
- **Dynamic Knob Identification:** For each combination of parameter settings, the compiler uses dynamic influence tracing (which traces how the parameters influence values in the running application) to locate the control variables and record the values stored in each control variable.
- **Dynamic Knob Calibration:** Given a set of representative inputs and a precision goal, the compiler executes a training run for each input and combination of parameter settings. For each training run it records performance and precision information. It then processes this information to identify the Pareto-optimal points in the explored performance versus precision tradeoff space.
- **Dynamic Knob Insertion:** The compiler inserts calls to SEEC's systems programmer interface. As discussed in Chapter 3 SEEC's runtime uses the information provided through these calls to set the control variables to values previously recorded during dynamic knob identification, thereby moving the application to a different Pareto-optimal point in the performance versus precision trade-off space. Subsequent iterations of the main control loop will read the updated values in the control variables to (in effect) process further input as if the configuration parameters had been set to their corresponding different settings at application startup.

The result is an application that enables SEEC to dynamically control the point in the performance versus precision tradeoff space at which the application executes. In standard usage scenarios the application specifies a target heart rate. If SEEC observes a heart rate slower than the target, it uses the calibrated dynamic knobs to move the application to a new point in the tradeoff space with higher performance at the cost, typically small, of some precision. If the observed heart rate is higher than the target, SEEC moves the application to a new point with lower performance and better precision.

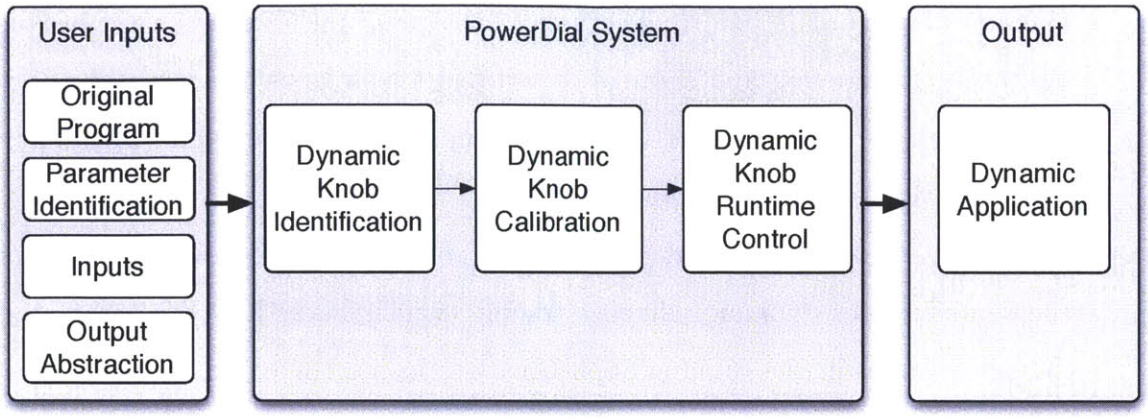


Figure 4-2: Dynamic Knob work flow.

4.2.3 Dynamic Knob Identification

To transform a given set of configuration parameters into something usable by SEEC, the compiler must identify a set of control variables that satisfy the following conditions:

- **Complete and Pure:** All variables whose values are derived from configuration parameters during application startup (before the application emits its first heartbeat) are control variables. The values of control variables are derived only from the given set of configuration parameters and not from other parameters.
- **Relevant and Constant:** During executions of the main control loop, the application reads but does not write the values of the control variables.

This compiler uses influence tracing [16, 29] to find the control variables for the specified configuration parameters. For each combination of configuration parameter settings, the compiler framework executes a version of the application instrumented to trace, as the application executes, how the parameters influence the values that the application computes. It uses the trace information to find the control variables and record their values, applying the above conditions as follows:

- **Complete and Pure Check:** It finds all variables that, before the first heartbeat, contain values influenced by the specified configuration parameters. It checks that these values are influenced only by the specified configuration parameters.

- **Relevance Check:** It filters out any variables that the application does not read after the first heartbeat — the values of these variables are not relevant to the main control loop computation.
- **Constant Check:** It checks that the execution does not write a control variable after the first heartbeat.

Finally, the compiler checks that the control variables are *consistent*, i.e.,that the different combinations of parameter settings all produce the same set of control variables. If the application fails any of these checks, the transformation is rejected.

For each combination of parameter settings, the value of each control variable is recorded. This information is passed to the SEEC runtime system through the systems programmer interface. Note that because this approach uses a dynamic influence analysis to find the control variables, it is possible for unexercised execution paths to violate one or more of the above conditions. The influence analysis also does not trace indirect control-flow or array index influence.

The influence tracing system is implemented as a static, source-based instrumentor for C and C++. It is built on the LLVM compiler framework [16, 55] and inserts code to trace the flow of influence through the values that the application computes. For each value, it computes the configuration parameters that influenced that value. The currently implemented system supports control variables with datatypes of int, long, float, double, or STL vector. It augments the production version of the application with calls to the SEEC system programmer interface to register the address of each control variable and read in the previously recorded values corresponding to the different dynamic knob settings. This mechanism gives the SEEC control system the information it needs to apply a given actuator setting.

4.2.4 Dynamic Knob Calibration

In this step, the compiler explores the performance versus precision trade-off space available to the application via the specified configuration parameters. The user provides an application, a set of representative inputs, a set of specified configuration

parameters, a range of values for each parameter, and a precision metric. Given these values, the compiler produces, for each combination of parameter settings, a specification of the point in the tradeoff space to which the parameter settings take the application. This point is specified relative to the baseline performance and precision of the parameter setting that delivers the highest precision (which, for our set of benchmark applications, is the default parameter setting).

The calibrator executes all combinations of the representative inputs and configuration parameters. For each parameter combination it records the mean (over all representative inputs) speedup of the application. It computes the speedup as the execution time of the application running with the default parameter settings divided by the execution time of the application with the current parameter combination. In a separate instrumented execution, it also records the values of the control variables (see Section 4.2.3).

For each combination of configuration parameters the compiler also records the mean (over all representative inputs) precision. The precision metric works with a user-provided, application-specific *output abstraction* which, when provided with an output from the program, produces a set of numbers $o_1, \ldots, o_m$. The output abstraction typically extracts relevant numbers from the output or computes a measure of output quality (such as, for example, the peak signal-to-noise ratio of the output). Given the output abstraction from the baseline execution $o_1, \ldots, o_m$ and an output abstraction $\hat{o}_1, \ldots, \hat{o}_m$ from the execution with the current parameter settings, we compute the precision loss as the *distortion* [74]:

$$prec = \frac{1}{m} \sum_{i=1}^{m} w_i \left| \frac{o_i - \hat{o}_i}{o_i} \right| \tag{4.1}$$

Here each weight w_i is optionally provided by the user to capture the relative importance of the ith component of the output abstraction. Note that a *prec* of zero indicates optimal precision, with higher numbers corresponding to worse precision. This approach supports caps on precision loss — if a specific parameter setting produces a precision loss that exceeds a user-specified bound, the system can exclude the

corresponding setting from further consideration.

At this point the compiler has created an application that can register goals and actions with SEEC. The SEEC runtime system then manages this application to maximize its precision subject to its stated performance goal.

4.2.5 Using the Compiler

The compiler framework is used to create four adaptive applications. swaptions, bodytrack, and x264 are all taken from the PARSEC benchmark suite as described above; swish++ is as open-source search engine [85]. For each application we acquire a set of representative inputs, then randomly partition the inputs into *training* and *production* sets. We use the training inputs to obtain the dynamic knob response model (see Section 4.2.4) and the production inputs to evaluate the behavior on previously unseen inputs. Table 4.5 summarizes the sources of these inputs. All of the applications support both single- and multi-threaded execution. In our experiments we use whichever mode is appropriate. In this section, each of these benchmarks is described in turn.

Benchmark	Training Inputs	Production Inputs	Source
swaptions	64 swaptions	512 swaptions	PARSEC & randomly generated swaptions
x264	4 HD videos of 200+ frames	12 HD videos of 200+ frames	PARSEC & xiph.org [3]
bodytrack	sequence of 100 frames	sequence of 261 frames	PARSEC & additional input from PARSEC authors
swish++	2000 books	2000 books	Project Gutenberg [1]

Table 4.5: Summary of Training and Production Inputs for Each Benchmark

swaptions

Description: This financial analysis application uses Monte Carlo simulation to solve a partial differential equation that prices a portfolio of swaptions. Both the

accuracy and the execution time increase with the number of simulations — the accuracy approaches an asymptote, while the execution time increases linearly.

Knobs: We use a single command line parameter, `-sm`, as the dynamic knob. This integer parameter controls the number of Monte Carlo simulations for each swaption. The values range from $10,000$ to $1,000,000$ in increments of $10,000$; 1,000,000 is the default value for the PARSEC native input.

Inputs: Each input contains a set of parameters for a given swaption. The native PARSEC input simply repeats the same parameters multiple times, causing the application to recalculate the same swaption price. We augment the evaluation input set with additional randomly generated parameters so that the application computes prices for a range of swaptions.

Precision Metric: Swaptions prints the computed prices for each swaption. The precision metric computes the distortion of the swaption prices (see Equation 4.1), weighting the prices equally to directly measure the application's ability to produce accurate swaption prices.

x264 Description: This media application encodes a raw (uncompressed) video according to the H.264 standard [94]. Like virtually all video encoders, it uses lossy encoding, with the visual quality of the encoding typically measured using continuous values such as peak signal-to-noise ration.

Knobs: We use three knobs: `--subme` (an integer parameter which determines the algorithms used for sub-pixel motion estimation), `--merange` (an integer which governs the maximum search range for motion estimation), and `--ref` (which specifies the number of reference frames searched during motion estimation). `--subme` ranges from 1 to 7, `--merange` ranges from 1 to 16, and `--ref` ranges from 1 to 5. In all cases higher numbers correspond to higher quality encoded video and longer encoding times. The PARSEC native defaults for these are 7, 16, and 5, respectively.

Inputs: The native PARSEC input contains a single high-definition (1080p) video. We use this video and additional 1080p inputs from xiph.org [3].

Precision Metric: The precision metric is the distortion of the peak signal to

noise ratio (PSNR, as measured by the H.264 reference decoder [33]) and bitrate (as measured by the size of the encoded video file), with the PSNR and bitrate weighted equally. This precision metric captures the two most important attributes of encoded video: image quality and compression.

bodytrack **Description:** This computer vision application uses an annealed particle filter and videos from multiple cameras to track a human's movement through a scene [24]. bodytrack produces two outputs: a text file containing a series of vectors representing the positions of body components (head, torso, arms, and legs) over time and a series of images graphically depicting the information in the vectors overlaid on the video frames from the cameras. In envisioned usage contexts [24], a range of vectors is acceptable as long as the vectors are reasonably accurately overlaid over the actual corresponding body components.

Knobs: bodytrack uses positional parameters, two of which we convert to knobs: `argv[5]`, which controls the number of annealing layers, and `argv[4]`, which controls the number of particles. The number of layers ranges from 1 to 5 (the PARSEC native default); the number of particles ranges from 100 to 4000 (the PARSEC native default) in increments of 100.

Inputs: bodytrack requires data collected from four carefully calibrated cameras. We use a sequence of 100 frames (obtained from the maintainers of PARSEC) as the training input and the PARSEC native input (a sequence of 261 frames) as the production input.

Precision Metric: The precision metric is the distortion of the vectors that represent the position of the body parts. The weight of each vector component is proportional to its magnitude. Vector components which represent larger body components (such as the torso) therefore have a larger influence on the precision metric than vectors that represent smaller body components (such as forearms).

swish++ **Description:** This search engine is used to index and search files on web sites. Given a query, it searches its index for documents that match the query and

returns the documents in rank order. We configure this benchmark to run as a server — all queries originate from a remote location and search results must be returned to the appropriate location.

Knobs: We use the command line parameter `--max-results` (or `-m`, which controls the maximum number of returned search results) as the single dynamic knob. We use the values 5, 10, 25, 50, 75, and 100 (the default value).

Inputs: We use public domain books from Project Gutenberg [1] as our search documents. We use the methodology described by Middleton and Baeza-Yates [67] to generate queries for this corpus. Specifically, we construct a dictionary of all words present in the documents, excluding stop words, and select words at random following a power law distribution. We divide the documents randomly into equally-sized training and production sets.

Precision Metric: We use F-measure [63] (a standard information retrieval metric) as our precision metric. F-measure is the harmonic mean of the *precision*[1] and *recall*. Given a query, precision is the number of returned documents that are relevant to the query divided by the total number of returned documents. Recall is the number of relevant returned documents divided by the total number of relevant documents (returned or not). We examine precision and recall at different cutoff values, using typical notation **P @N**.

Discussion These applications are broadly representative of our target set of applications — they all have a performance versus precision tradeoff and they all make that tradeoff available via configuration parameters. Other examples of applications with appropriate tradeoff spaces include most sensory applications (applications that process sensory data such as images, video, and audio), most machine learning applications, many financial analysis applications (especially applications designed for use in competitive high-frequency trading systems, where time is critically important), many scientific applications, and many Monte-Carlo simulations. Such applications (unlike more traditional applications such as compilers or databases) are typically

[1] At this point the term precision is overloaded. For the remainder of this paragraph *precision* refers to the information retrieval metric.

inherently approximate computations that operate largely without a notion of hard logical correctness — for any given input, they instead have a range of acceptable outputs (with some outputs more precise and therefore more desirable than others). This broad range of acceptable outputs, in combination with the fact that more precise outputs are often more computationally expensive to compute, gives rise to the performance versus precision tradeoffs that SEEC enables the applications to dynamically navigate.

There are a variety of reasons such applications would be deployed in contexts that require responsive execution. Applications that process soft real-time data for human users (for example, video-conferencing systems) need to produce results responsively to deliver an acceptable user experience. Search and information retrieval applications must also present data responsively to human users (although with less stringent response requirements). Other scenarios involve automated interactions. Bodytrack and similar probabilistic analysis systems, for example, could be used in real-time surveillance and automated response systems. High-frequency trading systems are often better off trading on less precise results that are available more quickly — because of competition with other automated trading systems, opportunities for lucrative trades disappear if the system does not produce timely results.

Chapter 5

Case Studies

5.1 Overview

This section presents numerous case studies demonstrating how SEEC's decoupled approach can be used to build real adaptive systems that meet goals accurately and efficiently. Section 5.2 discusses the overhead of the system. Next, Section 5.3 presents several simple examples demonstrating how performance, power, and precision can be controlled. Section 5.4 shows how SEEC can be used to manage power and performance tradeoffs on the machines from Section 4.2.1, maintaining a goal in one dimension and optimizing behavior in the other dimension. Section 5.5 shows how the same system can be used to tailor the behavior of a video encoder to specific inputs. Section 5.7 demonstrates how SEEC can control the behavior of applications. Section 5.6 shows some situations where SEEC's machine learning approach provides an advantage over adaptive control. Section 5.8 shows SEEC controlling multiple applications and reacting to a fluctuation in the environment.

Table 5.1: Performance of Heartbeats

Implementation	Heartbeat Throughput (Kbeat/s)	Heartbeat Latency (microseconds)
Shared Memory	1508.2	1.5
File I/O	0.9	136.2

5.2 Overhead

5.2.1 Heartbeats API

This section discusses several experiments conducted to measure the performance and overheads of our reference implementations of the Heartbeats API. There are two key metrics needed to evaluate the suitability of the interface for a given application or a system service. The first is the time taken to register a heartbeat, while the second is the delay from when the heartbeat is registered in an application to when it can be read in an external process. We refer to the first metric as the *heartbeat throughput* while the second is called the *heartbeat latency*.

To measure heartbeat throughput, we simply write an application that calls the heartbeat API function repeatedly in a loop. After exiting the loop, we read the global heart rate. The results for both the file-based and shared memory implementations are shown in Table 5.1. Not surprisingly, the shared memory implementation is significantly faster.

These throughput measures can be used to determine how much overhead the use of heartbeats will add to an application. For example, consider an application that anticipates a heart rate of 200 beats per second. Adding the shared memory based implementation of heartbeats will add 1/1500000s to each beat, for an overhead of approximately .01%. If instead, we used the file-based implementation of the API, we would expect an overhead of approximately 18.5%. Knowing these values allows applications developers to make informed decisions about the placement of Heartbeats within their applications.

We test the heartbeat latency of our implementations using two applications which "ping-pong" heartbeats between each other. Both applications emit heartbeats while

reading the other application's heartbeat data. The first application sends a heartbeat and then waits to see the second application register a heartbeat with the same tag. The second application works similarly, waiting for the first application and then emitting a heartbeat. We measure the time taken from when the first application sends its heartbeat until it detects a heartbeat with the same tag from the second application. We measure this value 10000 times and take the average. This average value represents the time taken to transmit two heartbeats (from the first application to the second and from the second back to the first) so we divide the time in half to obtain the heartbeat latency.

The values for heartbeat latency are also shown in Table 5.1. Knowing these values can aid the development of autonomic system services as heartbeat latency represents the minimum time required for the heartbeat data generated in an application to reach the service that is requesting this data. This also represents the minimum amount of time required for any change in behavior to be reflected in the heartbeat.

For the file-based implementation, there is a tradeoff between heartbeat throughput and heartbeat latency. The throughput could be increased by buffering several heartbeats and writing multiple heartbeats worth of data to the file. This will decrease the overhead of file i/o in the application but will delay the ability of an external process from reading this data. In fact, it would cause multiple heartbeats to appear to an external observer simultaneously. We have therefore chosen a file i/o implementation which minimizes heartbeat latency. Applications are free to reduce heartbeat overhead by registering heartbeats less often and making corresponding adjustments to their desired heart rates.

5.2.2 Overhead of Decision Engine

We account for SEEC's runtime overhead by measuring the time it takes to make a new decision, which requires calculating a speedup, selecting actions, and possibly updating the application and system models. On machine 1, classical control sustains 39.22 million decisions per second (d/s), adaptive control sustains 18.83 million d/s,

and adaptive actuator selection sustains 8.87 million d/s. In practice the overhead of signaling the heartbeat is greater than that of making a decision.

5.3 Controlling Performance, Power, and Precision

The first study demonstrates how SEEC can control performance, power, and precision. Example systems built to control each of the three metrics are described in turn.

5.3.1 Performance Examples

This section presents five examples of SEEC controlling performance. The first four show SEEC controlling application performance using various system-level adaptations available on machine 1 as described in Section 4.2.1. The fifth example shows SEEC controlling application-level adaptations for x264 exposed by the compiler framework described in Section 4.2.2.

For each example, we first measure the minimum and maximum performance available through static allocation of actions. In each example, the application requests a target performance that is average of the maximum and minimum. Three different versions of the SEEC runtime are compared: pure delay, slow convergence, and oscillating, and each corresponds to a different instantiation of the parameters in Equations 3.3–3.4 as described in Section 3.3.1.

The results of this study for all five systems controlling performance are shown in Figures 5-1–5-5. For each chart, the x-axis shows time while the y-axis shows performance normalized to the maximum value. The minimum and maximum performance of the system are shown with dotted lines, while the behavior of the three SEEC instantiations are shown with dashed lines.

Figure 5-1 shows SEEC controlling processor frequency for a single-core version of swaptions. Figure 5-2 shows the behavior of SEEC managing core allocation for

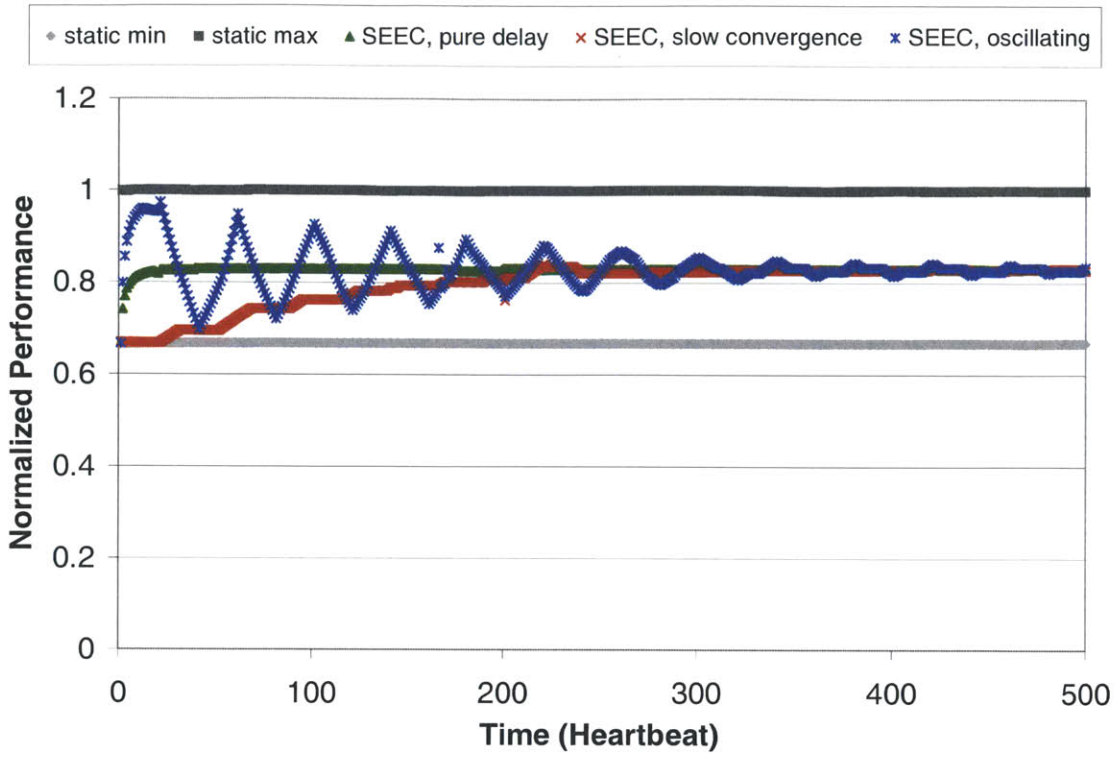


Figure 5-1: SEEC controlling processor speed for swaptions.

swaptions. In this case, the parallel version of swaptions is used and the noise in the feedback system increases significantly as cores are added. Despite this noise, the figure shows that all controllers converge to the desired performance, although the oscillating controller's curve is distorted. Figure 5-3 shows the behavior of SEEC controlling both clock speed and the number of cores. Again, all the curves converge to the desired performance with some distortion due to noise.

Figure 5-4 shows the behavior of SEEC on machine 1 managing the number of memory controllers assigned to the STREAM benchmark. As shown in the figure all controllers converge to the desired behavior. This is notable because the memory controller allocator only has two settings, but is still able to achieve arbitrary speedups using the SEEC control system. The "spikes" in the curves are due to the overhead of taking an action with this controller (which reallocates large chunks of memory).

Figure 5-5 shows the behavior of the adaptive x264 encoder using the PARSEC native input. Again, the controller is able to achieve the desired performance. In

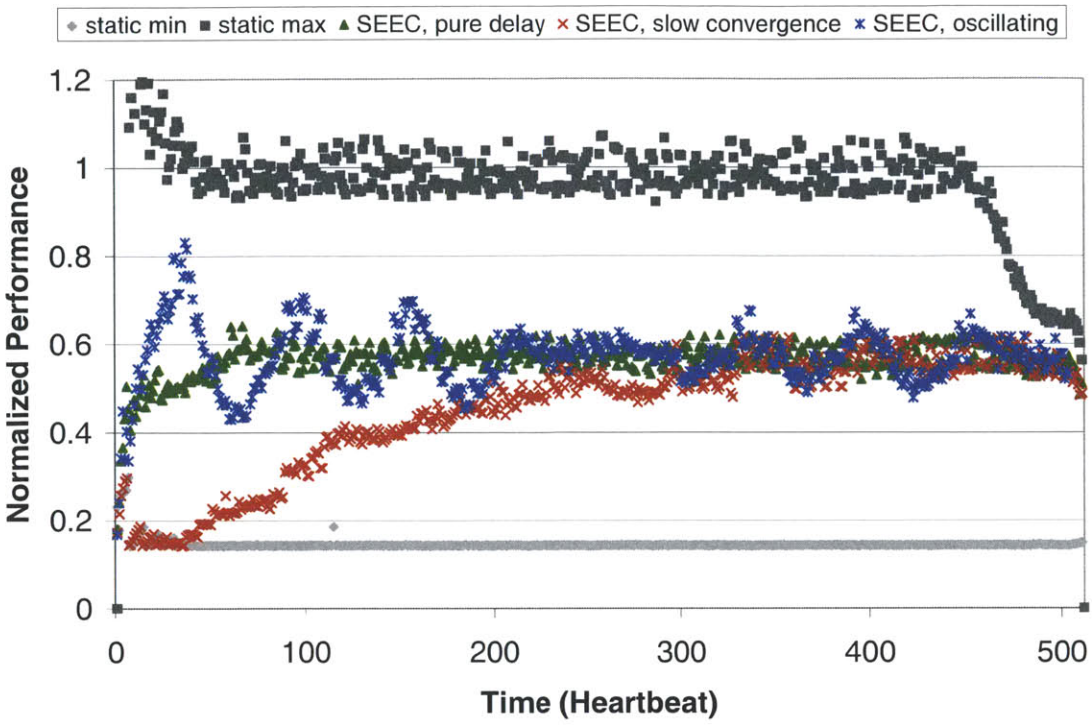


Figure 5-2: SEEC controlling core allocation for swaptions.

this case, the SEEC framework is able to turn an arbitrary application into a soft-real-time application with little work required on the part of the developer; SEEC automatically adjusts and controls performance using options that already existed as part of the software.

In summary, these results illustrate the generality and extensibility of the SEEC approach. While each example uses a different set of actuators, the SEEC runtime is able to manage all of them. These results demonstrate how SEEC can maintain accuracy despite its general approach to constructing adaptive systems; in all five examples SEEC converges to the target performance and does so following the trajectory predicted by Equations 3.3–3.4. This convergence is achieved despite the fact that the heartbeat signal is noisy for some examples. As expected, the oscillating controls are most affected by the presence of noise. The slowly converging controllers are least affected by noise while the pure delay controllers lie somewhere in between.

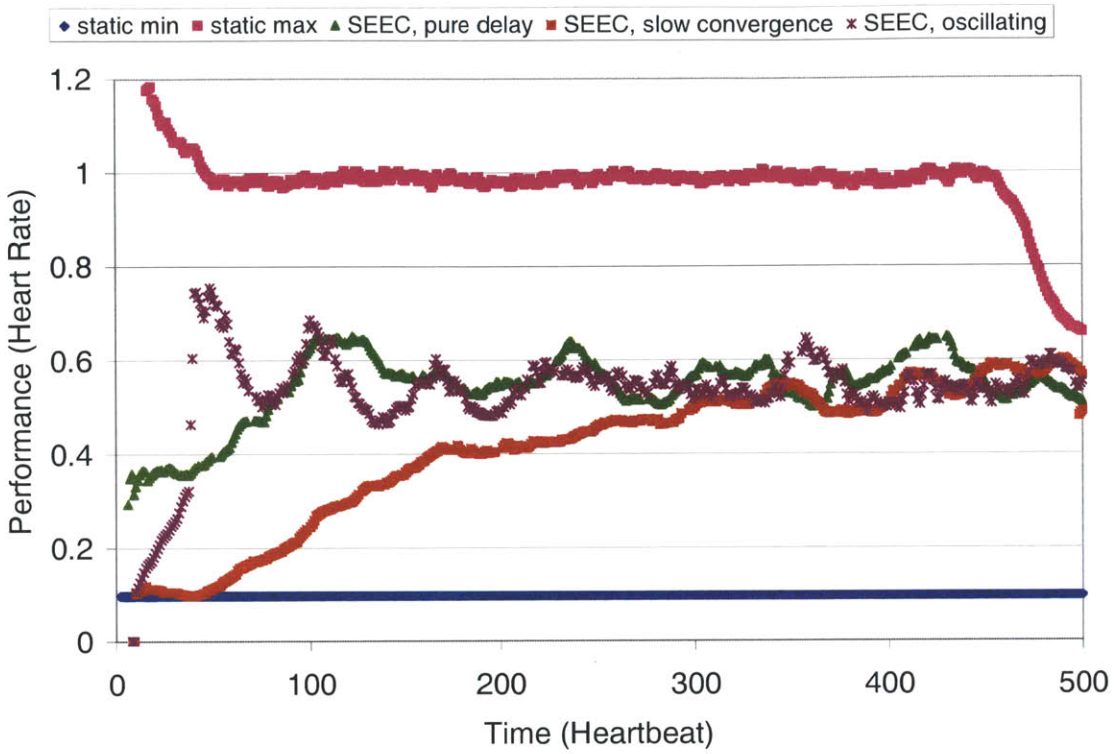


Figure 5-3: SEEC controlling cores allocation and processor speed for swaptions.

5.3.2 Power Example

This section presents an example showing SEEC controlling the power consumption of the dedup benchmark on machine 1 by managing all five available actuators. In this case, the minimum and maximum power consumption for this application are measured and the application requests a target performance that is halfway between these two values. SEEC's runtime is instantiated with a pure delay controller and the power consumption of the system under SEEC is compared to the power consumption of the system assigned maximum resources with no control.

Figure 5-6 shows the results of this experiment. Time (in seconds) is shown on the x-axis, while full system power consumption (in Watts) is shown on the y-axis. As can be seen in the figure, if this application is left uncontrolled, then the power spikes towards the middle of execution. However, using SEEC to control power, the spike is detected and reduced.

These results demonstrate that SEEC can accurately control power consumption

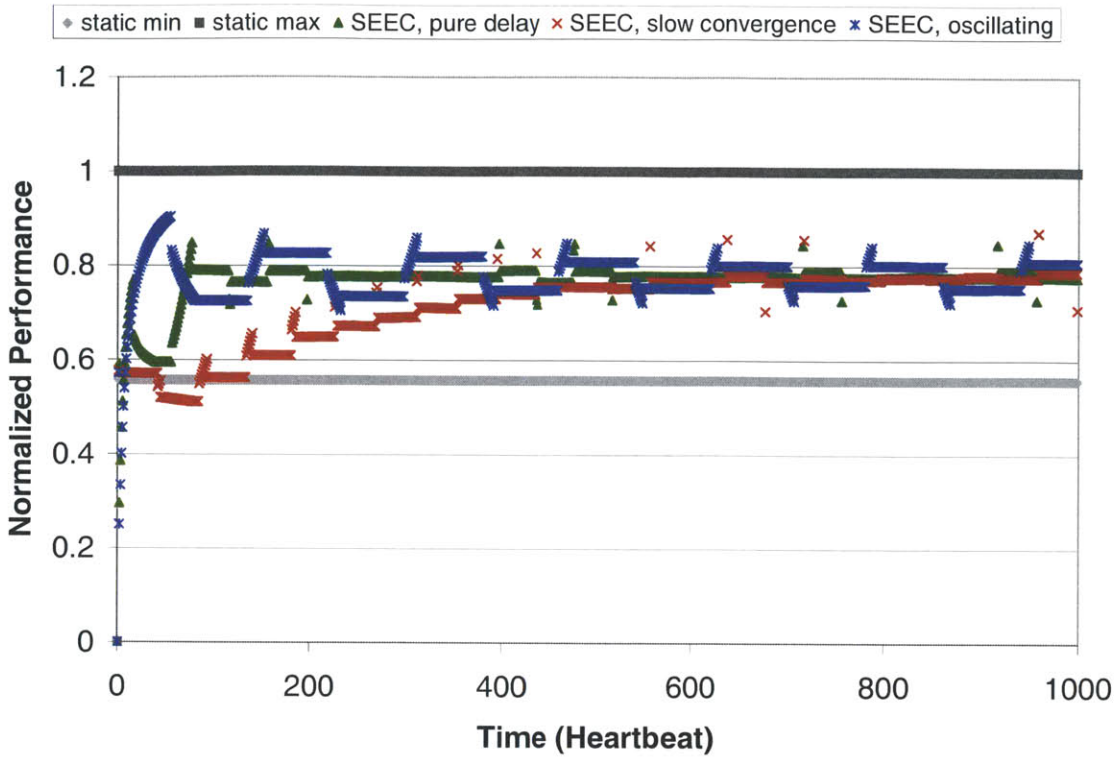


Figure 5-4: STREAM with the SEEC memory allocator.

as well as performance. In this case, SEEC significantly reduces the maximum power consumed by the system by eliminating a power spike.

5.3.3 Precision Example

This section presents an example showing SEEC controlling precision for the x264 benchmark on machine 1 by managing its algorithm using the actions exposed by the compiler described in Section 4.2.2. Here, the minimum and maximum precision for this application are measured and the application requests a target performance that is halfway between these two values. For this application, the precision refers to the peak signal to noise ratio (PSNR) achieved by the encoder and is measured on a frame by frame basis.

Controlling precision for this benchmark represents a challenge for SEEC. On average, the maximum precision configuration achieves a PSNR that is less than 3% higher than the minimum. This is a very small range of control. For this benchmark,

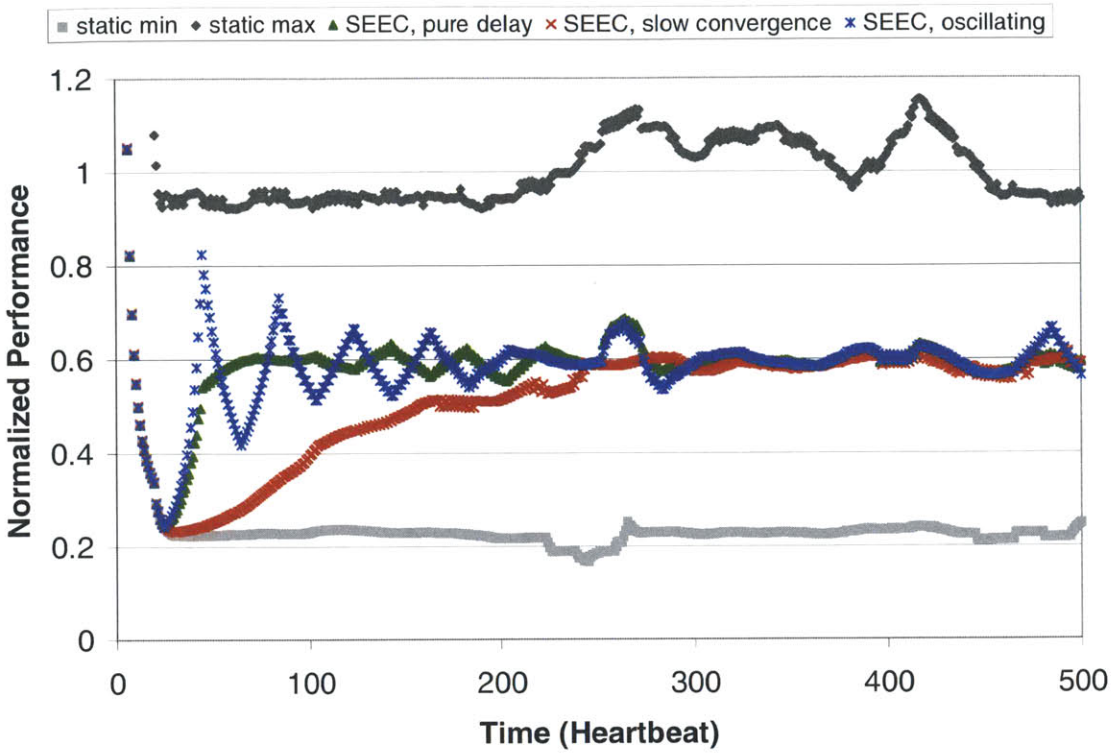


Figure 5-5: SEEC controlling application-level actions for x264.

variations in scene will have a larger effect on precision than any of the algorithmic knobs exposed to SEEC. Thus, in this example, SEEC will often be unable to achieve the requested precision. When SEEC detects that it is exceeding the precision goal, it will reduce precision and increase performance. When SEEC detects that it is below the precision goal, it will increase precision and use the most precise algorithm at a cost of performance.

Figure 5-7 shows the results of this experiment. Figure 5-7(a) shows precision as a function of time, with time (measured in heartbeats) shown on the x-axis and precision (measured in PSNR) shown on the y-axis. Figure 5-7(b) shows performance as a function of time with time shown on the x-axis and performance (measured in heart rate, or frame rate) shown on the y-axis. The figures show results when x264 is uncontrolled and when precision is actively controlled by the SEEC runtime system.

For this experiment, x264 processes a video with three distinct scenes, the first begins at heartbeat (frame) 0 and lasts until heartbeat 500, the second is from 500-

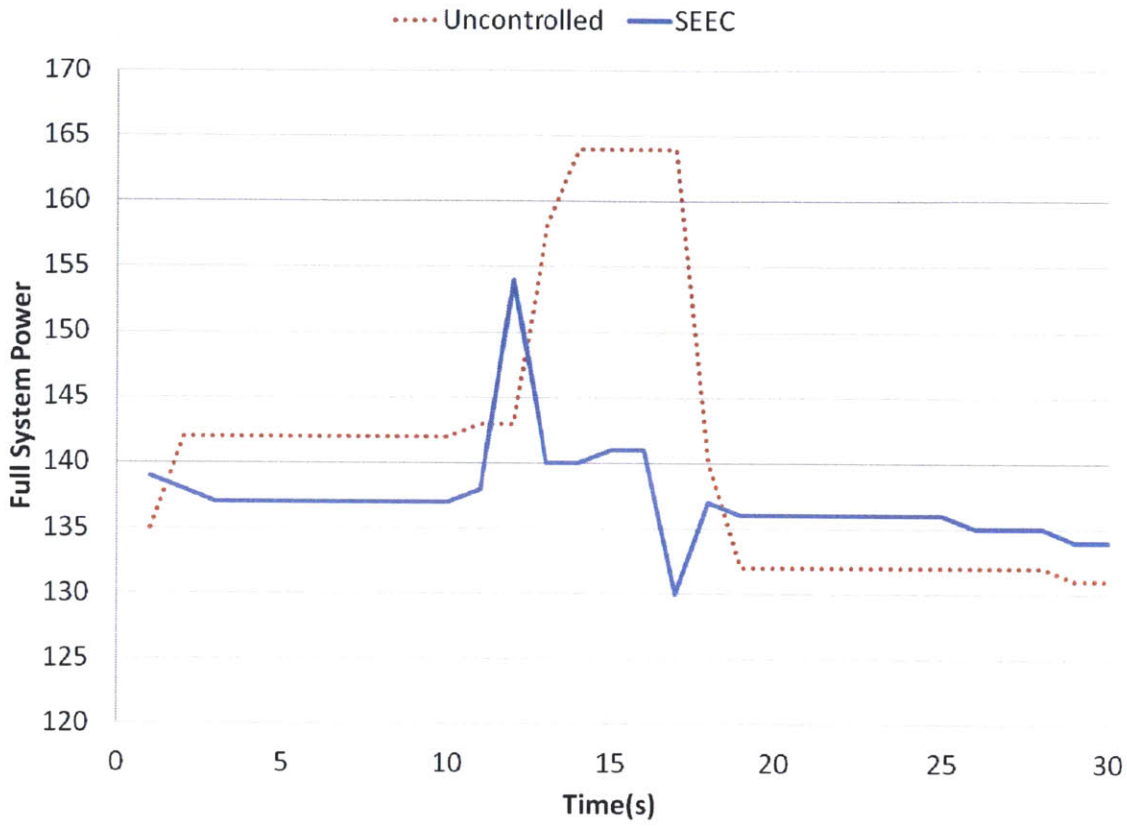
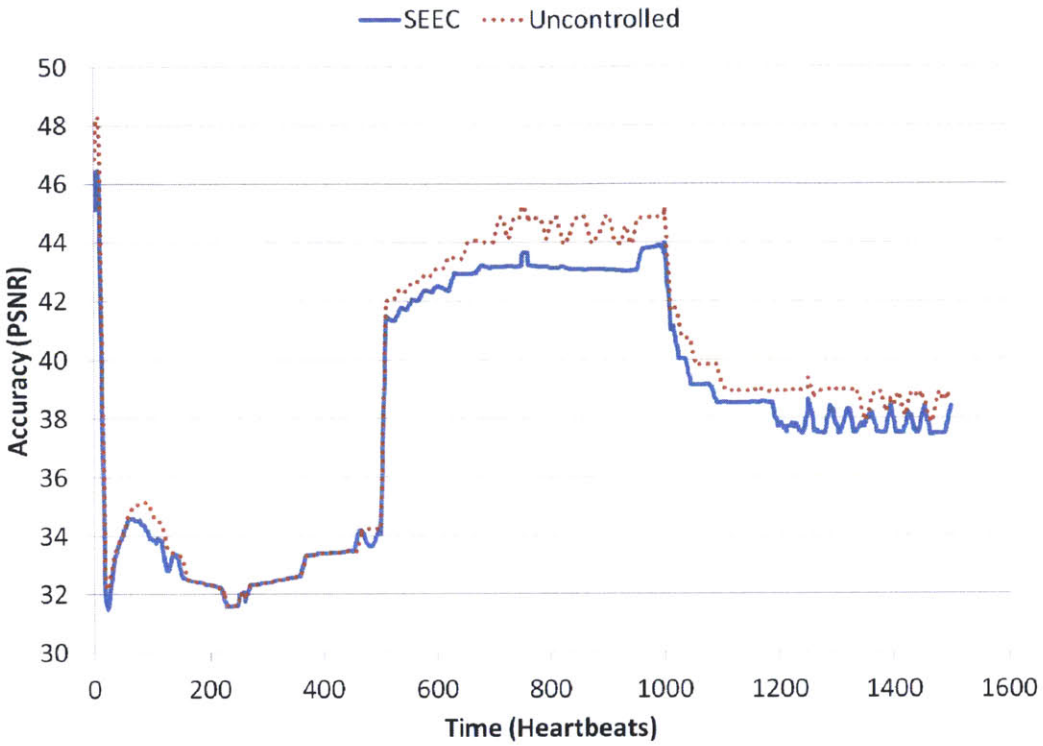


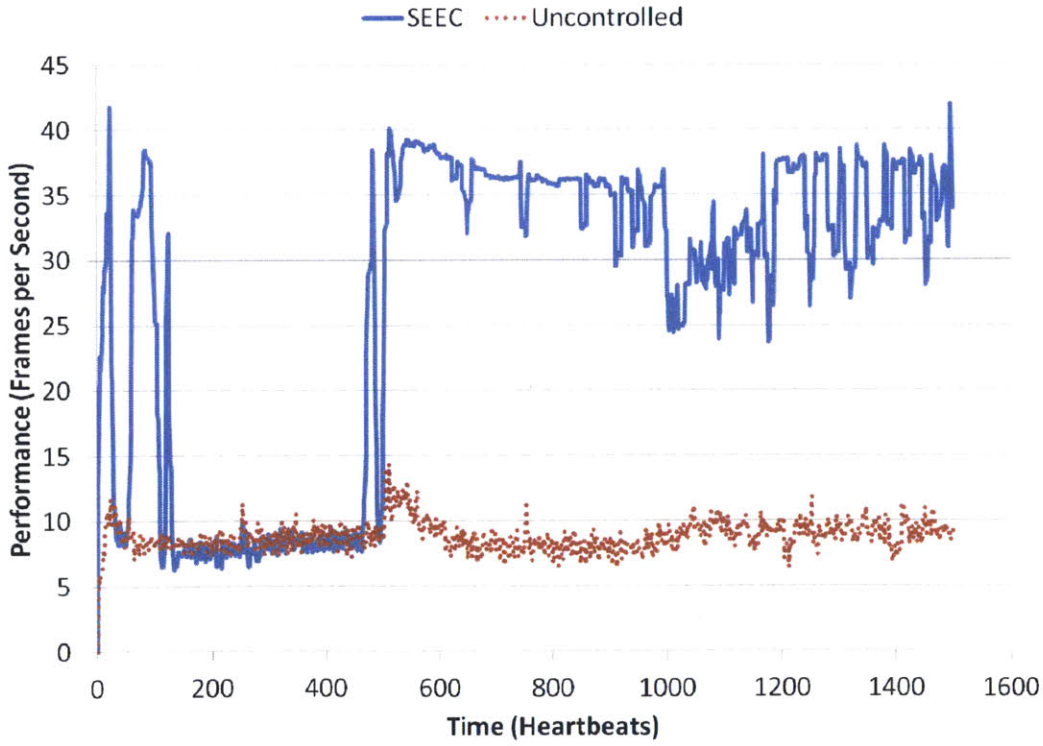
Figure 5-6: Controlling power consumption for the dedup benchmark.

1000, and the third is from 1000-1500. The first scene is the most difficult and during this scene, SEEC keeps precision at close to the maximum to try to meet the goal. The next two scenes are significantly easier to encode and here SEEC reduces precision because the goal is easily met even at the lowest precision setting. As shown in Figure 5-7(b) when SEEC is able to reduce the precision it significantly increases performance.

These results demonstrate SEEC's ability to control precision and to exchange precision for increased performance. These results hint at SEEC's ability to efficiently meet goals and the next section explores this in detail.



(a) Precision



(b) Performance

Figure 5-7: Controlling precision for the x264 benchmark.

5.4 Managing Power/Performance Tradeoffs

The second case study demonstrates how SEEC's decoupled approach can accurately and efficiently meet power and performance goals for a wide variety of applications (see Section 4.1) on the two machines described in Section 4.2.1.

5.4.1 Points of Comparison

To show the benefits of SEEC, we compare its decision engine to several other approaches. We compare two different instantiations of SEEC, one that uses adaptive control and adaptive actuate selection (referred to as SEEC AAS) and one that uses those two control schemes combined with machine learning (referred to as SEEC ML). These two versions of SEEC are compared to other approaches including:

Static Oracle:

This approach configures components for an application once, at the beginning of execution, but knows *a priori*the best setting for each benchmark. The static oracle is constructed by measuring the performance and power for all benchmarks with all available actuators on each machine. This approach provides an interesting comparison for active decision making as it represents the best that can be achieved without execution-time adaptation.

Uncoordinated Adaptation:

In this approach, components are tuned individually, without coordination. This approach uses all available actuators but each is tuned by an independent instance of the SEEC runtime. This approach represents what happens when the system-specific adaptive systems (see Section 2.3.1) work to manage the same application.

Classical Control:

This is the system described in Section 3.3.1. One difficulty implementing this approach is the specification of w in Equation 3.4. Ideally, w is determined on a per

application (or per input) basis, but these studies assume no *a priori* knowledge. Instead, we use a fixed value of $w = 0.5$ designed to maximize stability, as recommended in [62]. When controlling power, we use a fixed value of b which is easily measured for both of our test machines. The use of classical control as a point of comparison shows the benefits of SEEC's additional adaptive features over existing control-based approaches like that shown in [62].

Fixed System Adaptation:

This approach uses the SEEC runtime's adaptive control, but fixes the actuator selection strategy to the best strategy for the other for the other machine; i.e.,on machine 1, the best strategy for machine 2 is used, and vice versa. For example, if the best strategy for an application is *race-to-idle* on machine 1, then this approach uses *race-to-idle* on machine 2. This comparison demonstrates how system-specific approaches (see Section 2.3.1) fail to generalize when moving to a different set of components.

Dynamic Oracle:

At every heartbeat, this approach tunes actuators to the best settings for the next window of heartbeats. Obviously the dynamic oracle cannot be built in practice. Instead its behavior is computed after the fact by post processing empirical data for each application. The dynamic oracle represents an upper bound on the benefits of any adaptive system because it has 1) no overhead and 2) perfect knowledge of the future.

5.4.2 Metrics

To evaluate accuracy we compute:

- **Performance Error:** Is calculated as $(g - \min(g, h))/\ g$, where g is the performance goal, and h is the achieved performance (see Equation 3.1). This metric penalizes systems for not achieving the performance goal, but provides no reward or penalty for exceeding it. Note that exceeding the performance goal will

likely cause greater than optimal power consumption, and this will be reflected in the power efficiency.

- **Power Error:** Is calculated as $(g_{pow} - \max(g_{pow}, c))/\ g_{pow}$, where g_{pow} is the performance goal, and c is the achieved power consumption (see Equation 3.10). This metric penalizes systems for exceeding the power goal, but provides no reward or penalty for operating below it. Again, delivering power consumption below the goal will likely result in suboptimal performance which will be reflected in the performance efficiency. When measuring power error, we measure the total system power and subtract out the idle power, which magnifies the penalty for inaccuracy.

For the error metrics, lower values are better than higher ones.

To evaluate efficiency we compute:

- **Normalized Power:** Measures average power consumption when controlling a performance goal. Power is normalized to that achieved by the static oracle. For this metric lower values are better; normalized power consumption less than unity indicates a savings over the best possible non-adaptive strategy. We compute power by measuring total system power and subtracting out idle power.
- **Normalized Performance:** Measures average performance when controlling a power goal. Performance is normalized to that achieved by the static oracle. For this metric higher values are better; normalized performance greater than unity indicates higher performance than the best possible non-adaptive strategy.

For the efficiency metrics, lower values of normalized power are better, while higher values of normalized performance are better.

5.4.3 Controlling Performance and Minimizing Power

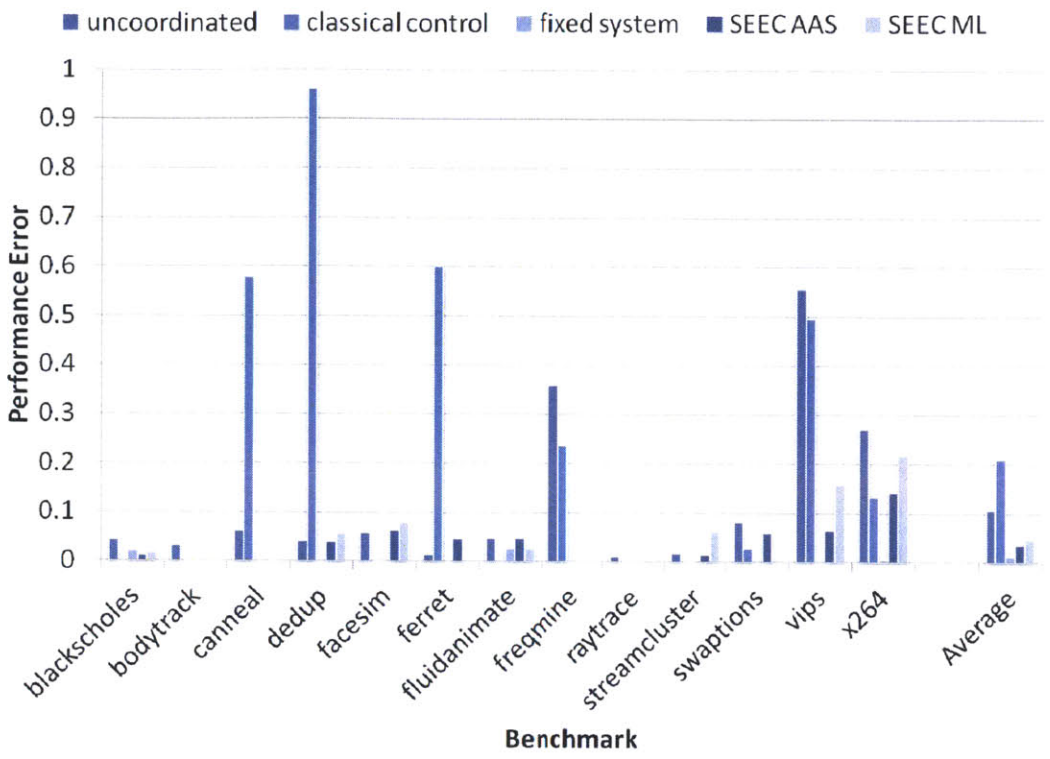
We launch each of the PARSEC benchmarks on a single core set to the minimum clock speed and each requests a performance equal to half the maximum achievable on Machine 1. For each benchmark, we compute the performance error and normalized

power and Figures 5-8 and 5-9 show the results. In all cases, the x-axis shows the benchmarks (and the average for all benchmarks). In Figures 5-8(a) and 5-9(a), the y-axes show performance error on the two different machines. In Figures 5-8(b) and 5-9(b), the y-axes show the normalized power for each machine (lower is better).

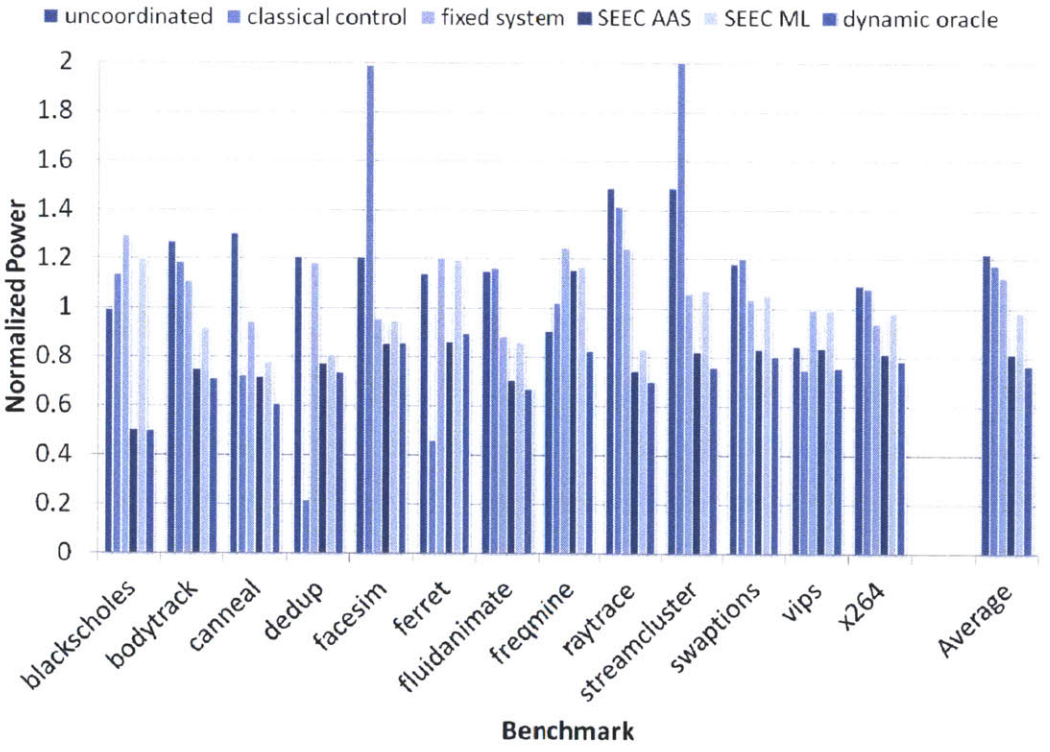
The results in Figures 5-8(a) and 5-9(a) indicate that, on average, SEEC is more accurate for performance targets than either uncoordinated adaptation or classical control. The average performance error on Machine 1 is 12.4% for uncoordinated control, 23.4% for classical control, and 4.5% for fixed system control. For SEEC, AAS achieves an error of 3.8%, while ML achieves an error of 4.8%. On Machine 2, the average performance error is 11.7% for uncoordinated control, 35.8% for classical control, and 9.0% for fixed system control. On the same machine, SEEC AAS achieves 1.4% error while ML achieves 4.7%. The lower performance error shows that both forms of SEEC do a better job of meeting goals than existing approaches.

Figures 5-8(b) and 5-9(b) show that SEEC is more efficient than uncoordinated adaptation; i.e.,provides lower power consumption for the given performance targets. In some cases, power consumption is lower than the dynamic oracle. These cases correspond to times when the performance target was missed so these additional savings are coming at a cost of not meeting the performance goal, whereas the dynamic oracle is 100% accurate. On machine 1, the normalized power is 1.18 for uncoordinated adaptation, 1.14 for classical control, and 1.09 for fixed system control. For SEEC, AAS achieves a normalized power of 0.82, while ML achieves 0.98. The best possible normalized power is that achieved by the dynamic oracle, 0.77. Uncoordinated, classical, and fixed system control are all worse than the static oracle, while SEEC is better. Uncoordinated control is 53% worse than the dynamic oracle while SEEC is only off of optimal by 7.8%.

On machine 2, the normalized power is 0.93 for uncoordinated adaptation, 0.95 for classical control, and 0.95 for fixed system control. In contrast, SEEC AAS achieves a normalized power of 0.84 while SEEC ML achieves a normalized power of 0.96. The dynamic oracle's normalized power is 0.79. In this case, all approaches improve on the static oracle, but SEEC AAS is closest to optimal. On machine 2, uncoordinated

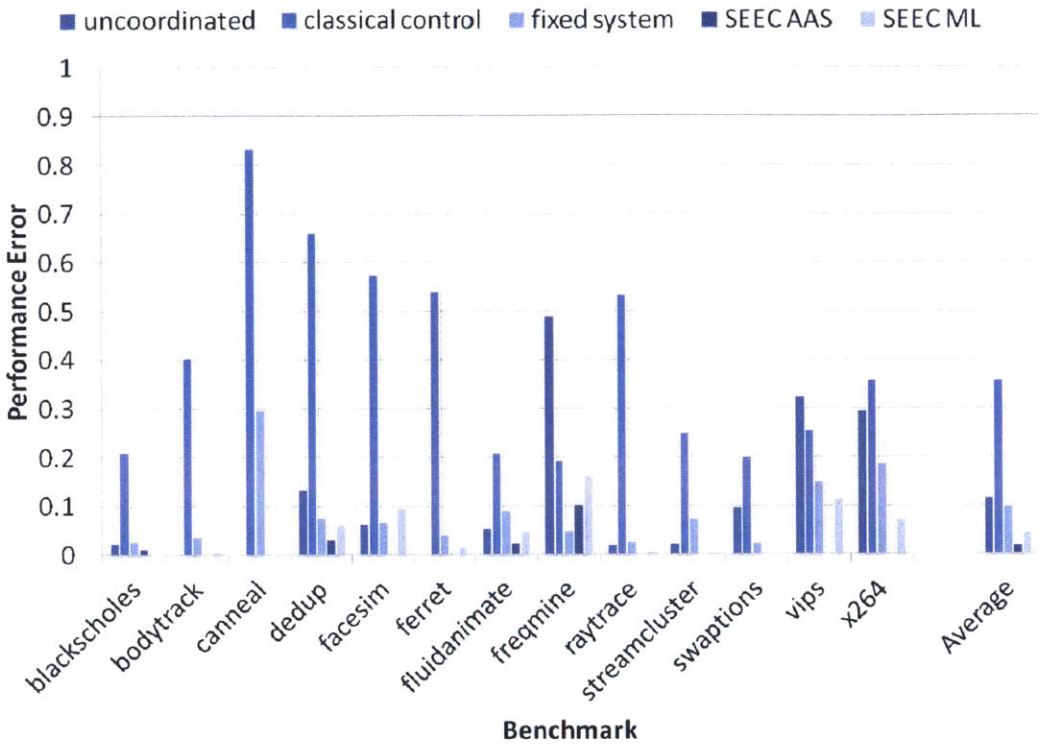


(a) Performance Error (lower is better)

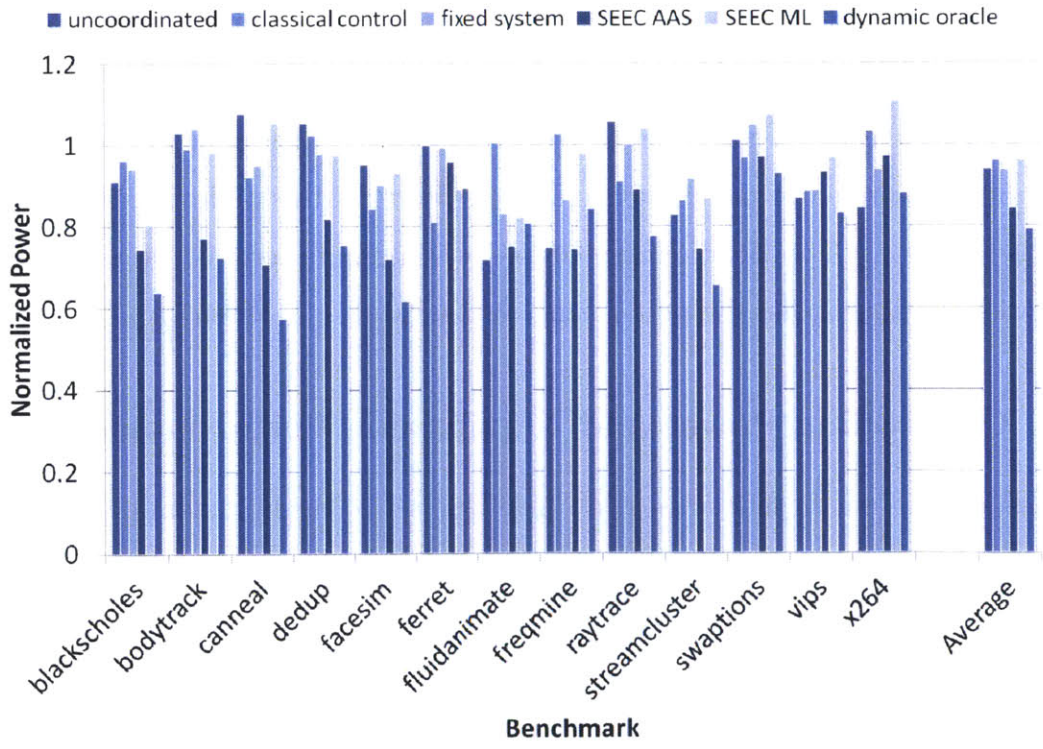


(b) Normalized Power (lower is better)

Figure 5-8: Controlling performance on machine 1.



(a) Performance Error (lower is better)



(b) Normalized Power (lower is better)

Figure 5-9: Controlling performance on machine 2.

control consumes 19% more power than the dynamic oracle while SEEC exceeds optimal by only 6.5%.

These results demonstrate the accuracy and efficiency of SEEC, especially SEEC AAS, which achieves the performance goal with low error and close to optimal savings. In contrast, uncoordinated control is less accurate, less efficient, and on Machine 1 is actually worse than the static oracle on average. Furthermore, the results indicate the benefits of SEEC's adaptive control and adaptive actuator selection, as SEEC is both more accurate and more efficient than classical control. These results illustrate that when coordination occurs through classical control methods, accuracy and efficiency can suffer unless control adapts to the application (e.g.,using adaptive control) and system (e.g.,using adaptive action scheduling) on which it is running.

On both machines, SEEC AAS outperforms SEEC ML. For this study, the system models are optimistic, but SEEC AAS is able to overcome errors because the relative costs and benefits are close to correct for these applications. The ML engine provides no additional benefit as it explores actions to learn exact models that are not necessary for efficient control. SEEC ML achieves a lower average error across both machines. The ML engine is more efficient than all adaptive systems (other than AAS) on Machine 1. On Machine 2, SEEC ML achieves greater accuracy than the non-SEEC approaches while providing comparable efficiency. Part of ML's performance relative to AAS is that the ML is still exploring when the benchmark terminates. For longer benchmarks ML approaches the performance of AAS.

5.4.4 Controlling Power and Maximizing Performance

We launch each of our benchmarks on a single core set to the minimum clock speed and each targets an average power halfway between the minimum and maximum achievable. For each benchmark, we compute the power error and normalized performance; Figures 5-10 and 5-11 show the results. In all cases, the x-axis shows the benchmarks (and the average for all benchmarks). A benchmark labeled with an asterisk denotes that the power savings is the same for the static and dynamic oracles. The y-axes in Figures 5-10(a) and 5-11(a) show the power error on the two

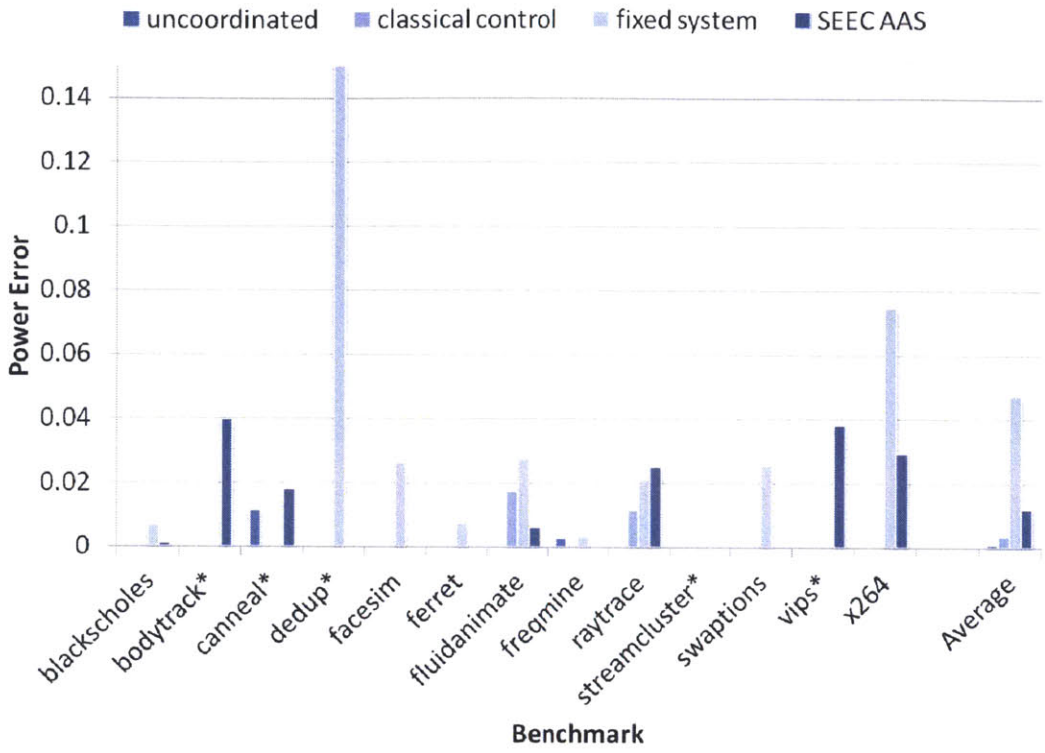
different machines. The y-axes in Figures 5-10(b) and 5-11(b) show the normalized performance for each machine (higher is better).

For this experiment, the benchmarks do not run long enough for SEEC ML to converge to a meaningful result due to the limitations on power sampling in our system. Therefore, all SEEC results in this section refer to SEEC AAS.

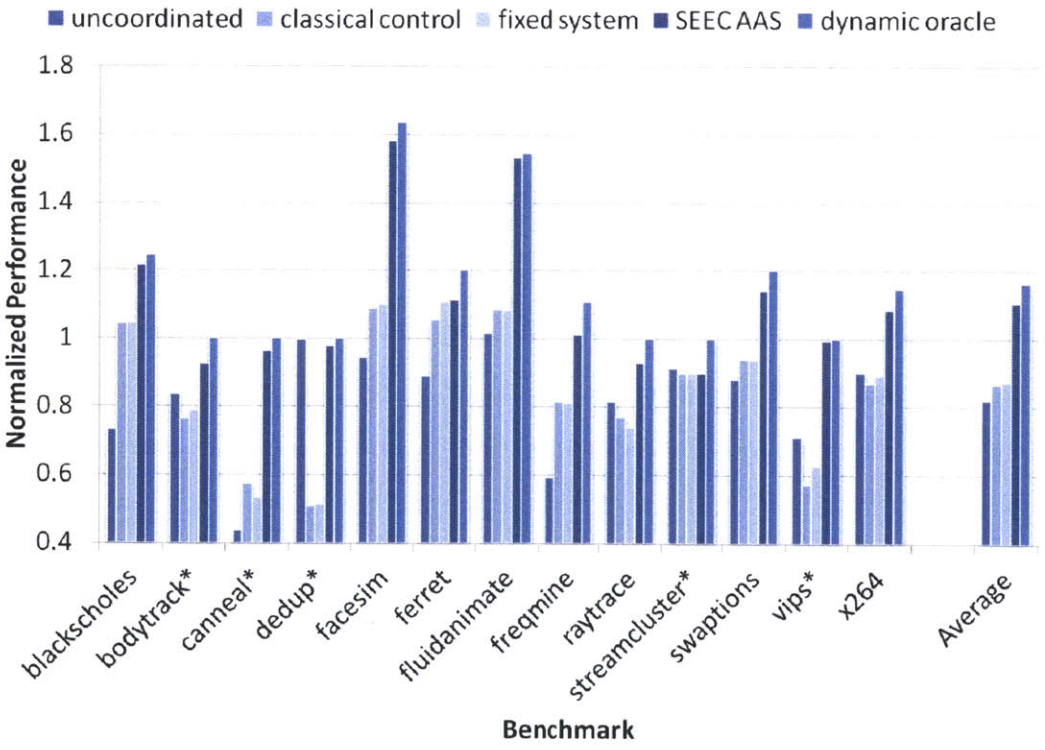
Figures 5-10(a) and 5-11(a) indicate that most approaches provide high accuracy on average. The average power errors on Machine 1 are 0.11% for uncoordinated control, 0.36% for classical control, 10.0% for fixed system, and 1.2% for SEEC. The average power errors on Machine 2 are 4.0% for uncoordinated control, 3.3% for classical control, 0.06% for fixed system, and 1.1% for SEEC. Overall, providing accurate control is easier for power than for performance, likely because it is less application dependent than performance, and all approaches do well.

Figures 5-10(b) and 5-11(b) show that, while all approaches are fairly accurate, SEEC is significantly more efficient for the given power targets. Again, performance can sometimes exceed that of the dynamic oracle when the power target is missed so the additional speed comes at a cost of not meeting the power goal. On machine 1, the normalized performance is 0.82 for uncoordinated adaptation, 0.87 for classical control, 0.88 for fixed system, 1.11 for SEEC and 1.16 for the dynamic oracle. As was the case when controlling performance on machine 1, uncoordinated, classical, and fixed system are worse than the static oracle, while SEEC is better. Fixed system control achieves only 67% of optimal performance, while SEEC achieves 96% of the maximum performance for the given power target. On machine 2, the normalized performance is 1.08 for uncoordinated adaptation, 1.30 for classical control, 0.95 for fixed system, 1.39 for SEEC and 1.43 for the dynamic oracle. Again, all approaches improve on the static oracle for machine 2, but SEEC is significantly closer to optimal. On machine 2, fixed system adaptation achieves only 76% of the maximum performance while SEEC again achieve 96% of the best possible performance.

These results show that all techniques can accurately manage power goals. However, SEEC is more efficient, providing far more performance for a given power goal than either uncoordinated adaptation or classical control.

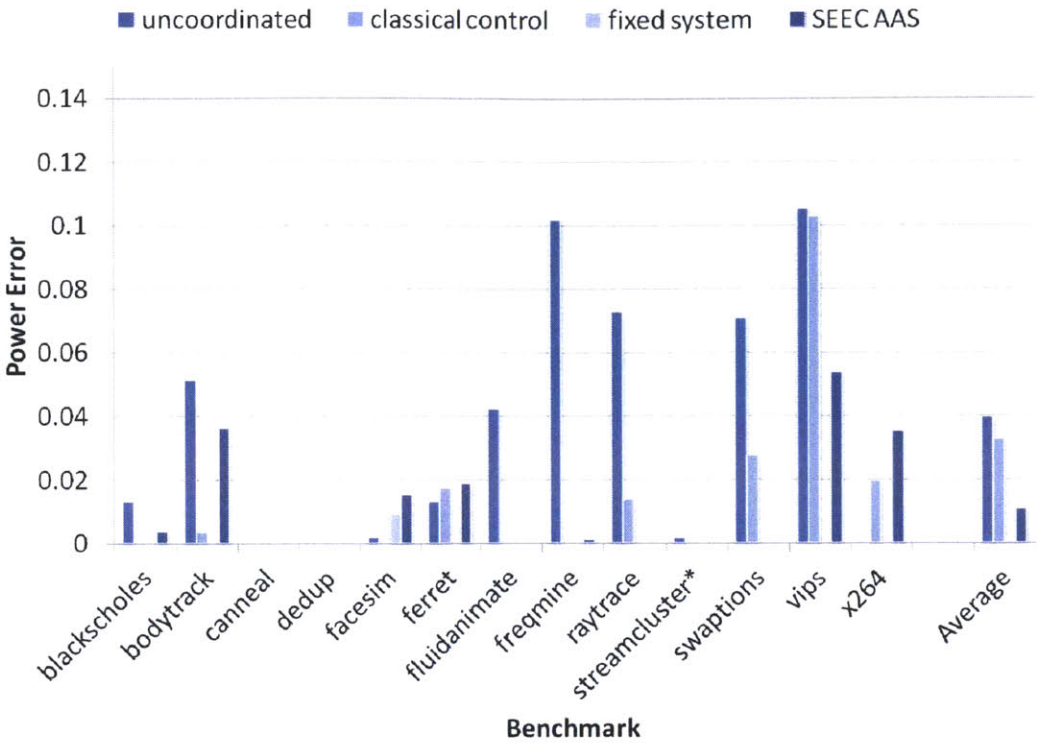


(a) Power Error (lower is better)

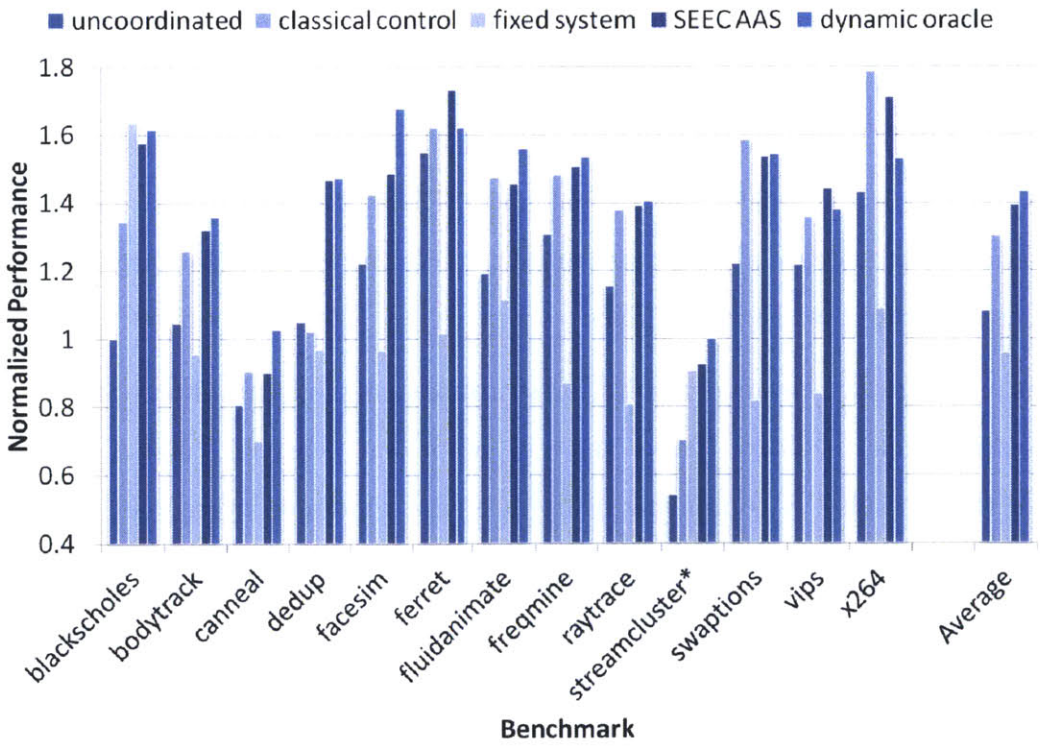


(b) Normalized Perf. (higher is better)

Figure 5-10: Controlling power on machine 1.



(a) Power Error (lower is better)



(b) Normalized Performance (higher is better)

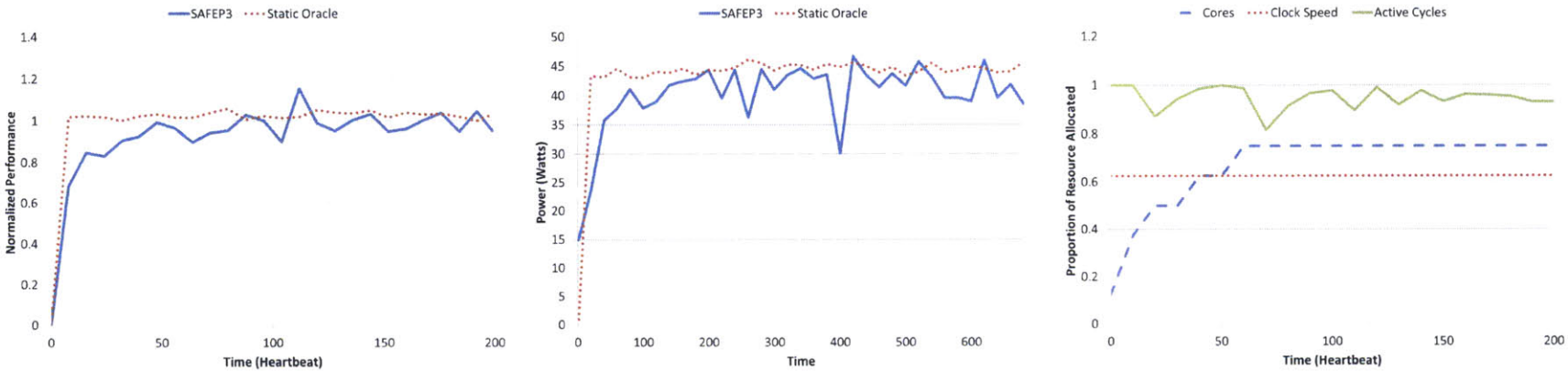
Figure 5-11: Controlling power on machine 2.

5.4.5 Detailed Results

This section discusses detailed results showing the behavior of SEEC AAS as a function of time for two benchmarks. We show one example of SEEC AAS controlling a low-variance benchmark (facesim) and one example of SEEC AAS controlling a high-variance benchmark (swaptions) (refer to Table 4.2 for variance of applications).

Figure 5-12 shows detailed results for the facesim benchmark when controlled by both SEEC and by the static oracle. Figures 5-12(a)–5-12(c) show the power consumption (subtracting out idle power), performance (normalized to the performance goal), and actuator settings as a function of time for facesim on machine 1. Figures 5-12(d)–5-12(f) show the same data for facesim on machine 2. In both cases, SEEC's performance is very close to that of the static oracle which is close to the desired performance. This is not surprising since facesim is a very regular benchmark; however, SEEC saves power compared to the static oracle in both cases. On machine 1, SEEC saves 7% on average power consumption by allocating the minimal amount of resource required to meet goals and idling for short amounts of time when there is slack in the schedule, as shown in Figure 5-12(c). On machine 2, SEEC's average performance is 98% of the target performance while its average power consumption is 27% less than the static oracle. On this machine, SEEC uses adaptive action scheduling to periodically allocate all resources and then idle the machine for large portions of time, as illustrated in Figure 5-12(f).

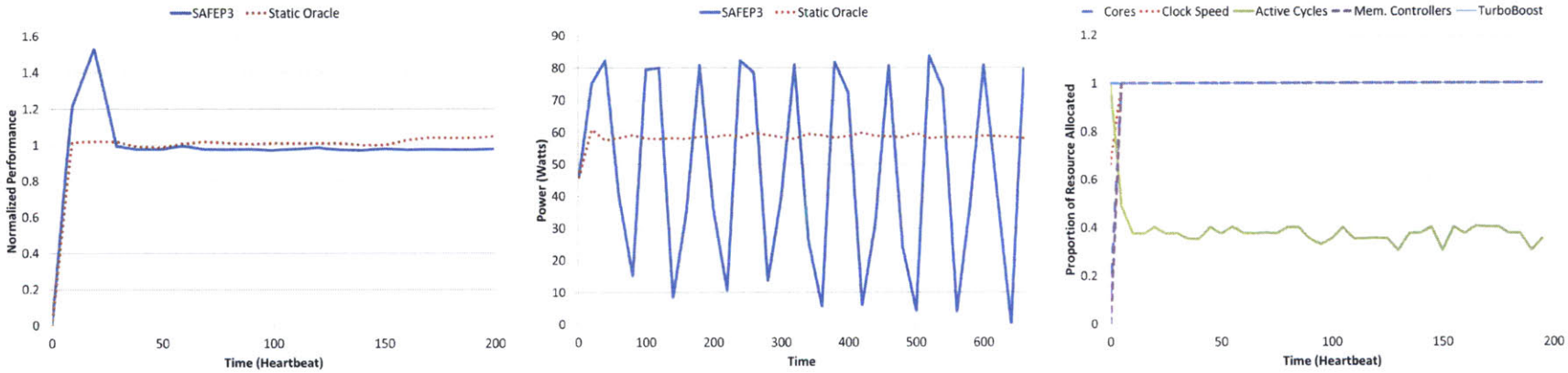
Figure 5-13 shows detailed results for swaptions controlled by both SEEC and by the static oracle. Given the variance in swaptions' heart rate signal, we have smoothed these results by computing heart rate on a windowed average of 16 heartbeats. Despite this, it is apparent that swaptions' behavior is much less regular than facesim. As an added benefit, SEEC holds average performance much closer to the desired level. On machine 1, SEEC achieves 94% of the target performance while consuming 10% less average power. For this machine SEEC allocates the minimal amount of resources required to meet goals and idles for very short amounts of time when there is slack in the schedule, as shown in Figure 5-13(c). On machine 2, SEEC



(a) Performance, Machine 1

(b) Power, Machine 1

(c) Actuators, Machine 1

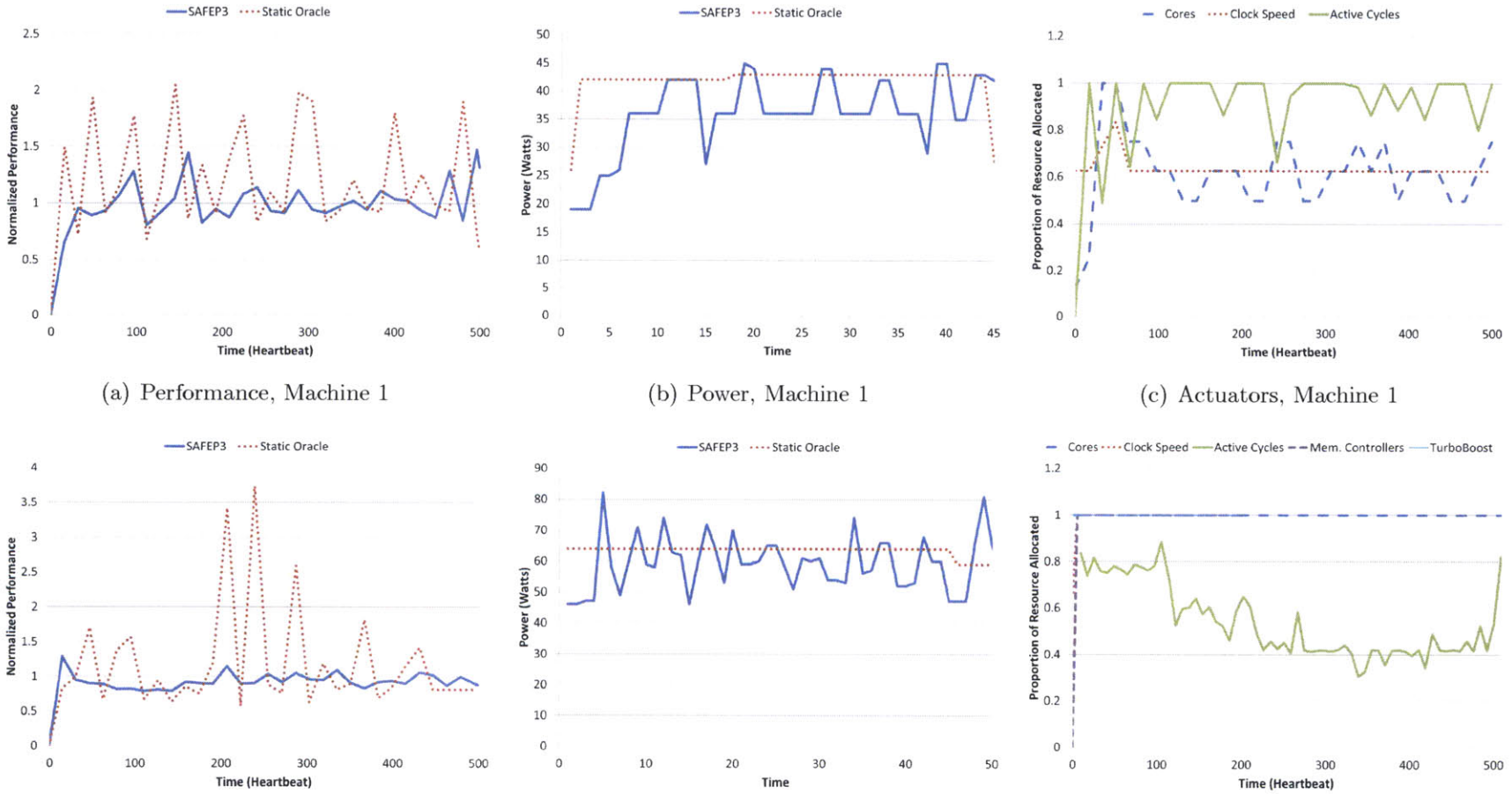


(d) Performance, Machine 2

(e) Power, Machine 2

(f) Actuators, Machine 2

Figure 5-12: Details of SEEC controlling facesim on two different machines.



(a) Performance, Machine 1

(b) Power, Machine 1

(c) Actuators, Machine 1

(d) Performance, Machine 2

(e) Power, Machine 2

(f) Actuators, Machine 2

Figure 5-13: Details of SEEC controlling swaptions on two different machines.

Table 5.2: Summary Results.

	Power Goals		Perf. Goals	
Method	Accuracy	Perf.	Accuracy	Power
Uncoord.	2.0%	73.2%	11.2%	38.8%
Classical	2.0%	83.0%	28.5%	37.5%
Fixed Sys.	2.4%	71.5%	5.6%	33.5%
SEEC AAS	**1.2%**	**96.1%**	**2.9%**	**7.2%**
SEEC ML			**4.8%**	**27.25%**

achieves 95% of the target performance while consuming 8.6% less average power than the static oracle. SEEC achieves these results using adaptive action scheduling to periodically allocate all resources and then idle the machine for large portions of time, as illustrated in Figure 5-13(f).

These results demonstrate the generality of the SEEC approach with respect to both applications and components. SEEC can accurately and efficiently manage both high-variance applications, like swaptions. In addition, SEEC can handle different sets of components with different tradeoffs. As shown in the figures, SEEC adopts different strategies on the two different machines and does so without redesign or re-implementation.

5.4.6 Summary

Table 5.2 summarizes the average behavior for each of uncoordinated, classical, and SEEC when managing power and performance goals. For power goals, the table shows the percentage of the maximum performance that was achieved. For performance goals, the table shows the additional power consumption over optimal.

The results show SEEC manages performance and power goals accurately and efficiently. SEEC outperforms other approaches for several reasons. SEEC beats the classical control system as adaptive control tailors response to specific applications and inputs. SEEC outperforms uncoordinated and fixed system adaptation because adaptive actuator selection takes a global view and avoids combinations of actuators that are suboptimal. SEEC outperforms the static oracle by adapting to phases within an application and tailoring resource usage appropriately.

In addition, these results illustrate that SEEC can work with different types of applications. As shown in Table 4.2 some benchmarks have high variance but SEEC can still meet those applications' goals accurately and efficiently.

Finally, these results show that SEEC can work with different sets of components without redesign and re-implementation. The two different machines used in this study have different sets of components and the most effective strategy is different for each machine. Despite these differences, the same SEEC runtime is able to accurately and efficiently manage resources on both platforms, demonstrating the generality of this approach with respect to the components being coordinated. SEEC provides more accuracy and more efficiency than fixed system adaptation because it tailors its response to the available components.

5.5 Adapting to Workload Fluctuations

This case study shows how SEEC can maintain a performance goal and minimize power consumption even when the application workload changes.

In this experiment, SEEC's decision engine maintains desired performance for the x264 video encoder across a range of inputs, each with differing compute demands. We use fifteen 1080p videos from xiph.org and the PARSEC native input. We alter x264's command line parameters to maintain an average performance of thirty frames per second on the most difficult video using all compute resources available on Machine 1. x264 requests a heart rate of 30 beat/s corresponding to a desired encoding rate of 30 frame/s. Each video is encoded separately, initially launching x264 on a single core set to the lowest clock speed.

5.5.1 Point of Comparison

In this study, we compare SEEC AAS and SEEC ML to the static oracle and classical control system (defined in Section 5.4). In addition, we compare to a scheme that allocates for *worst-case execution time* (wcet).

The wcet allocator knows *a priori* the amount of compute resources required to

meet an application's goals in the worst case (e.g.,for the most difficult anticipated input). We use this allocator as a point of comparison for the x264 video encoder benchmark, and we construct it by measuring the amount of resources required to meet goals for the hardest video. The wcet allocator assigns this worst-case amount of resources to all inputs.

5.5.2 Metrics

We measure the performance per Watt for each input when controlled by the classical control system, the wcet allocator, SEEC AAS and SEEC ML. Figure 5-14 shows the results of this case study. The x-axis shows each input (with the average over all inputs shown at the end). The y-axis shows the performance per Watt for each input normalized to the static oracle.

5.5.3 Results

On average, SEEC AAS outperforms the static oracle by 1.1×, the classical control system by 1.25×, and the wcet allocator by 1.44×. SEEC AAS bests these alternatives because its adaptive control system tailors response to particular videos and even phases within a video. Additionally, SEEC adaptively races-to-idle allowing x264 to encode a burst of frames using all resources and then idling the system until the next burst is ready. On average, SEEC AAS achieves 99% of the desired performance, while SEEC ML achieves 93% of the desired performance.

SEEC AAS again outperforms SEEC ML in this study, although the difference is just 10%. Again, the system models used here assume linear speedup and that is good enough for SEEC AAS to control x264. Using ML, SEEC explores actions to learn the true system models on a per input basis, but the exploration causes performance goals to be missed without a large resulting power savings. Despite this inefficiency, SEEC's ML approach achieves equivalent performance to the static oracle, and outperforms both classic control and wcet.

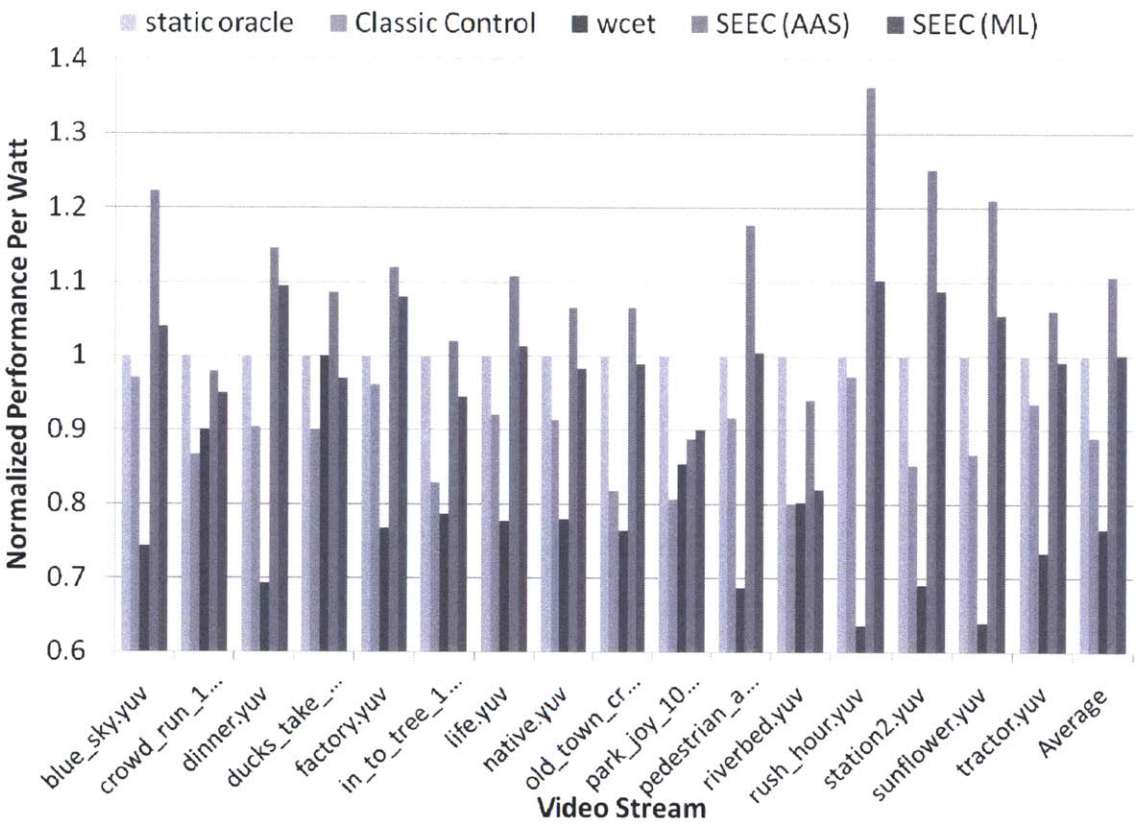


Figure 5-14: Controlling multiple x264 inputs with SEEC.

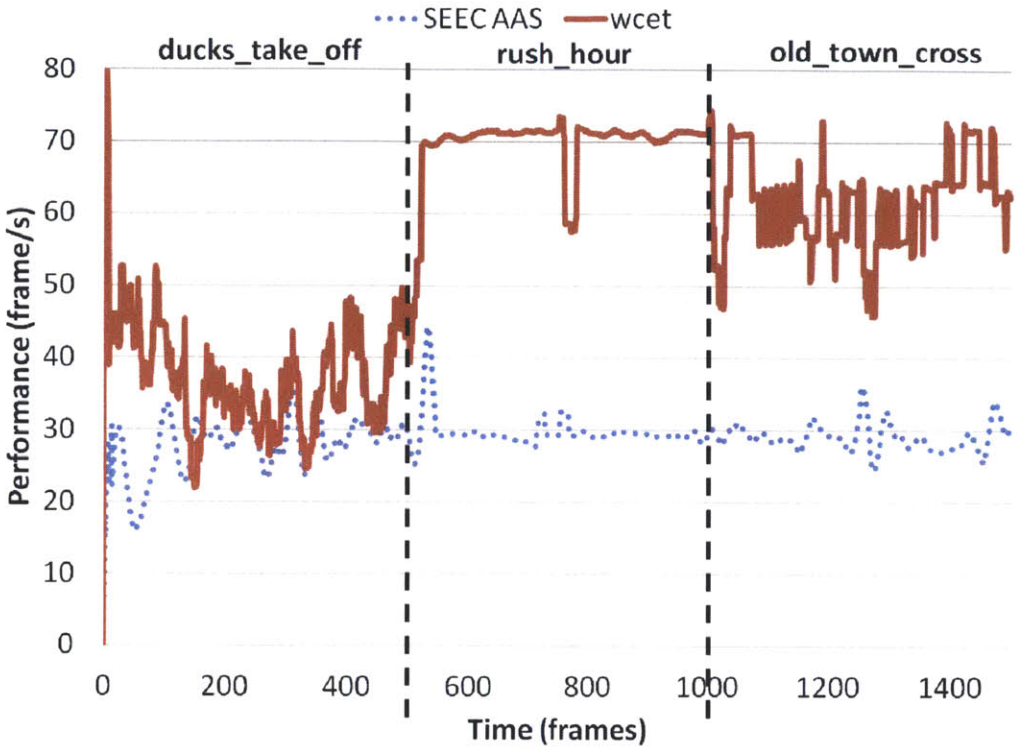
5.5.4 Detailed Results

SEEC outperforms the other approaches because it can adapt to phases within an application. To illustrate this, we create a new video input by concatenating three of our individual inputs: ducks_take_off, rush_hour, and old_town_cross. The first input is the hardest, the second one is easiest, and the third is in between. The encoder requests a performance of 30 frames per second. Concatenating these together creates a new video with three distinct phases, which forces SEEC to adapt to maintain performance as the workloads vary.

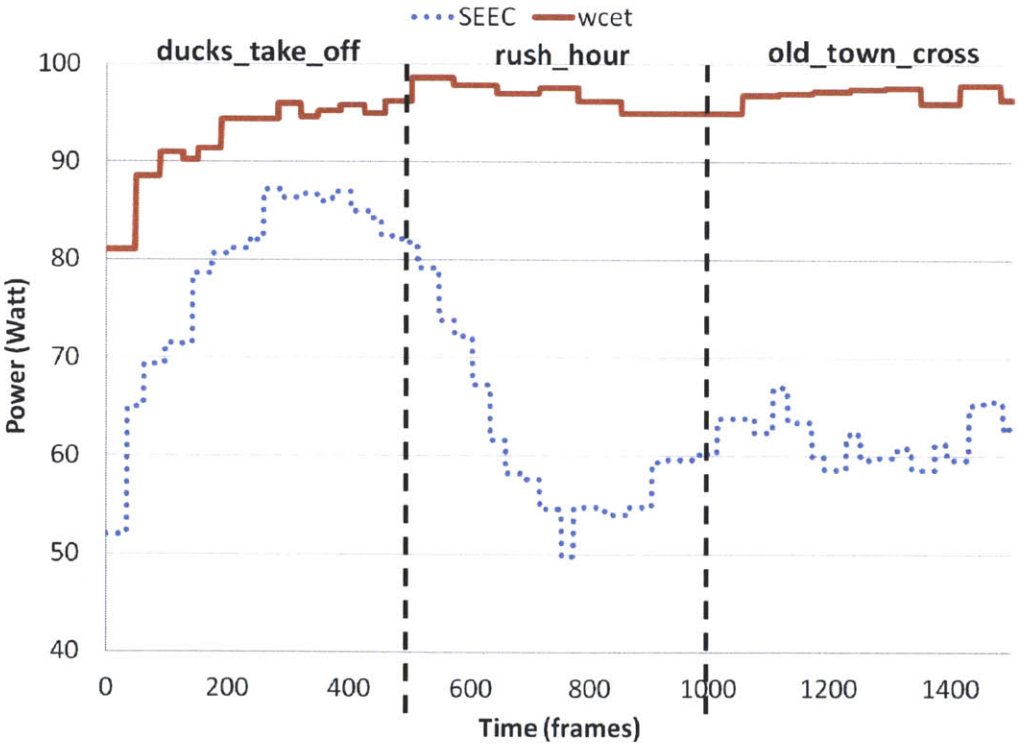
Figure 5-15 presents the results when SEEC controls x264 encoding the concatenated video. The x-axes show time, measured in heartbeats, and the y-axes show performance (Figure 5-15(a)) measured in frames per second and power (Figure 5-15(b)). Results are shown for both SEEC AAS and the worst-case-execution-time (wcet) allocator. As shown in the figures, the first phase causes SEEC to work hard and there are some sections for which neither SEEC nor wcet can maintain the target goal. In the second phase, SEEC quickly adjusts to the ease in difficulty and maintains the target performance while wcet has reserved over twice as many resources as needed, consuming unnecessary power. In the final phase, SEEC is able to closely maintain the target performance and save power despite the noise evident in this portion of the video.

5.6 Learning Models Online

In this section we test SEEC's ability to learn system models online and demonstrate two cases where SEEC's ML engine provides a clear benefit over AAS alone. For these test cases, SEEC must control the performance of STREAM and dijkstra on Machine 1 using the adaptations described in Section 4.2.1. Both applications request a heart rate of 75% the maximum achievable. We run these applications separately, each initially allocated a single core set to the lowest clock speed, and a single memory controller. We record the performance and power throughout execution for a classic control system, SEEC AAS, and SEEC ML.



(a) Performance



(b) Power

Figure 5-15: SEEC controlling x264.

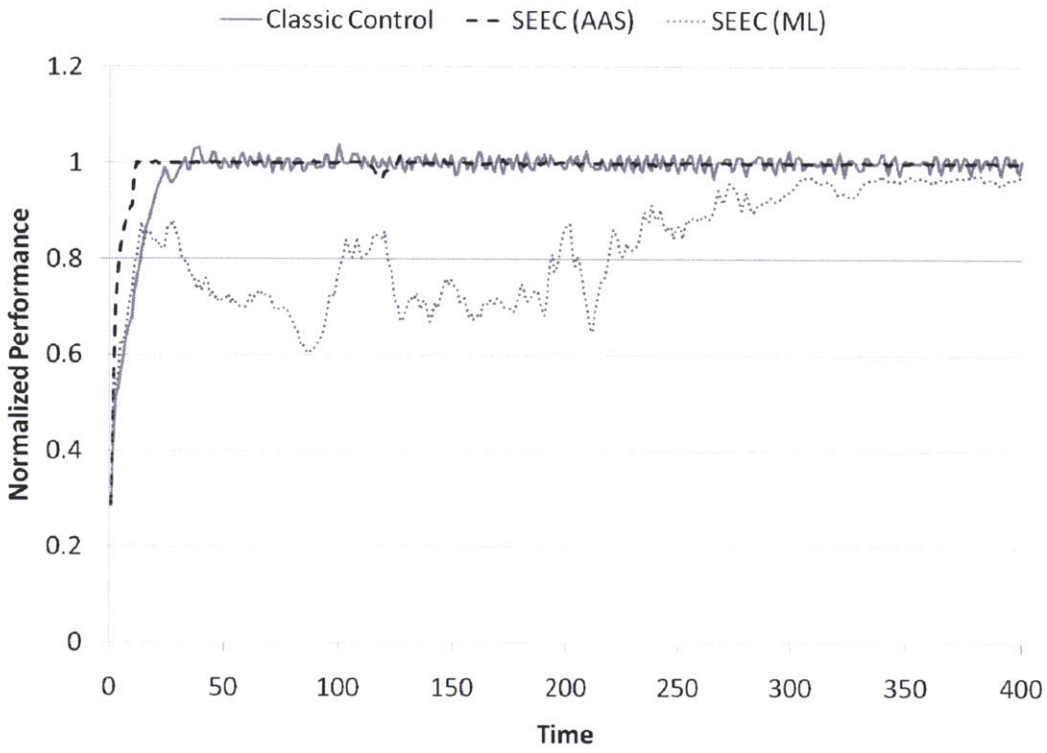
5.6.1 Results

The results of this case study are shown in Figures 5-16 and 5-17. In these figures time (measured in decision periods, see Section 3.3.3) is shown on the x-axis, while performance and power are shown on the y-axis of the respective figures and normalized to that achieved by a static oracle for the respective applications.

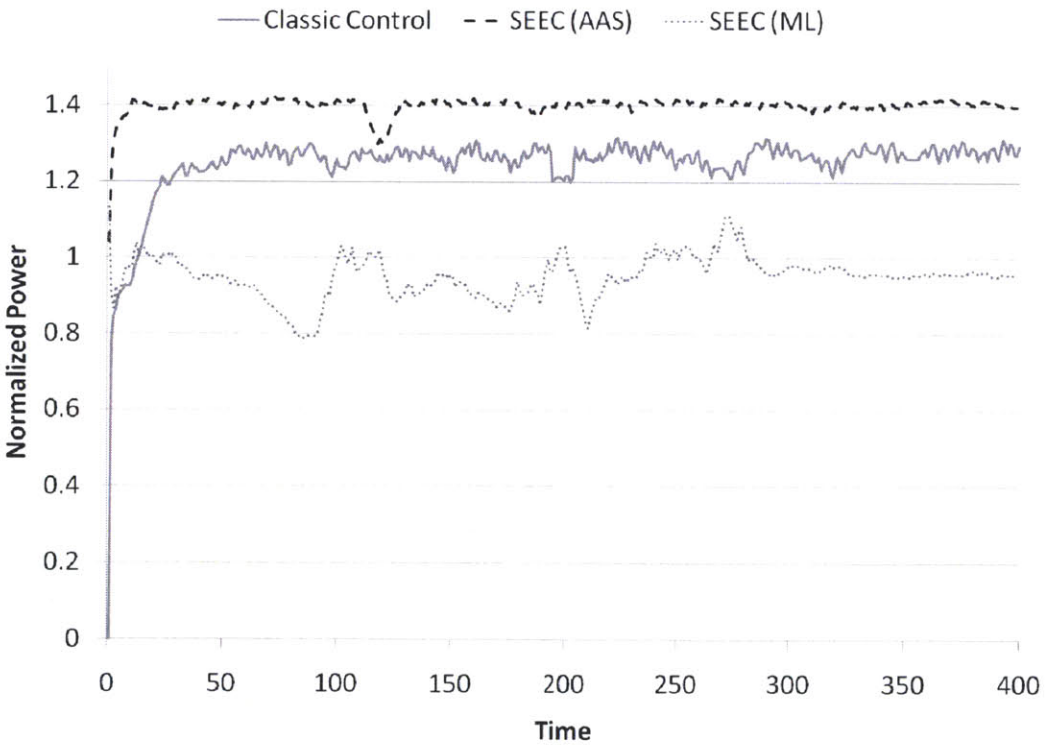
The two applications have very different characteristics, but the results are similar. In both cases, the SEEC AAS approach is the fastest to bring performance to the desired level, but it does so by over-allocating resources and using too much power. In the case of STREAM, SEEC AAS allocates the maximum amount of resources to the application and burns 40% more power to achieve the same performance. In the case of dijkstra, SEEC AAS over-allocates resources and then attempts to race-to-idle, but still burns about 10% more power than the static oracle. SEEC's AAS approach cannot adapt its system models and thus it cannot overcome the errors for these two applications.

In contrast, the SEEC ML approach takes longer to converge to the desired performance, but does a much better job of allocating resources. In the case of STREAM, SEEC ML is able to meet 98% of the performance goal while burning only 95% of the power of the static oracle. SEEC ML does so by allocating 4 cores and 2 memory controllers, running at the lowest clock speed, and using hybrid action selection to idle the system occasionally. For dijkstra, SEEC converges to the performance goal while achieving the same power consumption as the static oracle.

These experiments show the tradeoffs inherent using SEEC with and without ML. Without ML, SEEC quickly converges to the desired performance even when the system models have large error. However, these errors manifest themselves as wasted resource usage. In contrast, SEEC's ML engine takes longer to converge, but it does so without wasting resources. Both SEEC AAS and ML have advantages over the classic control system. SEEC AAS converges to the target performance more quickly, while SEEC ML saves average power. For dijkstra, the classic control system never converges, instead oscillating around the desired value.

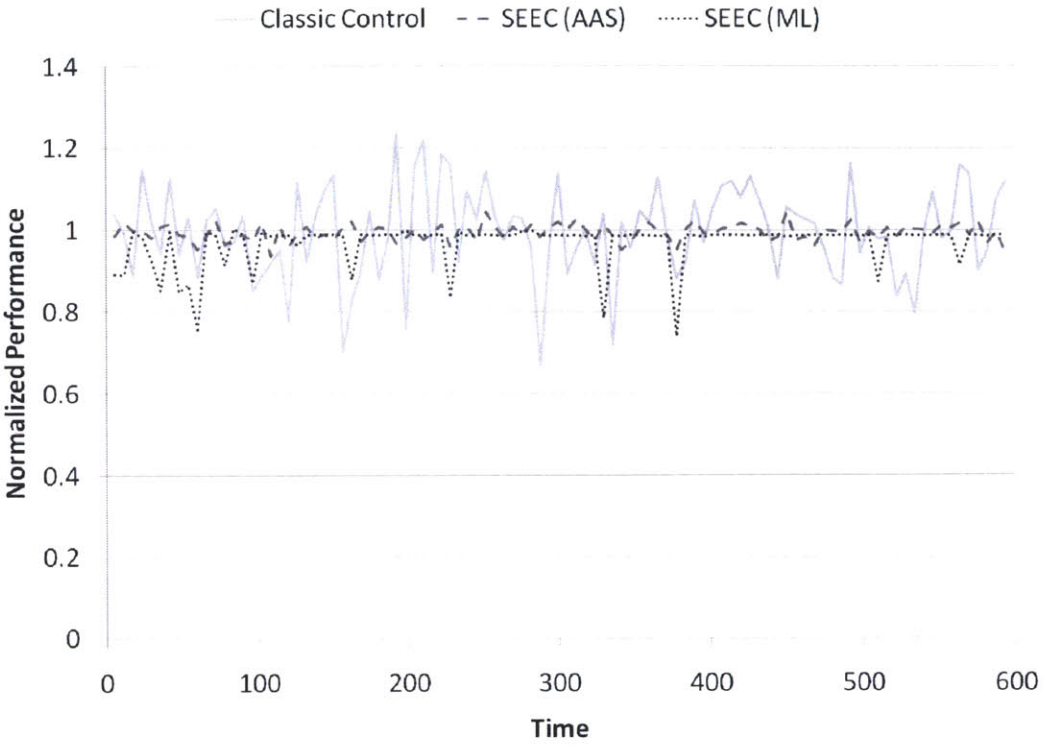


(a) Performance

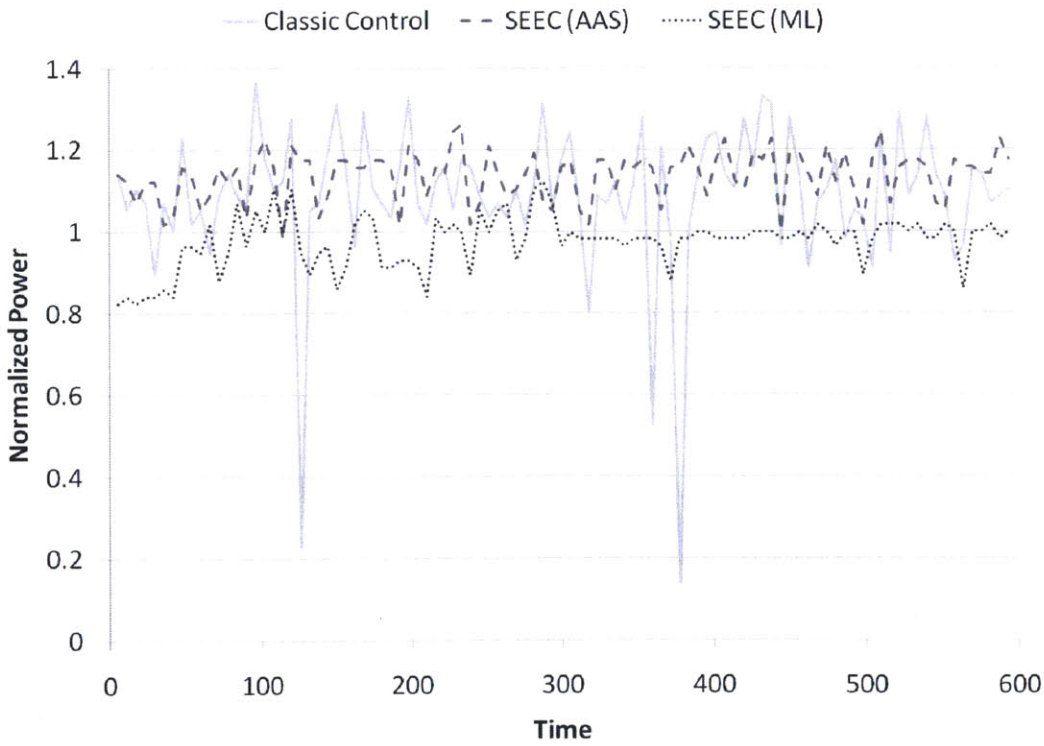


(b) Power

Figure 5-16: Learning System Models for STREAM.



(a) Performance



(b) Power

Figure 5-17: Learning System Models for dijkstra.

5.7 Adaptive Applications

This section evaluates SEEC's ability to control actions specified at the application level. Specifically, this section evaluates the adaptive applications described in Section 4.2.2. First, the performance/precision tradeoffs of each application are examined. Then, the power/precision tradeoffs are examined. Next, SEEC is shown adjusting the application in response to a power fluctuation. Finally, SEEC is shown adjusting the application in response to workload fluctuations.

5.7.1 Performance/Precision Tradeoffs

Dynamic knobs modulate power consumption by controlling the amount of computational work required to perform a given task. On a machine that delivers constant baseline performance (in this case Machine 2, with no system-level adaptations enabled), changes in computational work correspond to changes in execution time.

Figures 5-18–5-21 present the points that dynamic knobs make available in the speedup versus precision tradeoff space for each benchmark application. The points in the graphs plot the observed mean (across the training or production inputs as indicated) speedup as a function of the observed mean precision for each dynamic knob setting. Gray dots plot results for the training inputs, with black squares (connected by a line) indicating Pareto-optimal dynamic knob settings. White squares (again connected by a line) plot the corresponding points for these Pareto-optimal dynamic knob settings for the production inputs. All speedups and precision losses are calculated relative to the dynamic knob setting which delivers the highest precision (and consequently the largest execution time). We observe the following facts:

- **Effective Trade-Offs:** Dynamic knobs provide access to operating points across a broad range of speedups (up to 100 for swaptions, 4.5 for x264, and 7 for bodytrack). Moreover, precision losses are acceptably small for virtually all Pareto-optimal knob settings (up to only 1.5% for swaptions, 7% for x264, and, for speedups up to 6, 6% for bodytrack).

  For swish++, dynamic knobs enable a speedup of up to approximately a factor

Benchmark	Speedup	Precision Loss
x264	0.995	0.975
bodytrack	0.999	0.839
swaptions	1.000	0.999
swish++	0.996	0.999

Table 5.3: Correlation coefficient of observed values from training with measured values on production inputs.

of 1.5. The precision loss increases linearly with the dynamic knob setting. The effect of the dynamic knob is, however, very simple — it simply drops lower-priority search results. So, for example, at the fastest dynamic knob setting, swish++ returns the top five search results.

- **Close Correlation:** To compute how closely behavior on production inputs tracks behavior on training inputs, we take each metric (speedup and precision loss), compute a linear least squares fit of training data to production data, and compute the correlation coefficient of each fit (see Table 5.3). The correlation coefficients are all close to 1, indicating that behavior on training inputs is an excellent predictor of behavior on production inputs.

To characterize the power versus precision tradeoff space that dynamic knobs make available, we initially configure each application to run at its highest precision point on a processor in its highest power state (2.4 GHz) and observe the performance (mean time between heartbeats). The application then instructs the SEEC runtime system to maintain the observed performance. Externally, we use `cpufrequtils` to drop the clock frequency to each of the six lower-power states, run each application on all of the production inputs, and measure the resulting performance, precision loss, and mean power consumption (the mean of the power samples over the execution of the application in the corresponding power state). We verify that, for all power states, SEEC delivers performance within 5% of the target.

Figures 5-22–5-25 plot the resulting precision loss (right y axis, in percentages) and mean power (left y axis) as a function of the processor power state. For x264, the combination of dynamic knobs and frequency scaling can reduce system power

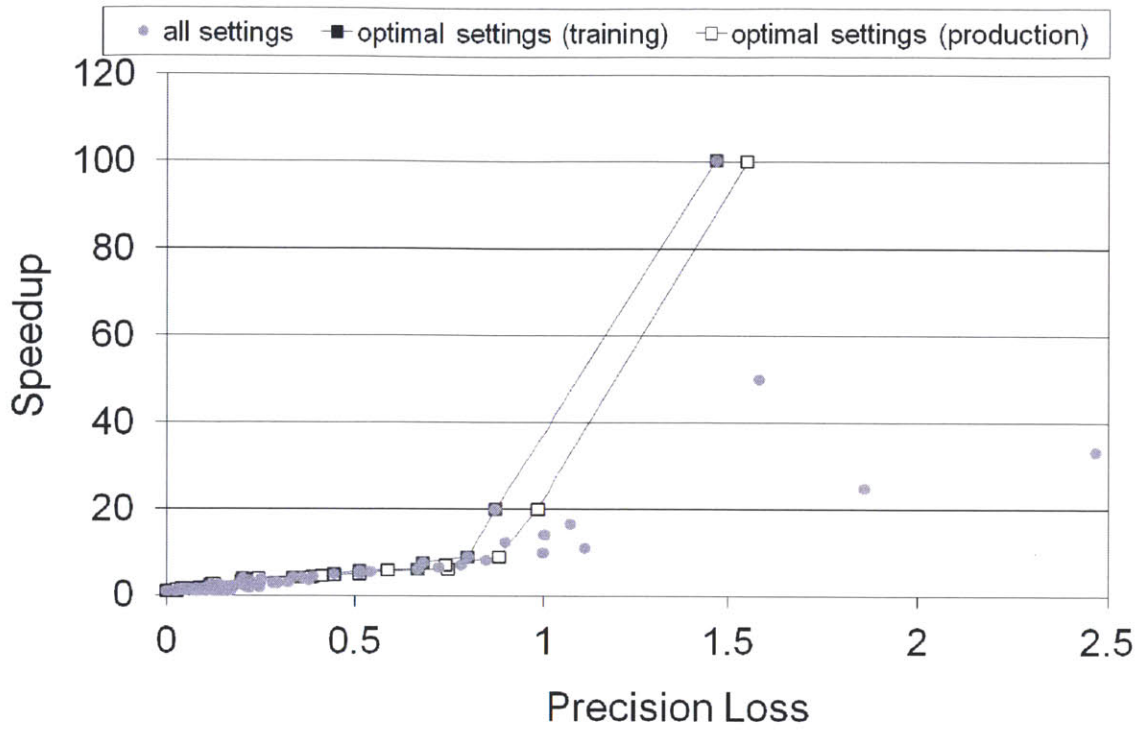


Figure 5-18: Performance versus precision for swaptions.

by as much as 21% for less than 0.5% precision loss. For bodytrack, we observe a 17% reduction in system power for less than 2.3% precision loss. For swaptions, we observe an 18% reduction in system power for less than .05% precision loss. Finally, for swish++ we observe power reductions of up to 16% for under 32% precision loss. For swish++ the dynamic knob simply truncates the list of returned results — the top results are the same, but swish++ returns fewer total results.

The graphs show that x264, bodytrack, and swaptions all have suboptimal application configurations that are dominated by other, Pareto-optimal dynamic knob settings. The exploration of the tradeoff space during training is therefore required to find good points in the tradeoff space. The graphs also show that because the Pareto-optimal settings are reasonably consistent across the training and production inputs, the training exploration results appropriately generalize to the production inputs.

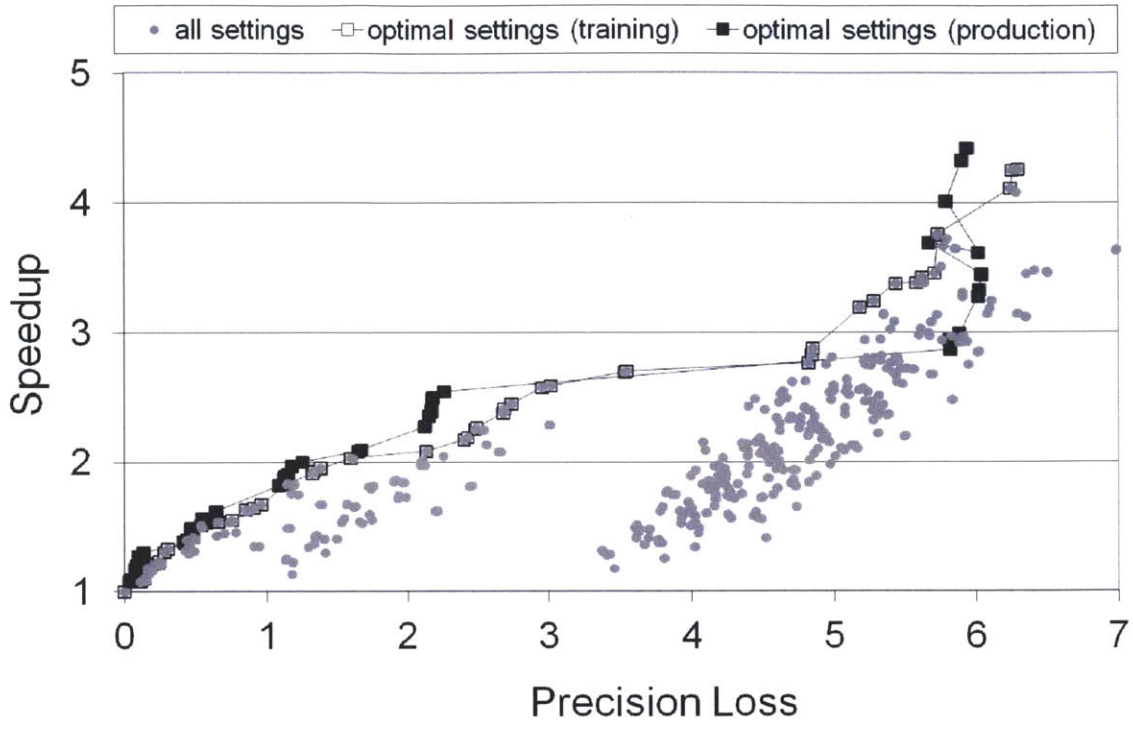


Figure 5-19: Performance versus precision for x264.

5.7.2 Adapting to Power Fluctuations

The SEEC system in combination with some set of actuators (in this case exposed through the application by a compiler) makes it possible to dynamically adapt application behavior to preserve performance (measured in heartbeats) in the face of *any* event that degrades the computational capacity of the underlying platform. We next investigate a specific scenario — the external imposition of a temporary power cap via a forced reduction in clock frequency. We first start the application running on a system with uncapped power in its highest power state (2.4 GHz). We instruct the SEEC control system to maintain the observed performance (time between heartbeats). Approximately one quarter of the way through the computation we impose a power cap that drops the machine into its lowest power state (1.6 GHz). Approximately three quarters of the way through the computation we lift the power cap and place the system back into its highest power state (2.4 GHz).

Figures 5-26–5-29 present the dynamic behavior of the benchmarks as they re-

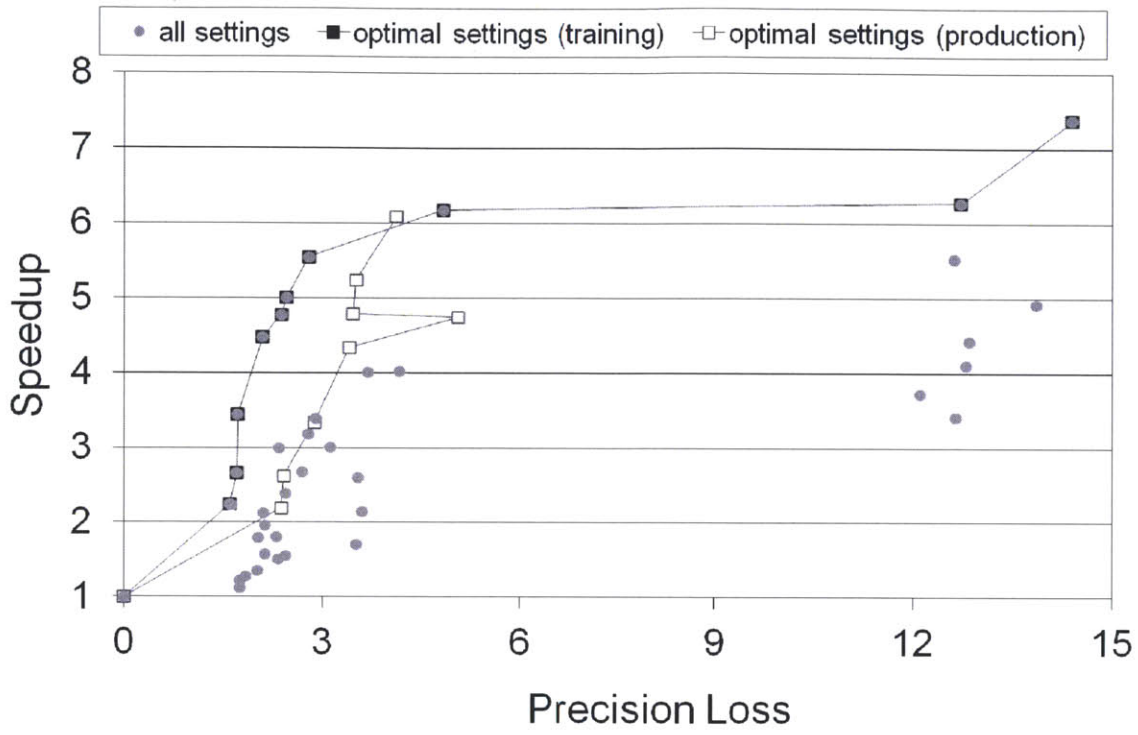


Figure 5-20: Performance versus precision for bodytrack.

spond to the power cap and corresponding processor frequency changes. Each graph plots the observed performance of the application (left y axis) as a function of time. We present the performance of three versions of the application: a version without adaptation (marked with an $\times$), a baseline version running with no power cap in place (black points), and a version that uses SEEC to preserve the performance despite the power cap (circles). We also present the "gain" or the instantaneous speedup achieved by the SEEC runtime (right y axis).

All applications exhibit the same general pattern. At the imposition of the power cap, SEEC adjusts application behavior, the gain increases (Knob Gain line), and the performance of the application first spikes down (circles), then returns back up to the baseline performance. When the power cap is lifted, SEEC adjusts again, the gain decreases, and the application performance returns to the baseline after a brief upward spike. For most of the first and last quarters of the execution, the application executes with essentially no precision loss. For the middle half of the execution, the

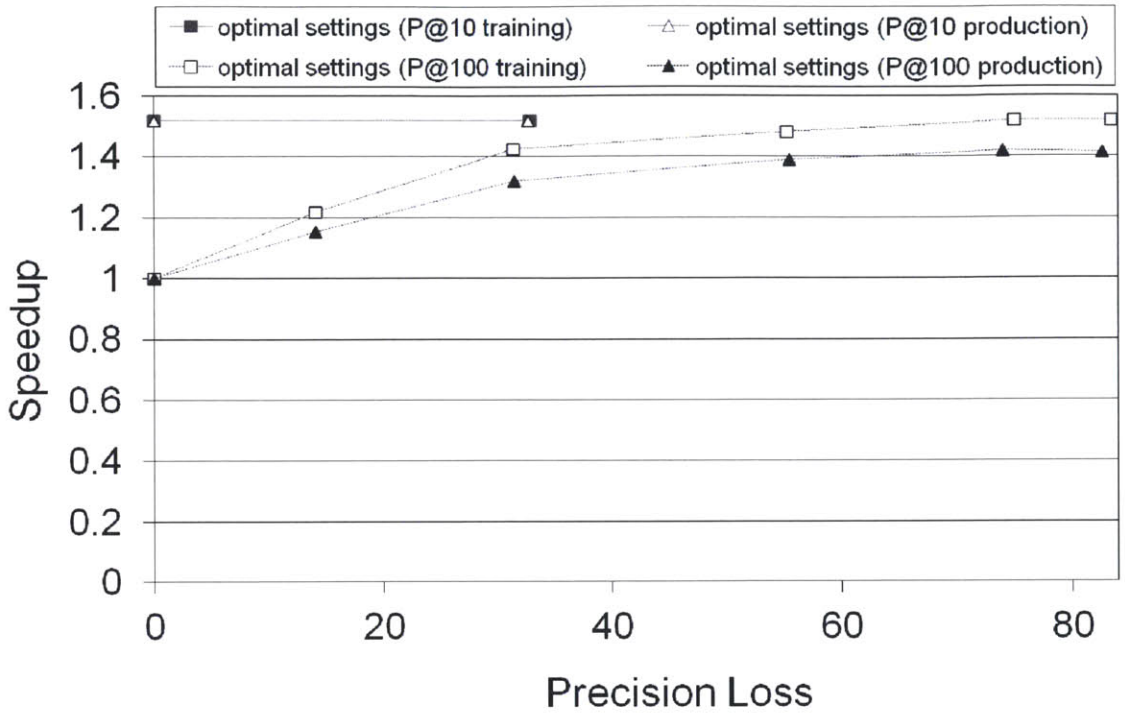


Figure 5-21: Performance versus precision for swish++.

application converges to the low power operating point plotted in Figures 5-22–5-25 as a function of the 1.6 GHz processor frequency. Without SEEC (marked with ×), application performance drops well below the baseline as soon as the power cap is imposed, then rises back up to the baseline only after the power cap is lifted.

Within this general pattern the applications exhibit varying degrees of noise in their response. Swaptions exhibits very predictable performance over time with little noise (in this scenario swaptions is run on a single core). swish++, on the other extreme, has relatively unpredictable performance over time with significant noise. x264 and bodytrack fall somewhere in between. Despite the differences in application characteristics, SEEC makes it possible for the applications to largely satisfy their performance goals in the face of dynamically fluctuating power requirements.

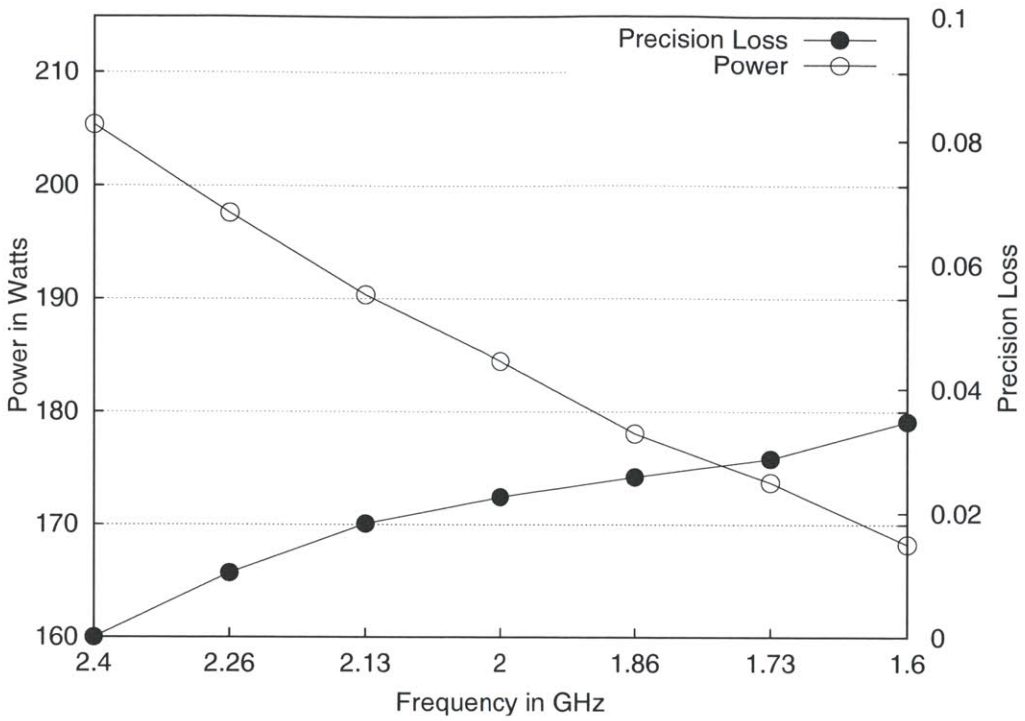


Figure 5-22: Power versus precision tradeoffs for swaptions.

5.7.3 Adapting to Workload Fluctuations

We next evaluate the use of dynamic applications to reduce the number of machines required to service time-varying workloads with intermittent load spikes, thereby reducing the number of machines, power, and indirect costs (such as cooling costs) required to maintain responsive execution in the face of such spikes:

- **Target Performance:** We set the target performance to the performance achieved by running one instance of the application on an otherwise unloaded machine.
- **Baseline System:** We start by provisioning a system to deliver target performance for a specific peak load of the applications running the baseline (default command line) configuration. For the three PARSEC benchmarks we provision for a peak load of 32 (four 8-core machines) concurrent instances of the application. For swish++ we provision for a peak load of three concurrent instances, each with eight threads. This system load balances all jobs proportionally across available machines. Machines without jobs are idle but not powered off.

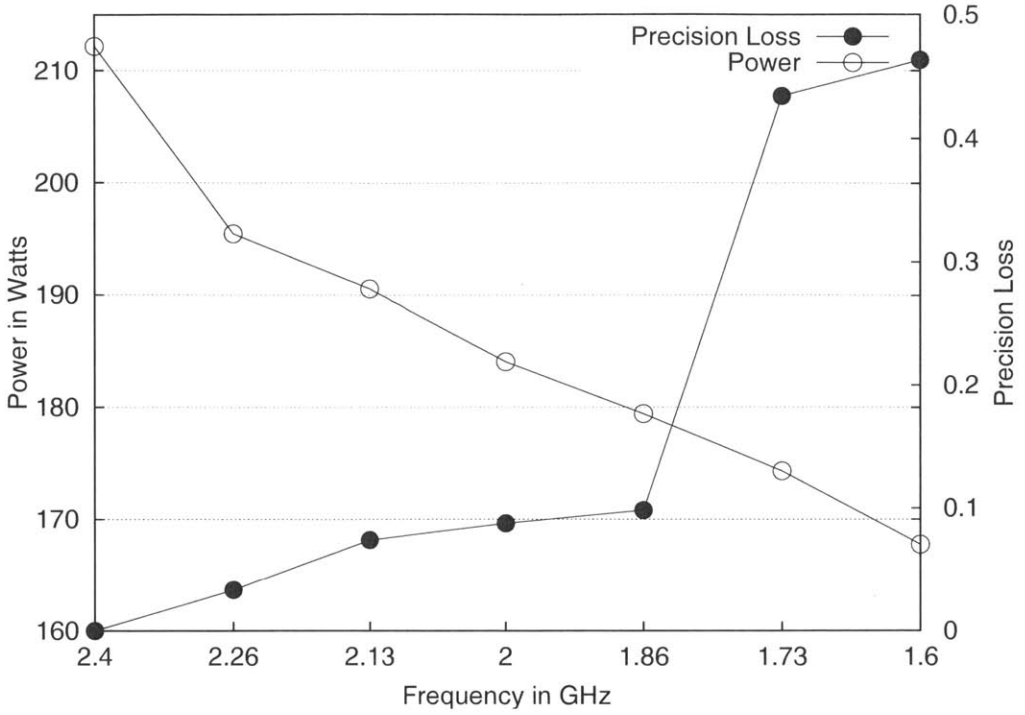


Figure 5-23: Power versus precision tradeoffs for x264.

- **Consolidated System:** We impose a bound of either 5% (for the PARSEC benchmarks) or 30% (for swish++) precision loss. We then provision the minimum number of machines required for SEEC to provide baseline performance at peak load subject to the precision loss bound. For the PARSEC benchmarks we provision a single machine. For swish++ we provision two machines.
- **Power Consumption Experiments:** We then vary the load from 0% utilization of the original baseline system (no load at all) to 100% utilization (the peak load). For each load, we measure the power consumption of the baseline system (which delivers baseline precision at all utilizations) and the power consumption and precision loss of the consolidated system (which uses SEEC to deliver target performance). At low utilizations, SEEC will configure the applications to deliver maximum precision. As the utilization increases, SEEC will progressively manipulate the applications to maintain the target performance at the cost of some precision loss.

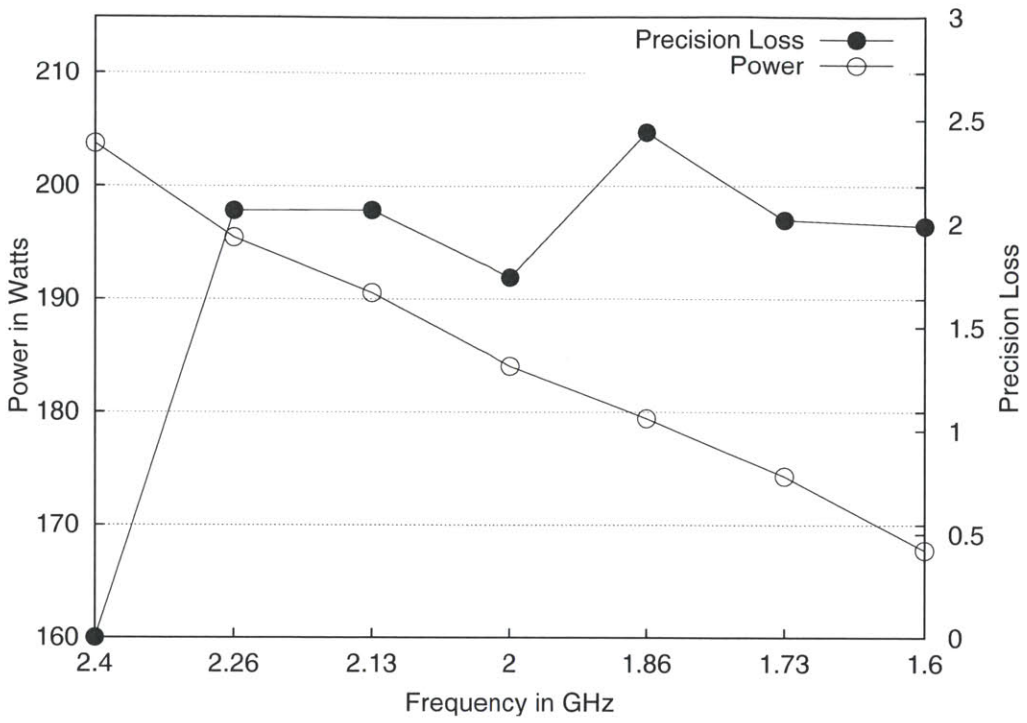


Figure 5-24: Power versus precision tradeoffs for bodytrack.

Figures 5-30–5-33 present the results of these experiments. Each graph plots the mean power consumption of the original (circles) and consolidated (black dot) systems (left y axis) and the mean precision loss (solid line, right y axis) as a function of system utilization (measured with respect to the original, fully provisioned system). These graphs show that using adaptive applications and SEEC to consolidate machines can provide considerable power savings across a range of system utilization. For each of the PARSEC benchmarks, at system utilization of 25%, consolidation can provide an average power savings of approximately 400 Watts, a reduction of 66%. For swish++ at 20% utilization, we see a power savings of approximately 125 Watts, a reduction of 25%. These power savings come from the elimination of machines that would be idle in the baseline system at these utilization levels.

Of course, it is not surprising that reducing the number of machines reduces power consumption. A key benefit of the SEEC response mechanism is that even with the reduction in computational capacity, it enables the system to maintain the same performance at peak load while consuming significantly less power. For the PAR-

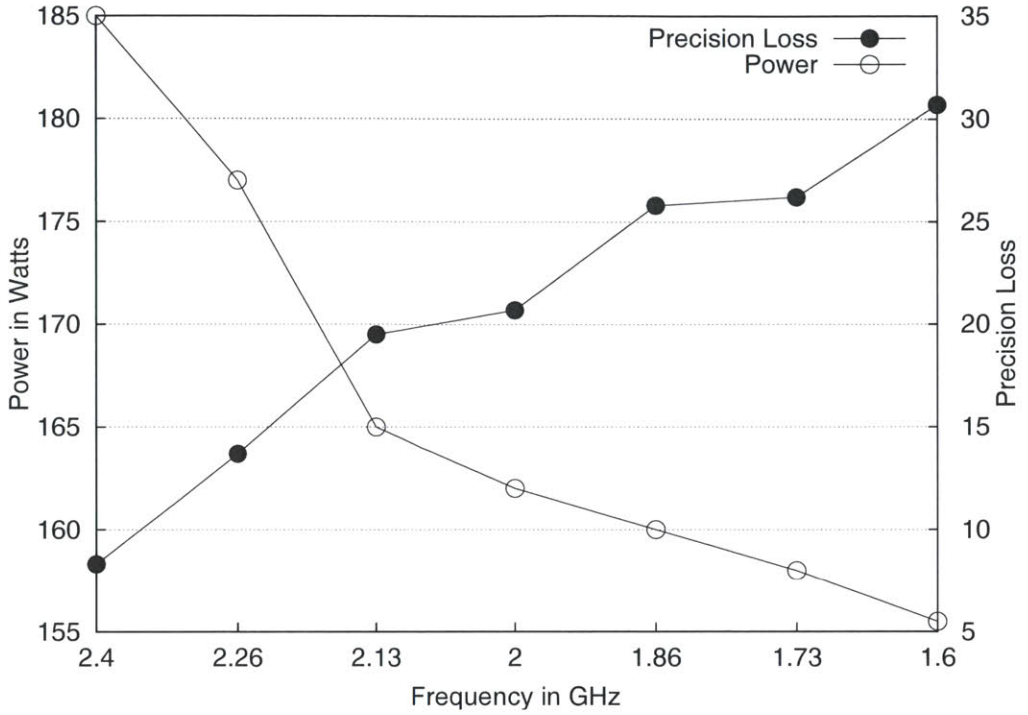


Figure 5-25: Power versus precision tradeoffs for swish++.

SEC benchmarks at a system utilization of 100%, the consolidated systems consume approximately 75% less power than the original system while providing the same performance. For swish++ at 100% utilization, the consolidated system consumes 25% less power.

The consolidated systems save power by automatically reducing precision to maintain performance. For swaptions, the maximum precision loss required to meet peak load is 0.004%, for x264 it is 7.6%, and for bodytrack it is 2.5%. For swish++ with P@10, the precision loss is 8% at a system utilization of 65%, rising to 30% at a system utilization of 100%. We note, however, that the majority of the precision loss for swish++ is due to a reduction in recall — top results are generally preserved in order but fewer total results are returned. The top results are not affected by the change in application behavior unless the P@N is less than the current setting. As the lowest setting used by SEEC is five, the order is always perfect for the top 5 results.

For common usage patterns characterized by predominantly low utilization punctuated by occasional high-utilization spikes [8], these results show that SEEC com-

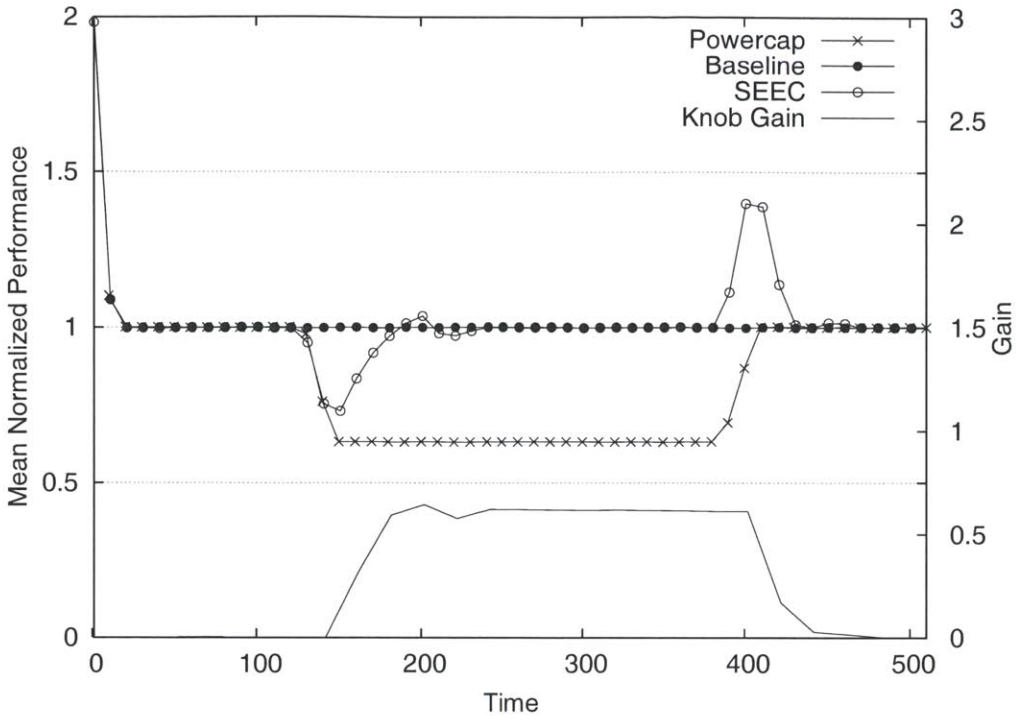


Figure 5-26: Behavior of swaptions with SEEC in response to power cap.

bined with the complier framework from Section 4.2.2 can substantially reduce overall system cost, deliver the highest (or close to highest) precision in predominant operating conditions, and preserve performance and acceptable precision even when the system experiences intermittent load spikes.

5.8 Controlling Multiple Applications

This experiment demonstrates SEEC using both system and application-level actions to manage multiple applications in response to a fluctuation in system resources. In this scenario, SEEC uses the adaptive version of x264 (described in Section 4.2.2) and the statically configured version of bodytrack that is available through the PARSEC benchmarks. Both applications are simultaneously launched on Machine 1 and request a performance of half the maximum achievable, so the system has just enough capacity to meet these goals. x264 is given lower-priority than bodytrack. In addition, x264 indicates a preference for system-level adaptations indicating that SEEC should only

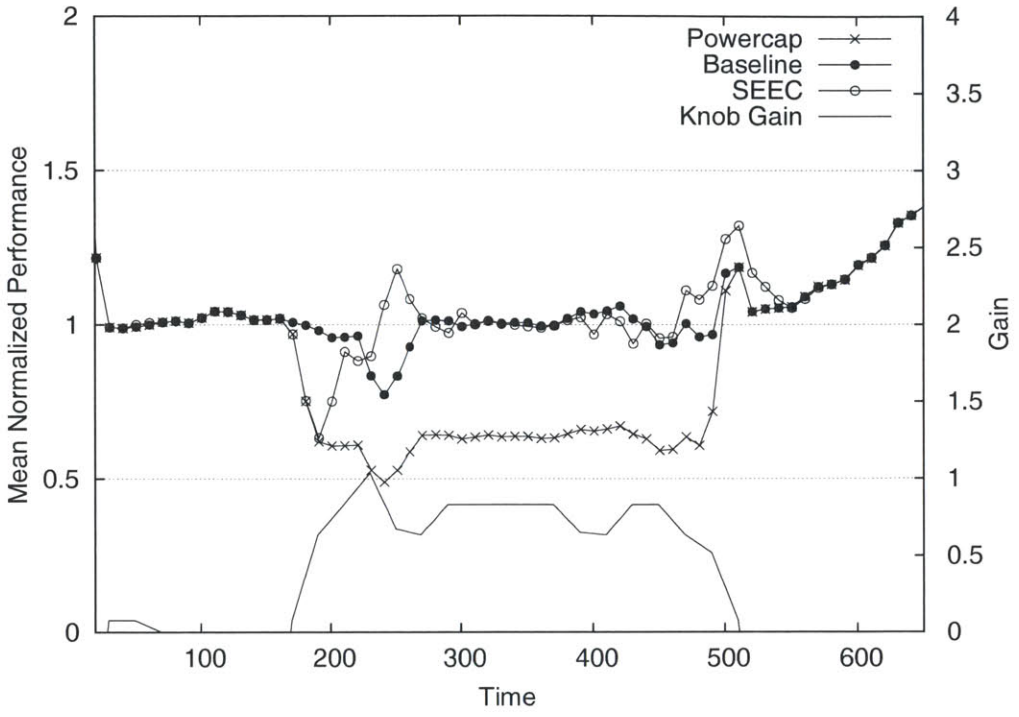


Figure 5-27: Behavior of x264 with SEEC in response to power cap.

take application-level actions if it has exhausted all system-level ones.

Approximately 10% of the way through execution we simulate a thermal emergency as might occur if the chip is in danger of overheating. In response, the hardware lowers the processor frequency to its lowest setting. To simulate this situation, we force Machine 1's clock speed to its minimum and disable the actions that allow SEEC to change this value. Doing so forces the SEEC decision engine to adapt to try to meet performance despite the loss of processing power and the fact that some of its actions no longer have the anticipated effect. Online adaptation to the removal of actions would not be possible with prior control systems such as ControlWare [95] and METE [79], but is possible with SEEC.

5.8.1 Results

Figures 5-34(a) and 5-34(b) illustrate the behavior of SEEC AAS in this scenario, where Figure 5-34(a) depicts bodytrack's response and Figure 5-34(b) shows that of x264. Both figures show performance (normalized to the target performance) on the

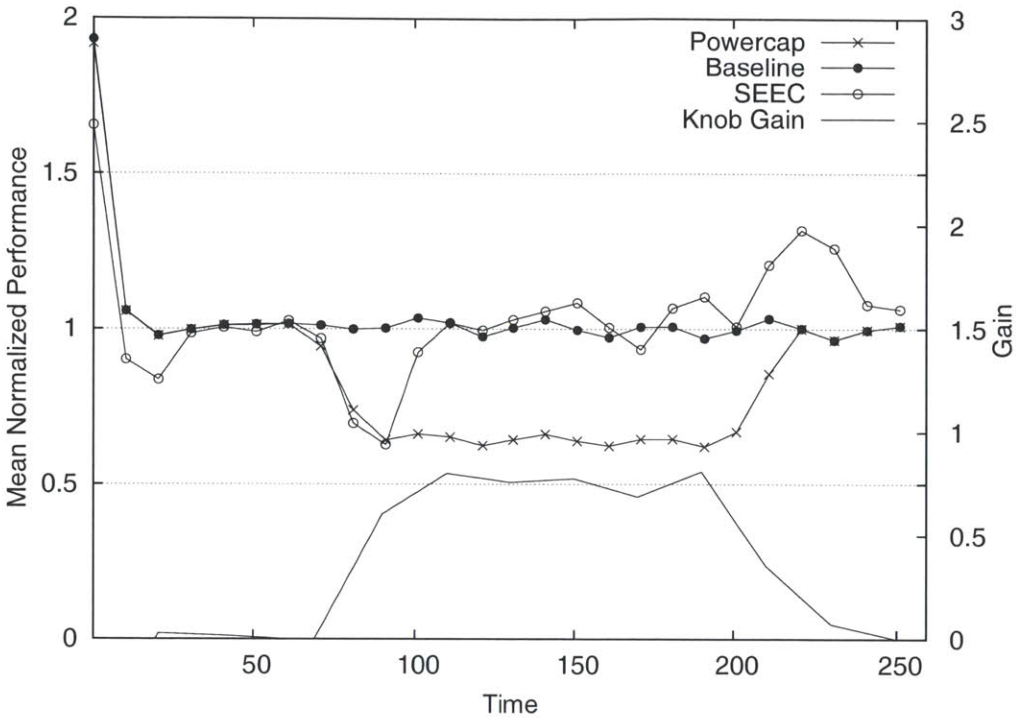


Figure 5-28: Behavior of bodytrack with SEEC in response to power cap.

left y-axis and time (measured in heartbeats) on the x-axis. The time where frequency changes is shown by the solid vertical line. For each application, performance is shown with clock frequency changes but no adaptation ("No adapt"), and with SEEC adapting to clock frequency changes using both AAS and ML.

Figure 5-34(a) shows that SEEC AAS maintains bodytrack's performance despite the loss in compute power. SEEC observes the clock speed loss as a reduction in heart rate and deallocates two cores from the lower-priority x264, assigning them to bodytrack. Without SEEC bodytrack would only achieve 65% of its desired performance, but with SEEC bodytrack meets its goals. SEEC ML also can bring bodytrack back to its desired performance, but it takes longer and is done at a cost of oscillation as the ML algorithm explores different actions.

Figure 5-34(b) shows how SEEC sacrifices x264's performance to meet the needs of bodytrack. SEEC deallocates cores from x264 but compensates for this loss by altering x264's algorithm. By managing both application- and system-level adaptations, SEEC is able to resolve resource conflicts and meet both application's goals. We note

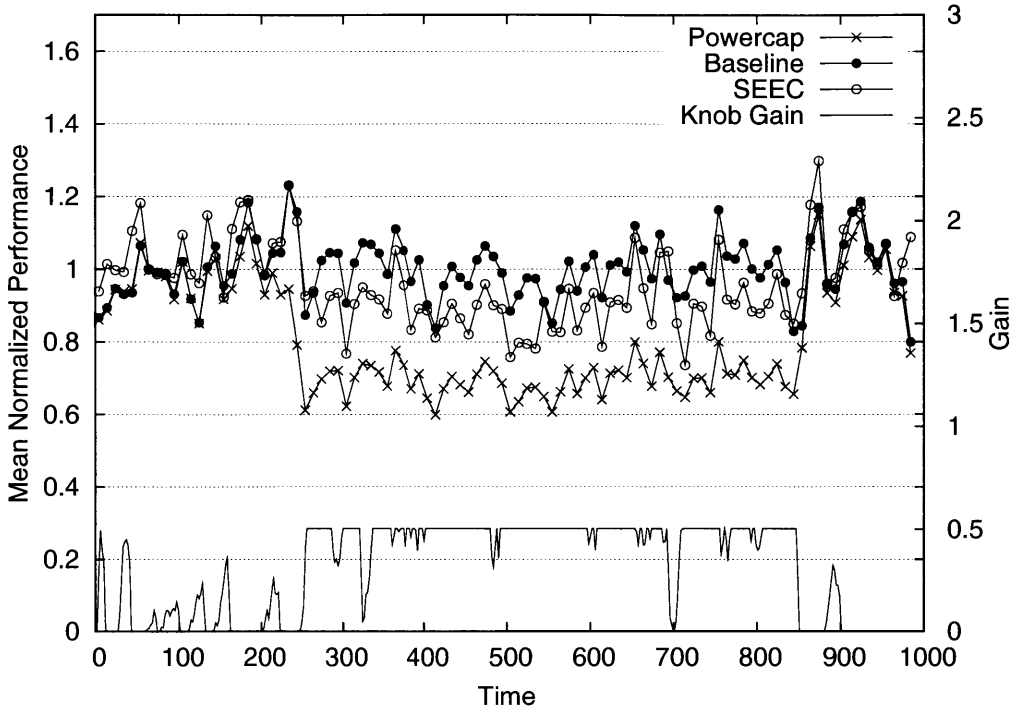


Figure 5-29: Behavior of swish++ with SEEC in response to power cap.

that if x264 had been the high-priority application, SEEC would not have changed its algorithm because x264 requests system-level adaptations before application-level ones. In this case, SEEC would have assigned x264 more processors and bodytrack would not have met its goals. As with bodytrack, SEEC AAS is able to adapt more quickly than SEEC ML, but both approaches converge to the desired value.

This study shows SEEC controlling multiple applications, some of which are themselves adaptive. This is possible because SEEC's decoupled implementation allows application and system adaptations to be specified independently. In addition, SEEC can automatically adapt to fluctuations in the environment by directly observing application performance and goals. SEEC does not detect the clock frequency change directly, but instead detects a change in the applications' heart rates, and SEEC can respond to any change that alters the performance of the component applications.

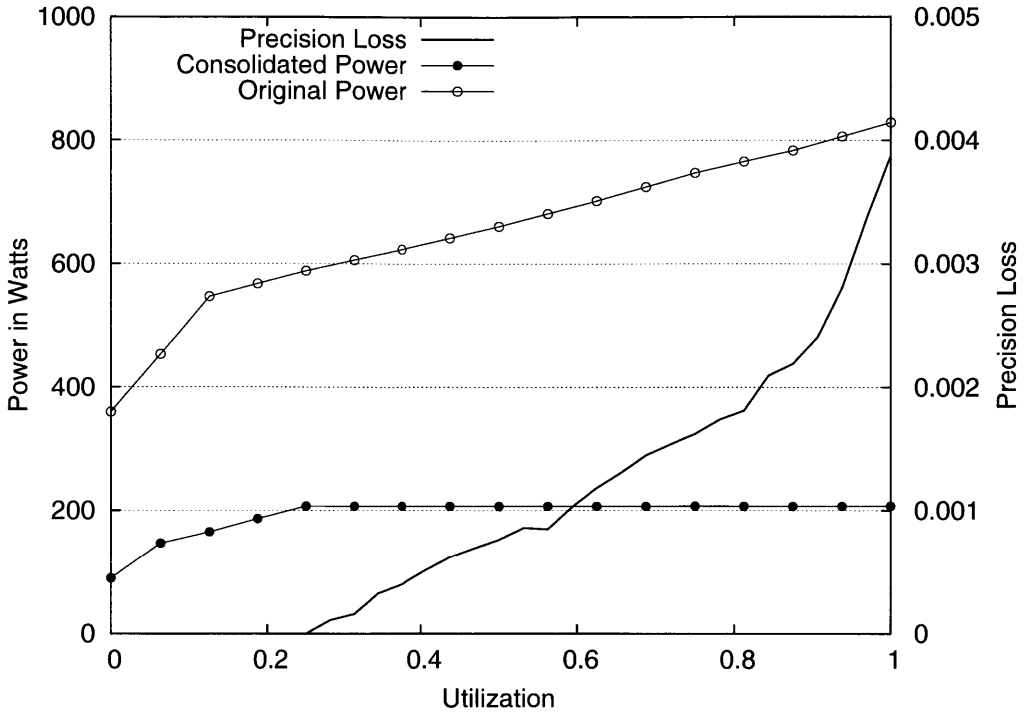


Figure 5-30: Using SEEC and dynamic swaptions for system consolidation.

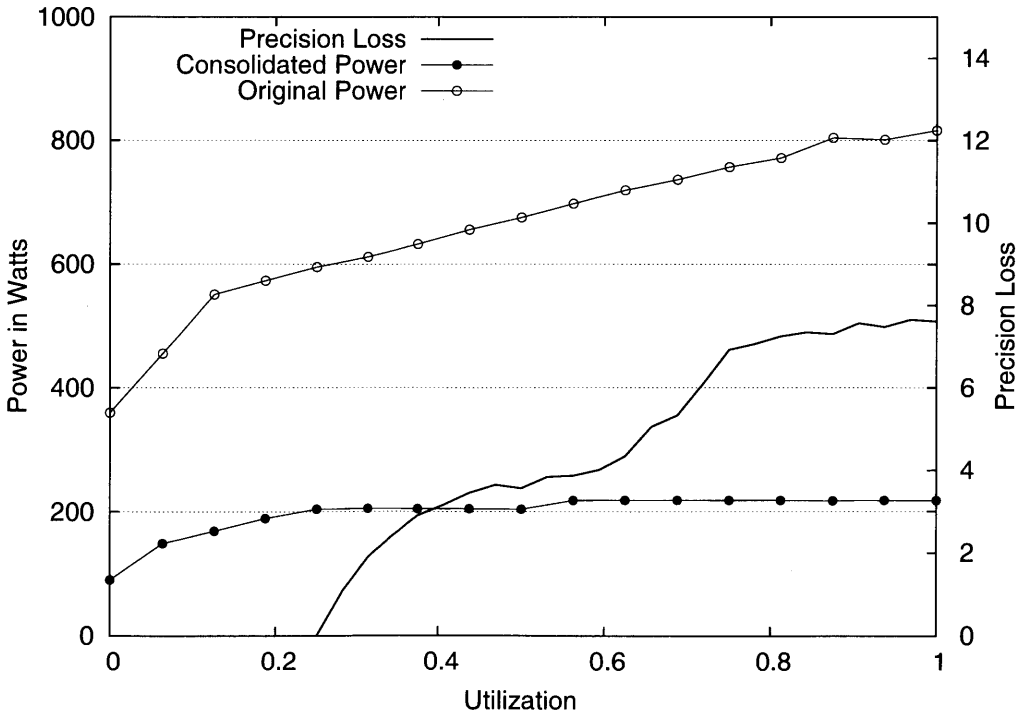


Figure 5-31: Using SEEC and dynamic x264 for system consolidation.

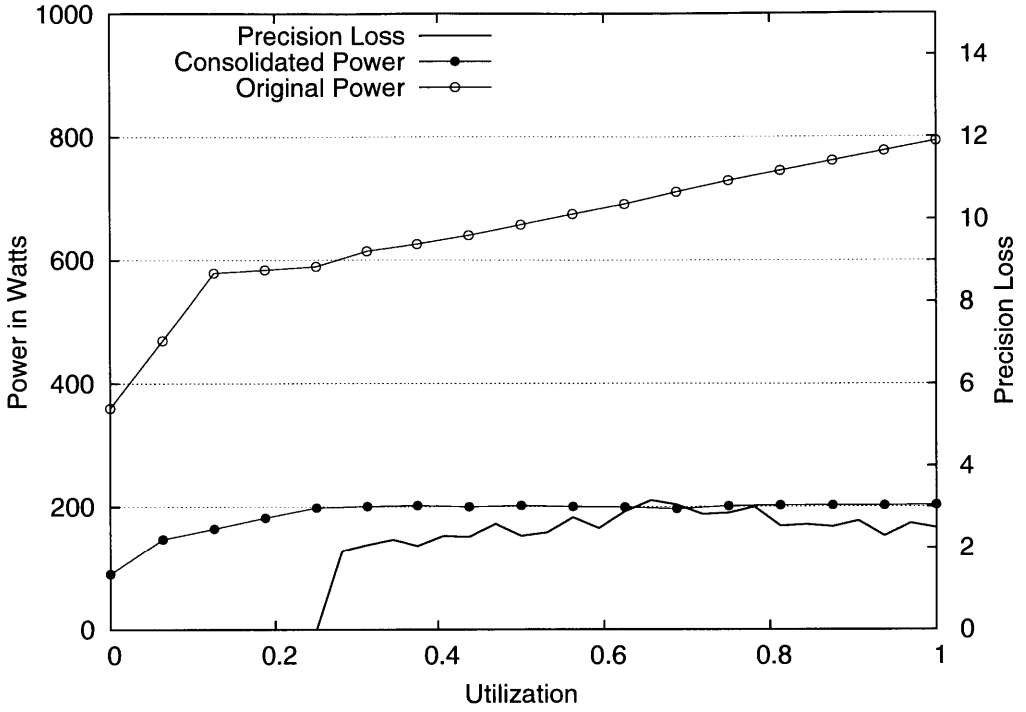


Figure 5-32: Using SEEC and dynamic bodytrack for system consolidation.

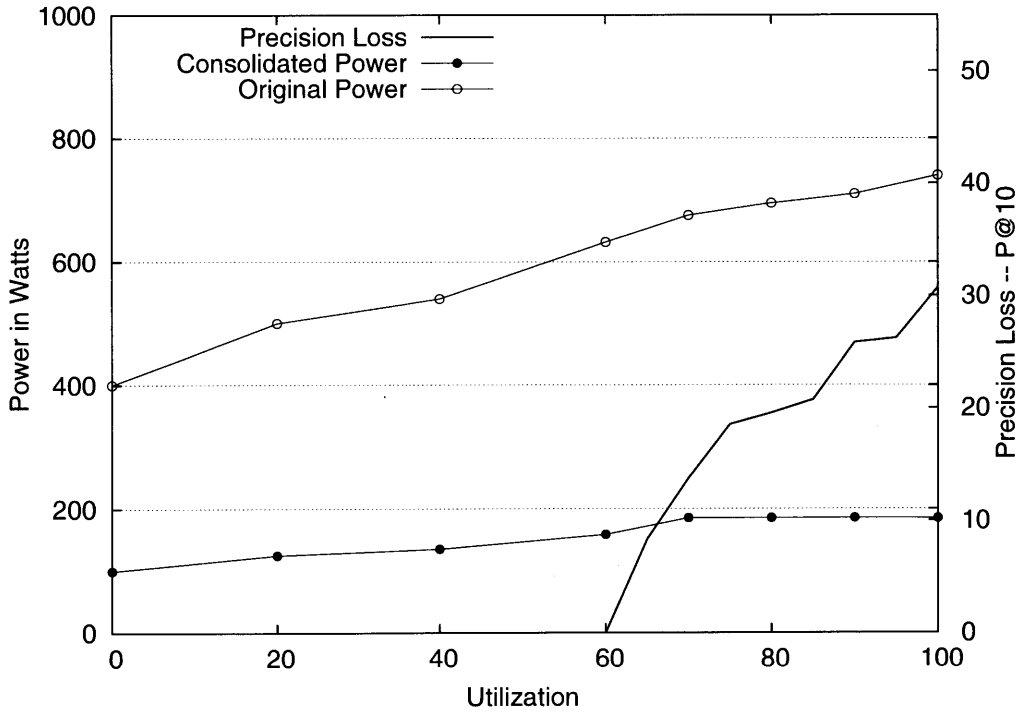
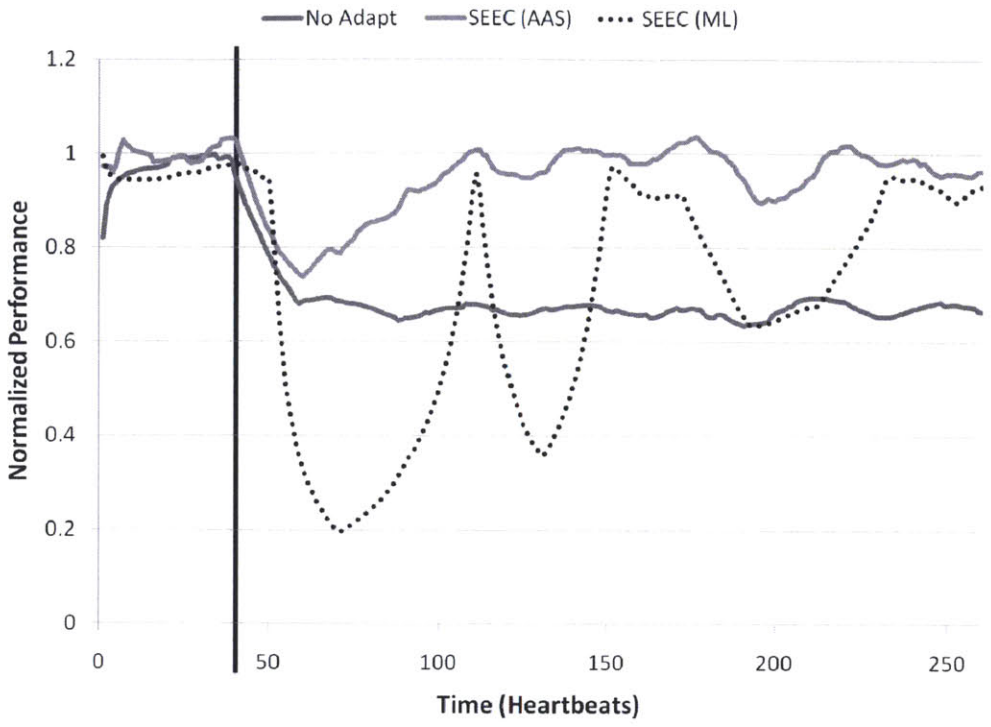
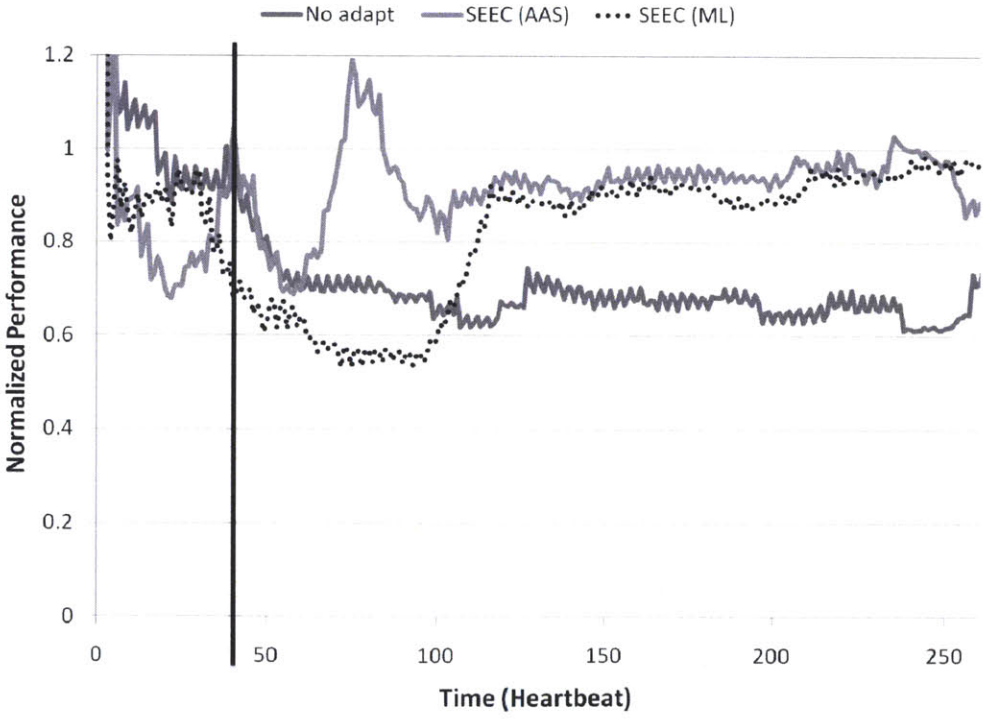


Figure 5-33: Using SEEC and dynamic swish++ for system consolidation.



(a) bodytrack



(b) x264

Figure 5-34: SEEC responding to clock speed changes.

Chapter 6

Properties of SEEC

SEEC automatically makes decisions that affect the behavior of applications and systems in an attempt to drive the system toward application-level goals. This design approach raises some questions about what guarantees SEEC can provide about how it drives the system to meet goals and this section discusses those guarantees.

As discussed in Chapter 3, the SEEC decision engine consists of multiple layers of adaptation. The lowest level, classical control, provides the most guarantees about its behavior, but achieves those guarantees through a set of strict assumptions. Additional layers of adaptation relax some of these assumptions, allowing for greater flexibility. Thus, each layer of the SEEC decision engine presents a tradeoff in the guarantees about its behavior and the flexibility to adapt as illustrated in Figure 6-1. As will be shown in this section, the classical control system presents the strongest guarantees and the least flexible set of assumptions. In contrast, the reinforcement learner presents the most flexible decision mechanism at the cost of providing the fewest guarantees.

In particular, this work is concerned with characterizing SEEC's behavior in terms of five desirable properties. The first four properties are the SASO properties: stability, accuracy, settling time, and maximum overshoot) [36]. The fifth property is efficiency. This section begins by defining the properties under analysis, and then describes the guarantees that each level of the decision engine provides in terms of these properties.

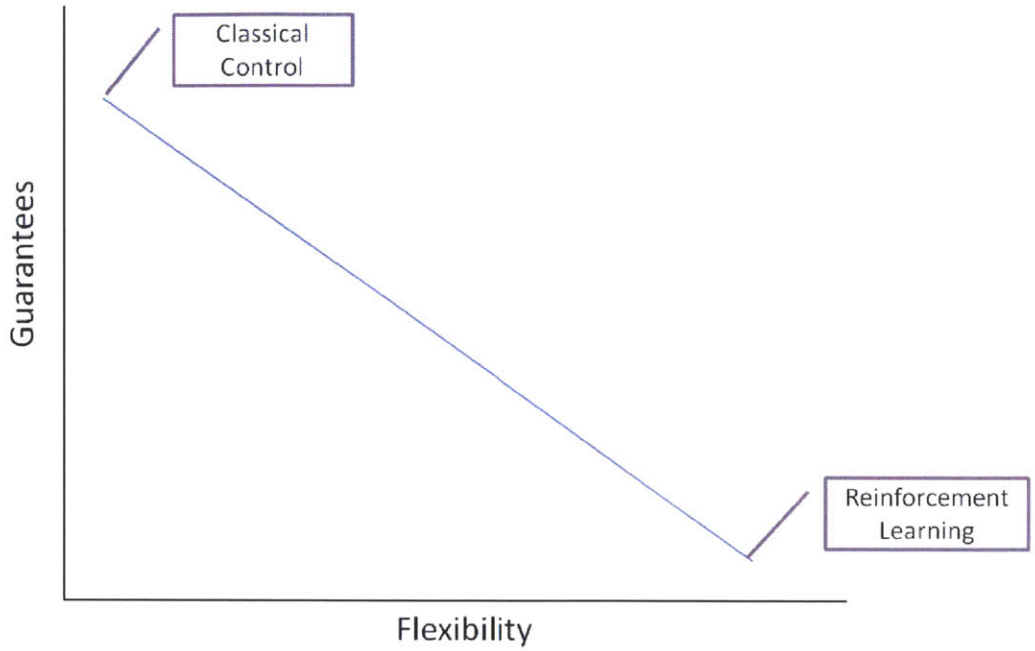


Figure 6-1: Different features of the SEEC decision engine provide different tradeoffs between the guarantees they provide and their flexibility to adjust to violations in assumptions.

6.1 Definitions of Properties

This section defines the properties of stability, accuracy, settling time, maximum overshoot, and efficiency. These properties describe the behavior of the SEEC decision engine as it drives the system to a goal. The first four properties (SASO) describe the behavior of the system in the goal dimension, while the fifth property (efficiency) describes the behavior of the system in a free dimension. The SASO properties are illustrated in Figure 6-2. For example, an application may express a performance goal while a system developer may specify actions that change the performance/power tradeoff space. In this case, the SASO properties will describe application performance, while efficiency will describe the behavior of SEEC in the power dimension (e.g.,, Figures 5-8–5-9). Alternatively, if the application had a power goal, the SASO properties describe the behavior in the power dimension, while efficiency describes the behavior in the performance dimension (e.g.,, Figures 5-10–5-11).

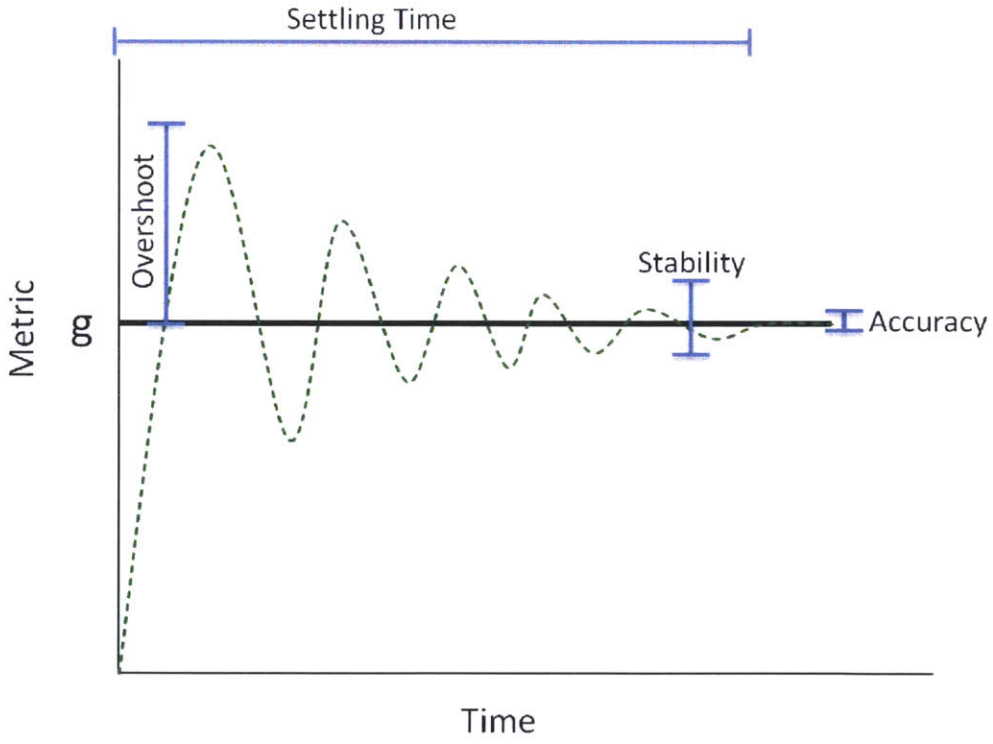


Figure 6-2: Illustration of the SASO (stability, accuracy, settling time, and overshoot) properties.

6.1.1 Stability

Stability refers to the change in the target metric over time. A *stable* system converges to a single value; i.e.,the derivative of the signal eventually becomes zero. The remaining SASO properties are defined in terms of a stable system. For example, if an application has a performance goal, stability refers to the property that the performance converges to a steady value.

6.1.2 Accuracy

An accurate system is a stable system that converges to the target value. This is the same property referred to in the introduction and demonstrated in Chapter 5. This is one of the most important properties of a system built with SEEC because guaranteeing this property means guaranteeing that the goals are met. For example, if an application has a performance goal and performance target g, accuracy refers to the property that the performance converges to g.

6.1.3 Settling Time

Settling time refers to the time that passes from system startup to the point where the system becomes stable.

6.1.4 Max Overshoot

The maximum overshoot refers to the largest amount by which SEEC might fail to miss the target on its way to becoming stable.

6.1.5 Efficiency

The first four properties describe the behavior of the system in the target dimension. Efficiency characterizes behavior in the dimension of freedom. Specifically, efficiency measures how far that dimension is from the best possible configuration that accurately meets the goal.

6.2 Properties of the Classical Control System

We begin by analyzing the properties of the classical control system. This section follows the framework presented by Hellerstein et al for analyzing the properties of a control system [36] with a focus on the unique features of the SEEC system.

6.2.1 Assumptions

This analysis begins with the following assumptions; subsequent sections will relax these assumptions:

Assumption 1 *The application produces a stable heart rate signal; i.e.,application performance converges to a steady value.*

These properties assume that the application is, itself stable. If the application's behavior does not converge without SEEC, SEEC will not correct this issue. The application can be composed of phases, however; and in this case each phase must converge to a stable value.

Assumption 2 *The desired speedup lies between the maximum and minimum achievable speedups made available through various actuators.*

If this assumption is violated, it is impossible to achieve the desired target, so it does not make sense to describe SEEC's behavior on the way to the target.

Assumption 3 *For any application under control, the workload — w in Equation 3.4 — is 1) known and 2) constant.*

This assumption is reasonable for any application-specific system where the application can be profiled ahead of time and the application's performance does not vary based on input.

Assumption 4 *Any set of actions that achieves the desired speedup has an equivalent cost.*

This assumption is reasonable for systems with a single actuator. In this case, it is easy to find Pareto-optimal settings whose use makes the assumption valid.

Assumption 5 *The speedup of all actions available in the system and specified through the system programmer API are 1) known and 2) constant.*

This assumption is reasonable for application-specific systems where the application response to a specific knob can be profiled ahead of time. This assumption may also be reasonable for classes of applications and actuators that are highly predictable; e.g.,for a known class of compute-bound applications.

6.2.2 Properties

Stability Stability is determined by analyzing the transfer function of the system as presented in Equation 3.3 and reproduced here:

$$F_i(z) = \frac{(1-p_1)(1-p_2)}{1-z_1} \frac{z-z_1}{(z-p_1)(z-p_2)}$$

The stability of the system is dependent on the specific values used to instantiate the controller. In particular if $|p_1|, |p_2| < 1$ then the system is stable, and thus converges to a steady performance.

Accuracy Accuracy measures the error in the steady-state performance — $|e_i(t)|$ in Equation 3.2. Application i converges to the target performance when $e_i(t) = 0$ for all $t > t_{steady}$, where t_{steady} represents the time at which the system reaches steady state. The accuracy can be determined by computing the gain of the transfer function. Zero error is achieved if and only if the steady state gain of the transfer function is unity. The steady state gain can be analyzed simply by evaluating the transfer function (Equation 3.3) at $z = 1$. Clearly, SEEC's transfer function has been constructed to assure accuracy as $F(1) = 1$ for any values of p_1, p_2 and z_1.

Settling time Settling time is a function of the poles of Equation 3.3, and specifically depends on the dominant pole: $\max(|p_1|, |p_2|)$. The settling time of the system

can be approximated by [36]:

$$t_{settle} \approx \frac{-4}{\log \max(|p_1|, |p_2|)} \tag{6.1}$$

The choice of p_1 and p_2 has an interesting effect on the behavior of SEEC systems. For maximum response time, both values can be set to 0 causing SEEC to react almost instantly to changes in behavior. While this agility can be beneficial in many scenarios, it might not be desirable when the system is noisy and prone to disturbance. Disturbance is modeled as δh_i in Equation 3.1. To reject larger disturbances, the system should be initialized with $\max(|p_1|, |p_2|)$ close to 1 (but not equal to ensure stability). For examples of system behavior with larger dominant poles, see the line marked "slow convergence" in Figures 5-1–5-5.

Maximum Overshoot As with settling time, the maximum overshoot O_{max} is derived from the poles. Specifically, it is approximated by:

$$O_{max} = \begin{cases} 0 & \text{if } \max(p_1, p_2) \geq 0 \\ |\max(p_1, p_2)| & \text{if } \max(|p_1|, |p_2|) < 0 \end{cases} \tag{6.2}$$

Efficiency Efficiency for the classical control system is trivial under Assumption 4. This assumption states that all methods of achieving a given speedup are assumed to have the same costs. In practice, it assumes that there is only one method for obtaining a given speedup.

6.2.3 Results

Although SEEC is evaluated empirically in Chapter 5, it is informative to examine the behavior of the classical control system in isolation to examine the practical effects of the assumptions needed to provide these guarantees.

Figure 6-3 depicts data from the experiment described in Section 5.4.3. Specifically, these results show SEEC's classical control system managing a performance

goal and attempting to minimize power on machines 1 and 2. Here, the y-axis shows performance per Watt normalized to the static oracle, while the x-axis shows the results for each benchmark. The bar shows the performance of the classical control system.

The results shown in Figure 6-3 are not particularly good. On average, the classical control system only achieves 60% of what the static oracle can accomplish. The reason for the poor results lies primarily in Assumption 3 and Assumption 4. Following Assumption 3, the classical control system uses a fixed value for the workload and the same controller then has to manage every application. In the best case, the wrong value of workload will increase the settling time of the system. In the worst case, the system will not stabilize. In fact, in Figure 2-2 we have already seen one example of how the wrong value of workload can destabilize the system. In either case, the system will be inefficient.

6.3 Properties of Adaptive Control

6.3.1 Assumptions

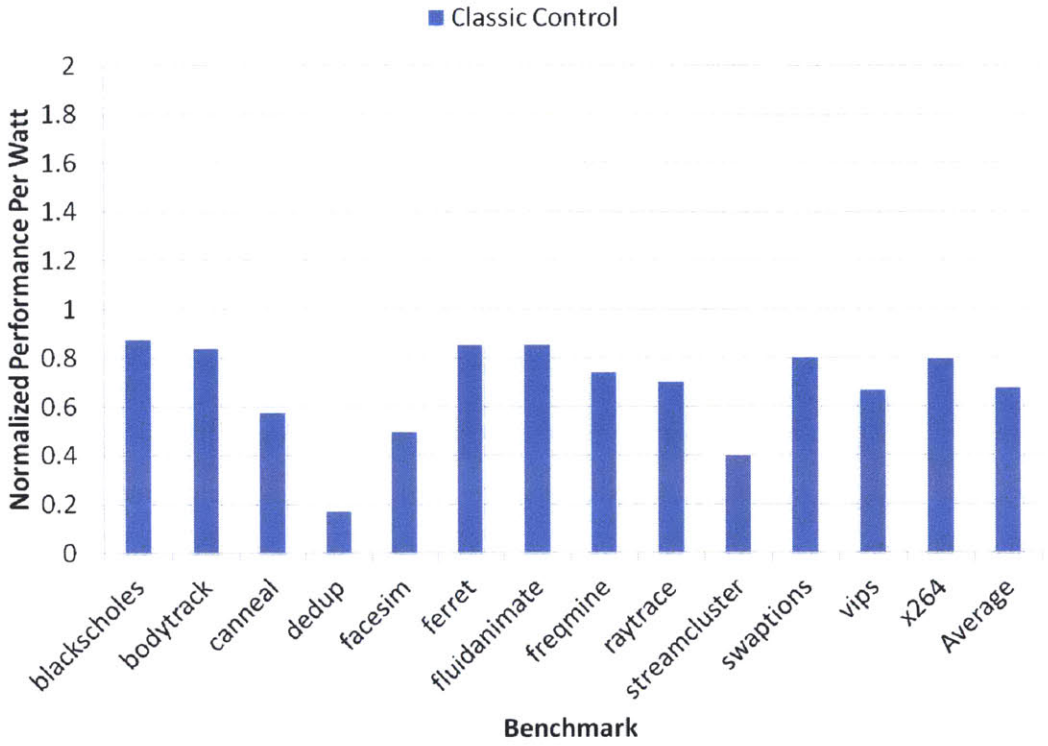
To try to improve on the classical control system we relax Assumption 3. We replace this with the following weaker assumption:

> **Assumption 6** *The process noise, i.e.,the variance in i application's heart rate signal $q_i(t)$, is bounded above and below. Therefore,*
>
> $$q_{min} \leq q_i(t) \leq q_{max}, \quad \forall t \tag{6.3}$$

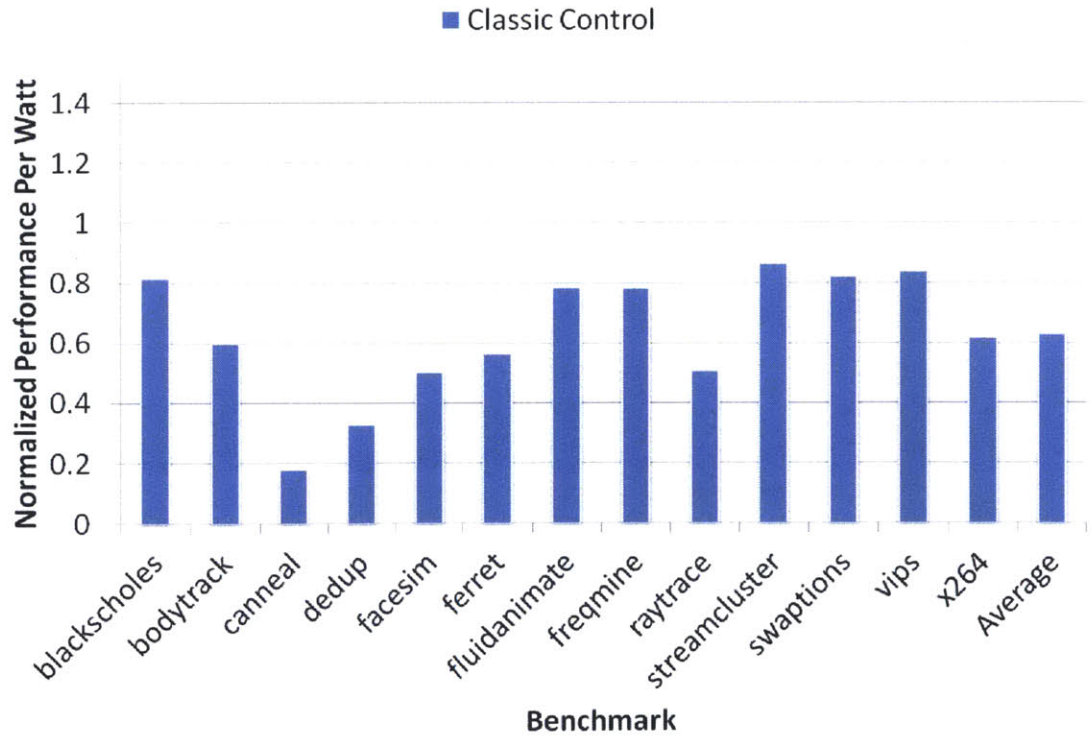
This assumption indicates that the variance in the application is bounded.

From Assumption 5 we can conclude that the maximum speedup is bounded by a constant:

$$s_i(t) \leq s_{max}, \quad \forall t \tag{6.4}$$



(a) Machine 1



(b) Machine 2

Figure 6-3: The classic control system does not work well when confronted with a range of different applications.

Furthermore, if we require SEEC to hold the speedup constant until the Kalman filter's change has reduced below some threshold, we can assume that the sequence of speedups $s_i(t)$ is persistently excitable, i.e.,:

$$a \leq s_i(t)^2 \leq b = s_{max}^2 \tag{6.5}$$

6.3.2 Properties

To describe the properties of adaptive control, the convergence of the Kalman Filter is demonstrated. Then, the effects of the Kalman filter on the classical control system are described.

The convergence of SEEC's Kalman filter is demonstrated following the process first presented by Cao and Schwartz [15]. While the Cao and Schwartz technique is general for the family of Kalman Filters, in this work we focus on the particular features of the SEEC Kalman filter formulation.

To begin, we reproduce SEEC's Kalman filter here (from Equation 3.6):

$$\begin{aligned}
\hat{x}_i^-(t) &= \hat{x}_i(t-1) \\
p_i^-(t) &= p_i(t-1) + q_i(t) \\
k_i(t) &= \frac{p_i^-(t)s_i(t-1)}{[s_i(t)]^2 p_i^-(t) + o_i} \\
\hat{x}_i(t) &= \hat{x}_i^-(t) + k_i(t)[h_i(t) - s_i(t-1)\hat{x}_i^-(t)] \\
p_i(t) &= [1 - k_i(t)s_i(t-1)]p_i^-(t)
\end{aligned}$$

To describe the behavior of Equation 3.6 we first define the error of the Filter at time t as $\tilde{x}(t)$[1]:

$$\tilde{x}(t) = \hat{x}(t) - x_0 \tag{6.6}$$

where x_0 is the true value of x.

Cao and Schwartz show that Kalman filters are exponentially convergent and

[1]For visual clarity, we drop the subscript i in this section, which denotes the application under consideration. However, all values in this section are considered per application values unless otherwise noted.

provide the following bounds on the convergence:

$$\frac{\alpha}{(1+\mu_2)^t}V_0 \leq \tilde{x}_i(t) \leq \frac{\beta}{(1+\mu_1)^t}V_0 \tag{6.7}$$

where $V_0 = \tilde{x}_0^2/p_i(0)$. The remaining values in these inequalities are complex functions of the filter, so we will list each in turn.

First, α and β represent bounds on $p(t)$:

$$\alpha \leq p(t) \leq \beta \quad \forall t \tag{6.8}$$

where

$$\alpha = q_{min} + (\frac{1}{q_{min}} + \frac{s_{max}^2}{o}) \tag{6.9}$$

and

$$\beta \leq q_{max} + \frac{o}{a} \tag{6.10}$$

The values μ_1 and μ_2 are relatively complicated functions of the filter parameters and are defined as:

$$\begin{aligned} \mu_1 &= \min_{t,\, \forall t>0} \frac{\min_{t,\, \forall t>0} M(t)}{\max_{t,\, \forall t>0} p(t-1)} \\ \mu_2 &= \max_{t,\, \forall t>0} \frac{\max_{t,\, \forall t>0} M(t)}{\min_{t,\, \forall t>0} p(t-1)} \end{aligned} \tag{6.11}$$

where $M(t)$ is

$$M(t) = \frac{[p(t-1)]^2\,[s(t)]^2}{o} + q(t)\left(1 + \frac{[s(t)]^2\,p(t-1)}{o}\right)^2 \tag{6.12}$$

As $M(t)$ is a fairly complicated function of the filter parameters, Cao and Schwartz provide looser, but easier to calculate bounds:

$$\frac{\alpha}{(1+\delta/\alpha)^t}V_0 \leq \tilde{x}_i(t) \leq \frac{\beta}{(1+\gamma/\beta)^t}V_0 \tag{6.13}$$

where

$$\gamma \leq M(t) \leq \delta, \quad \forall t \tag{6.14}$$

and

$$\begin{aligned} \gamma &\geq \frac{q_{min}\,\alpha^2}{\beta^2} \\ \delta &\leq \frac{\beta^2\, s_{max}^2}{o} + q_{max}\left(1 + \frac{\beta\, s_{max}^2}{o}\right)^2 \end{aligned} \tag{6.15}$$

These bounds allow us to argue about how the incorporation of adaptive control affect SEEC.

Stability The bounds in Equation 6.7 demonstrate that the Kalman filter converges, therefore the control system that estimates workload will converge; i.e.,adaptive control is stable.

Accuracy Accuracy is guaranteed by combining Assumption 5 with Equation 6.7. If the speedup used by the filter is accurate (per the assumption) and the filter converges, then it will converge to an accurate value.

In fact, it turns out that accuracy can be guaranteed even when we relax Assumption 5 if we allow the interpretation of the filter to change. As described in Section 3.3.2, the filter is designed to estimate the workload of a controlled application. This value is computed using the speedup applied to the application (indeed, the bounds for convergence depend on the square of the magnitude of the speedup). When the speedup is accurate, the value produced by the filter will also be accurate.

As can be seen from Equations 6.7–6.12, even when the speedup is not accurate, the filter will still converge. How, then, do we interpret the meaning of the value produced by the filter? We note that speedup and workload have an inverse relationship in terms of their effect on heartrate (see Equation 3.1). Thus assume, that the true speedup applied at time t is $s_0(t)$ and that $s(t) = k \cdot s_0(t)$. In this case, the filter will converge to provide an estimate of w that is off by another factor of k; i.e.,the filter will converge but to an inaccurate value. While the filter value is inaccurate in this

case, its accuracy is wrong by the same factor that speedup is inaccurate. Therefore, the application heartrate will converge to the correct value even when the speedup is inaccurate. In conclusion, adaptive control maintains the accuracy of the classical control system even when Assumption 5 is violated.

Settling time Under the adaptive control system, we lose guarantees on settling time. In the classical control system the settling time is a simple function of the poles of the control system and easy to calculate. With the adaptive control system, settling time is determined by the bounds in Equation 6.7. While these bounds demonstrate that the system is exponentially convergent under the stated assumptions, the bounds are somewhat difficult to use in practice.

Both bounds are dependent on V_0, which in turn is a function of the initial error in the estimate. This has the benefit that when the workload is known, the initial error is zero and the system converges instantaneously. If, however, it is not possible to quantify the initial error we cannot provide a quantitative bound on convergence. Instead, we can say that in practice settling time is proportional on the error between the application's stated workload and true workload. Furthermore, settling time will depend on the noise in the application (o in Equation 3.6) and the speedup required by the application $s(t)$. Thus, while we lose the guarantee on settling time, Equation 6.7 tells us that the settling time is dependent on things the application can control including: 1) error in workload estimate, 2) application noise, and 3) application resource need. Significantly, settling time is *not* dependent on any parameters of the runtime system that are beyond the application's control or visibility.

Maximum Overshoot With the use of the Kalman filter we lose the guarantee of maximum overshoot. Once the filter converges, the guarantees from the classical control system apply.

Efficiency The use of adaptive control does not change the efficiency of the system in theory. In practice, efficiency improves when workload is not known *a priori* because the system quickly adapts to the right workload, as demonstrated in the

next section.

6.3.3 Results

Figure 6-4 depicts data from the experiment described in Section 5.4.3. Specifically, this figure shows a stacked bar chart representing the additional performance per Watt gains when using SEEC's adaptive control feature to meet a performance goal while minimizing power on machines 1 and 2. Here, the y-axis shows performance per Watt normalized to the static oracle, while the x-axis shows the results for each benchmark.

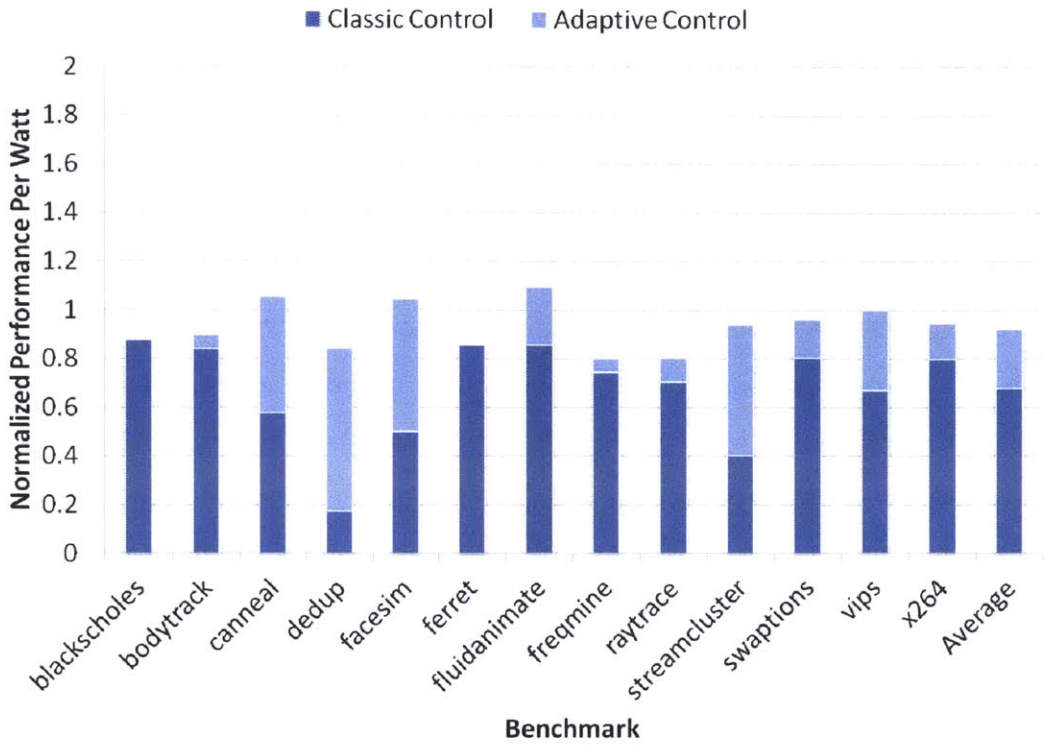
As can be seen from Figure 6-4, incorporating adaptive control into SEEC improves the performance per Watt by almost 50% and makes it competitive with the static oracle. It is somewhat ironic that this improvement is due almost entirely to the fact that the system is much faster to converge to the target heart rate, yet we have lost specific guarantees about the convergence. These results demonstrate that sometimes it is better to relax assumptions and lose guarantees than to have specific guarantees that are based on assumptions that cannot be met in practice.

As described in the previous section, the adaptive controller will converge even when the speedup specified by the systems developer is incorrect. This phenomenon has been demonstrated in the results depicted in Figure 5-16(a). This figure shows an experiment where the speedup and costs associated with actions are incorrect, yet the adaptive control system still quickly converges to the desired performance.

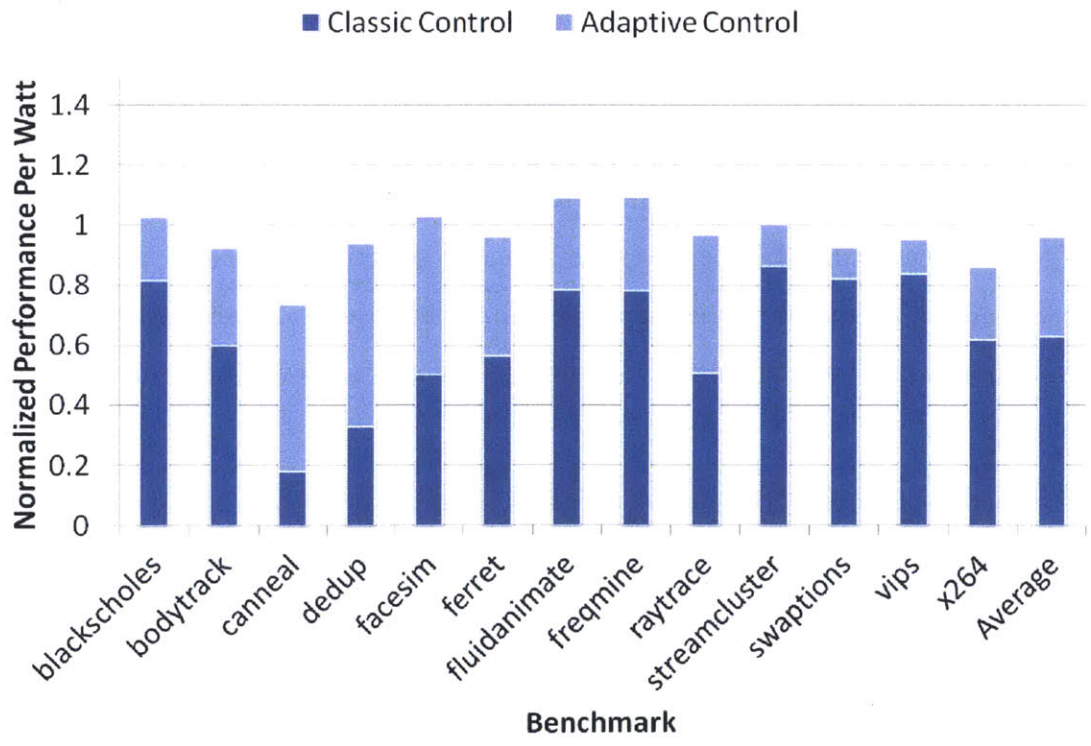
6.4 Properties of Adaptive Actuator Selection

6.4.1 Assumptions

This section relaxes Assumption 4 which states that all combinations of actuators that achieve the same speedup are equivalent. Thus, the adaptive actuator selection process recognizes that there may be multiple methods for achieving a given speedup by that the costs (measured in power or precision) of different combinations



(a) Machine 1



(b) Machine 2

Figure 6-4: Adaptive control provides greater performance per Watt across a range of applications.

of actuators which achieve the same speedup may differ.

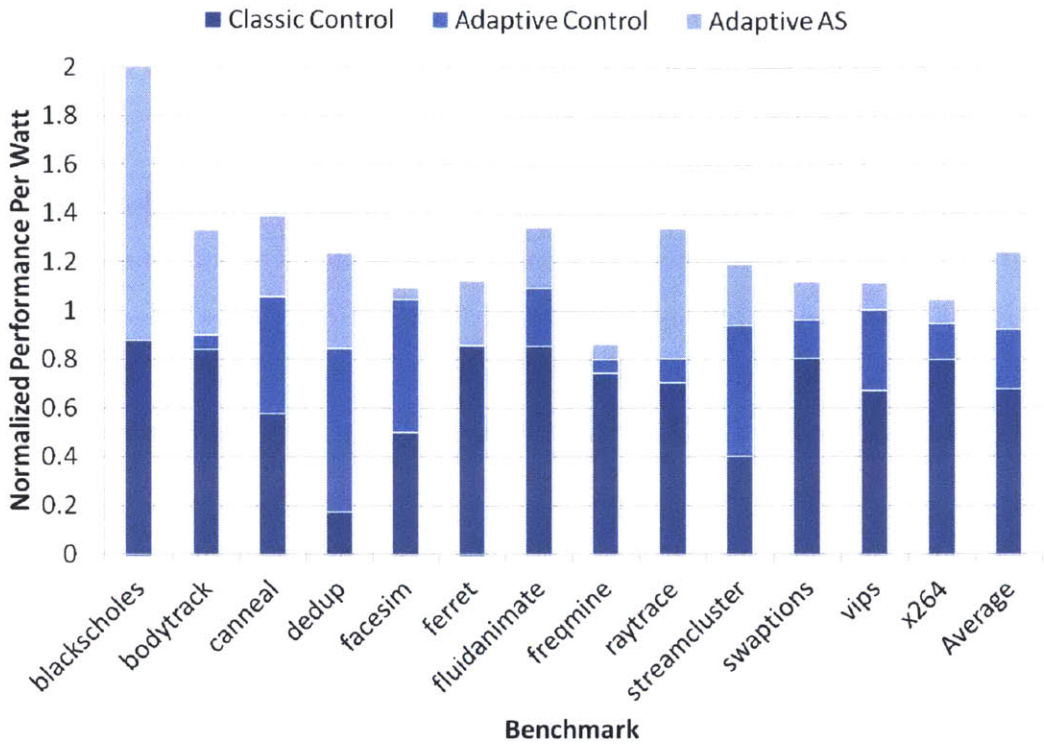
6.4.2 Properties

Adaptive actuator selection uses the speedup signal produced by the adaptive control system and does not affect the behavior of the system in the dimension under control. Therefore, the SASO properties of the adaptive control system carry over to the adaptive actuator selection system. To address efficiency, adaptive actuator selection considers solutions to the linear optimization problem of Equation 3.7. Therefore, any solution to this system is efficient, so the efficiency property is maintained despite the elimination of Assumption 4.

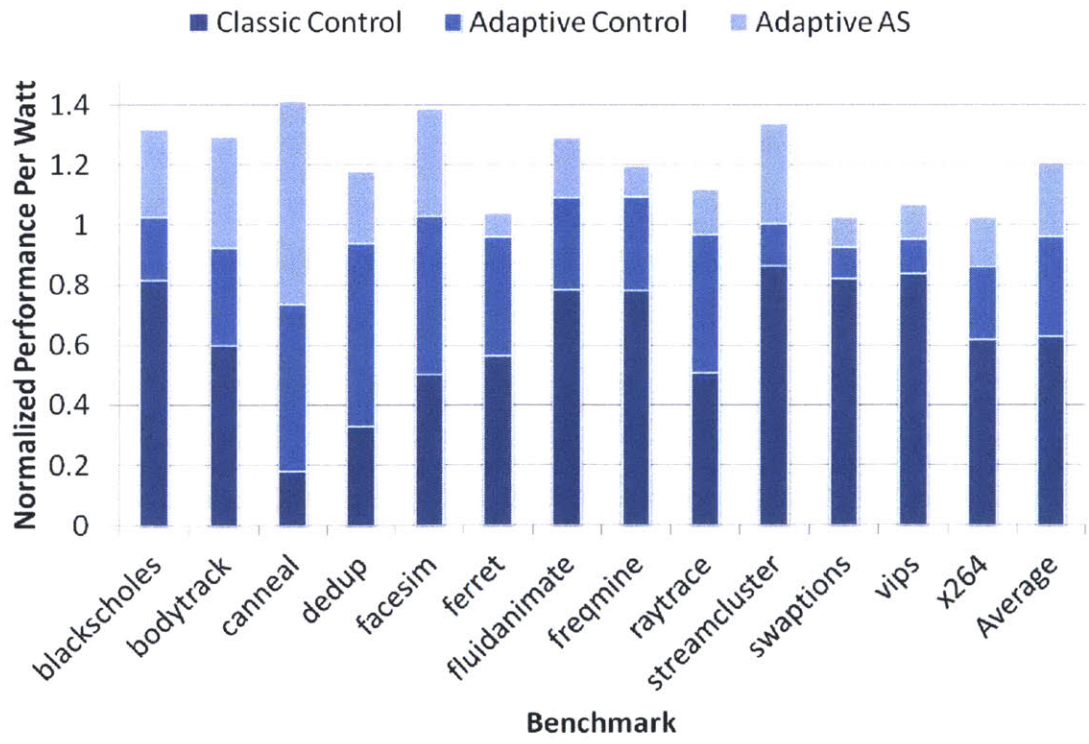
6.4.3 Results

Figure 6-5 depicts data from the experiment described in Section 5.4.3. Specifically, this figure shows a stacked bar chart representing the additional performance per Watt gains when using SEEC's adaptive actuator selection to meet a performance goal while minimizing power on machines 1 and 2. Here, the y-axis shows performance per Watt normalized to the static oracle, while the x-axis shows the results for each benchmark.

As shown in Figure 6-5, adaptive actuator selection provides a significant benefit over adaptive control. This benefit comes from recognizing that there are typically multiple configurations of actuators that meet a goal and that they do not have equivalent costs. In general, on machine 1, adaptive actuator selection attempts to allocate just enough resources to keep the machine busy most of the time. In contrast, on machine 2, adaptive actuator selection uses race to idle to keep the machine busy as much as possible.



(a) Machine 1



(b) Machine 2

Figure 6-5: Adaptive actuator selection provides the best performance per Watt across a range of applications.

6.5 Properties of ML

6.5.1 Assumptions

When invoking SEEC's machine learner, we relax all assumptions except for Assumption 1 and Assumption 6. These assumptions indicate that the application converges to some performance when uncontrolled. Such assumptions are necessary so that the learner can learn the performance of the application. If application performance or noise is not convergent it is clearly impossible to learn what the true application behavior is.

6.5.2 Properties

Stability The machine learner is stable under Assumptions 1 and 6. Given these assumptions, Equation 3.8 will eventually stabilize [90]. Therefore, the predicted Q-value of any state will eventually converge to the true value, and the probability of the machine learner exploring the space will approach zero. Once that probability becomes insignificant, the system behaves as the SEEC control system, but with one caveat: there is a small, but non-zero, probability that the system converges before exploring every state. In this case, Assumption 5 is violated and replaced with an assumption that some subset of the costs and speedups of available actions are known.

Accuracy While the machine learner is stable because it is convergent, it provides no guarantees that it stabilizes to the desired goal because it is not clear that it will explore all states. What is known is that when the system stabilizes, it will stabilize at a point that has been explored (and thus learned; i.e., Equation 3.9 is stable). Furthermore, of all the explored states, the one to which the system converges is the one which has the highest reward (see Equation 3.8).

Settling Time Unfortunately, settling time is not bounded for the machine learner, so this approach provides no guarantees.

Maximum Overshoot Similar to settling time, the machine learner provides no guarantees about maximum overshoot. We note, however, that it is implemented to idle once performance has been exceeded. Therefore, in practice it is not going to overshoot the target. The larger issue is that there is no guarantee on minimum undershoot.

Efficiency Unfortunately, there is only a weak guarantee on efficiency. When the system converges, it will converge to an actuator configuration with the highest reward of all explored states, but there is no guarantee on the extent to which the system will explore.

6.5.3 Results

We have already presented several experiments that show the tradeoffs of SEEC's ML approach. Figures 5-8 and 5-9 show that SEEC ML is better than other approaches but is not competitive with SEEC AAS. In this scenario, the costs and speedups are not known but they represent reasonable guesses for the benchmark applications under study. The overhead of learning the true costs and benefits overwhelms the gain.

In contrast, for the dijkstra and STREAM benchmarks presented in Section 5.6, SEEC ML outperforms SEEC AAS. While both are accurate, SEEC ML is more efficient than SEEC AAS for these applications. The issue here is that the initial models for cost and speedup are so bad (Assumption 5 is violated to such a degree) that the control systems (both SEEC AAS and classical control) converge to an inefficient state. In contrast, when SEEC ML enters such a state, the difference between its predicted utility and measured utility will be large and thus it will move to another state with a very high probability.

Table 6.1: Summary of Assumptions and Properties for SEEC Decision Engines

	Classic	Adaptive	AAS	ML
Assumptions	1—5	1,2, 4—6	1,2,5,6	1,6
Stability	✓	✓	✓	✓
Accuracy	✓	✓	✓	✗
Settling Time	✓	✗	✗	✗
Max Overshoot	✓	✗	✗	✗
Efficiency	✓	✓	✓	✗

6.6 Summary

Table 6.1 summarizes the assumptions and properties covered in this section. The table shows each of the four layers of SEEC, the assumptions made at each level, and the guarantees provided at each level. A ✓ means that the property can be guaranteed. A ✗ indicates that the property cannot be guaranteed.

The data in the table matches the general trend in Figure 6-1. Classical control provides the most guarantees, but also requires the strictest set of assumptions. Machine learning requires the smallest set of assumptions, but in turn provides the fewest guarantees.

Chapter 7

Future Work and Conclusions

7.1 Future Work

7.1.1 Three or More Objectives

The SEEC runtime decision engine as described in Section 3.3 is built with two basic assumptions: 1) the goals of the application and systems developer are the same and 2) there is a single degree of freedom (i.e.,efficiency is only measured in one dimension). These assumptions cover a large number of cases, but clearly not every case. In the future, we plan to extend SEEC to cover scenarios where the application developer and systems developer may be at odds and where there are multiple degrees of freedom. For example, consider a real-time video encoder, which has a clear performance goal; however, the application programmer would like to maximize precision while the systems developer would like to minimize power consumption. In this example, there are two degrees of freedom (precision and power), but the application and systems developers are competitors.

In this scenario, the runtime faces two key challenges: 1) *arbitrating conflicts* and 2) *achieving multi-dimensional efficiency.* Conflicts occur when applications' goals exceed the capabilities of the system or when applications and systems have competing goals. Even without conflicts there may be multiple ways to achieve a set of goals (e.g.,increase processor speed or increase the allocation of processors), and

the runtime needs to ensure that actions are scheduled to meet goals efficiently in both dimensions.

These two challenges represent problems of *policy translation*, which is an open challenge in the creation of self-adaptive systems [77]. One solution is to address both conflict arbitration and action scheduling using economic models, and thus translate the problem to an economic problem. For each goal in the system, developers specify a function that assigns a value, measured in *monetary units* (MU), for progress towards that goal. For example, a video encoder might assign 100 MU for meeting a performance goal of 30 frames per second or higher, 90 MU for 25-29.9 frames per second, and 0 MU otherwise. Similarly, a systems programmer could specify a function that converts energy usage into MU (analogous to what utility companies do). Using this valuation of individual goals, the SEEC runtime can arbitrate conflicts and schedule actions optimally by maximizing the total value in the system.

There are two attractive features to this programming model. First, it provides a scalable method for reasoning about different goals; new goals are translated into MU, so SEEC reasons about them by simply optimizing the total system MU. Second, having a single metric for goodness allows optimization with multiple degrees of freedom. For example, consider a system with a hard performance requirement, but which can meet the performance requirement by either adjusting accuracy or power consumption. This system represents an optimization problem with three dimensions, one of which is a requirement (performance) and two of which are degrees of freedom (accuracy and power). In this system, there is insufficient information to reason about whether to decrease accuracy or increase power consumption in order to meet the performance goal. However, once values are assigned to all three dimensions, the optimization problem simply becomes a problem of maximizing total system value.

Value is assigned to goals by specifying a translation function for each goal that turns measured progress into a value in MU. Separate translation functions must be provided for each separate goal dimension (e.g.,performance, power, etc.).

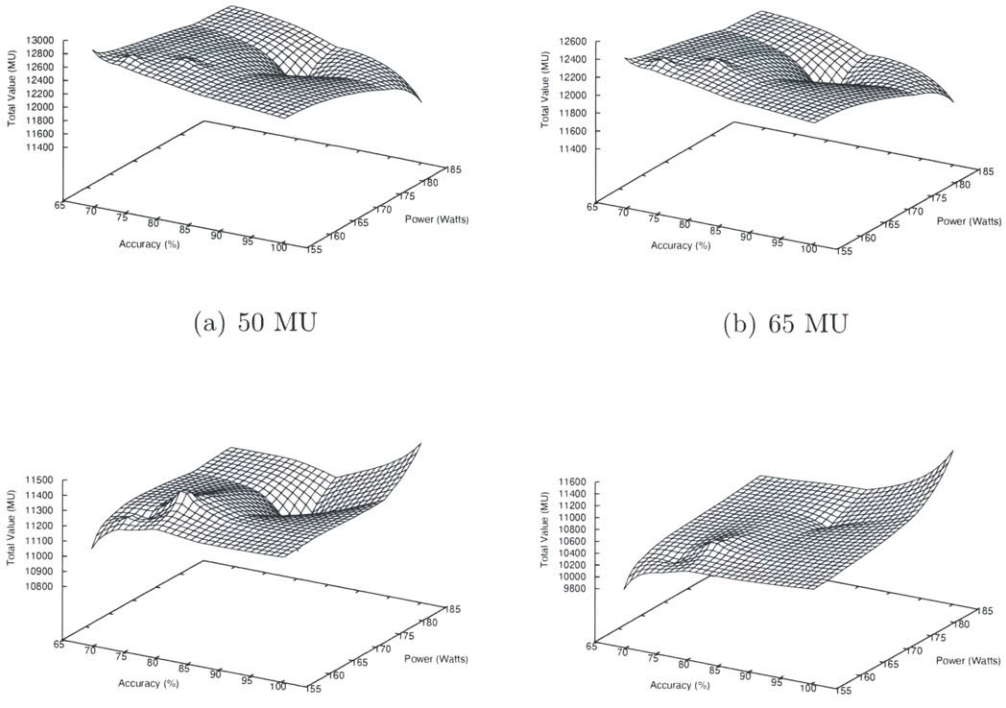
Example

A brief example shows how the economic, self-aware model might be used to implement a search engine. The search engine has a performance goal, measured in requests processed per second, and an accuracy goal measured as the percentage of the maximum number of results returned. This search engine can operate at a range of different performance/accuracy tradeoffs [39]. The system on which the search engine is running has a goal of minimizing power consumption, and there is one actuator which uses DVFS to change the clock speed of the server running the search engine. In this example, the goals of high accuracy, high performance, and low power are in conflict, but these conflicts are easily resolved by optimizing total value in the system.

For this example performance is worth 30000 MU when the goal is met or exceeded and 0 MU otherwise. Power is valued linearly at -100 MU/Watt (representing a cost for power consumption). To illustrate the effects of different value functions we use 4 different valuations of accuracy, and examine how they change the optimal configuration of the system. Specifically, each unit of accuracy is worth either 50, 65, 110, or 150 MU.

The results are shown in Figures 7-1(a) and 7-1(d). Each figure is a three-dimensional plot which shows the total value of the system as a function of the achieved accuracy and power consumption. For each different valuation of accuracy the system achieves a different total value. Furthermore, this optimal value occurs in a different configuration. When accuracy has low value (50 MU), it is better to save power and the system moves to a configuration with the lowest power consumption (155 Watts) and an accuracy of 69.7%. When accuracy has high-value (150 MU), it is better to use the maximum power to achieve maximum accuracy without sacrificing the performance goal (185 Watts, 100%). When accuracy has an intermediate value (65 or 110 MU), the optimal configuration occurs at intermediate values of accuracy and power consumption (158 Watts, 73.8%; 162 Watts, 79.3%).

This example shows how a single runtime system can optimize total system per-



(a) 50 MU

(b) 65 MU

(c) 110 MU

(d) 150 MU

Figure 7-1: Changes in optimal configuration for different valuations of accuracy.

formance using economic models because the system is aware of the value of achieving multiple goals in different dimensions. Through awareness of these values the runtime system understands how to compare accuracy to power and performance, can weigh the relative benefits of increasing accuracy and decreasing power, and can drive system behavior to a global optimum.

7.1.2 Architectural Support for the Model

The SEEC model was designed independently of any particular applications, system software, or hardware. Chapter 5 presents several experiments that demonstrate SEEC working with actions and observations supported by real systems. While these results demonstrate the benefits of the SEEC approach, they also raise questions about how much more could be achieved on a system built to support the SEEC model.

We have proposed a processor architecture, which focuses on features that explicitly support SEEC [38]. Angstrom is a massively manycore processor that supports self-aware computing by exposing a wide array of actions (in the form of different hardware configurations) and observations (including both traditional performance counters [87] and energy counters [75]) to the SEEC runtime system.

Observation

The Angstrom processor design supports SEEC by providing visibility into the hardware in the form of traditional performance counters, event probes, and non-traditional sensors. This information allows SEEC's runtime decision engine to diagnose either why goals are not being met, or whether there might be a lower cost set of actions that would achieve the same goal.

Performance counters provide valuable insight into the behavior of an application on a particular hardware architecture. Unfortunately, many existing systems limit the number of counters that can be read simultaneously by software. This limitation means that application tuning requires multiple profiling runs and prevents dynamic exploitation of performance counter information. The Angstrom design exposes multiple performance counters that are memory-mapped and can be read by any level of the software stack. These count simple events such as: memory operations, cache hits and misses, pipeline stall cycles, network flits sent and received, etc. These are useful for assessing average behavior over a period of time but since they must be polled by software, they cannot be queried too frequently.

Besides observing processor state, Angstrom includes sensors to monitor things like temperature, voltage, battery charge, and energy consumption [75]. This allows the runtime decision engine to react to changing environmental conditions (such as cooling failures or dying batteries) as well as observe how its actions impact these quantities to handle goals like minimizing power consumption or limiting temperature extremes. We expect some of these sensors to be deployed in a fine-grained manner to measure variations between the 1000 cores.

Action

As further support for the SEEC model, the Angstrom processor exposes a number of different actions or different hardware configurations. Angstrom provides these "knobs" but relies on the SEEC runtime system to set them in coordination with other adaptations specified at the software level. This section discusses some of the adaptations exposed by the Angstrom processor at both the intra- and inter-core level.

Intra-core Adaptation In the Angstrom design, each core is capable of running at different voltages and frequencies. Operating the processor designs at lower voltage levels has been shown to increase energy efficiency, as with the voltage-scalable 32-bit microprocessor design demonstrated in [42]. This processor operates at peak performance with nominal voltages while supporting an energy efficient mode at 0.54 V with only 10.2 pJ/cycle energy consumption. Similarly, making each Angstrom core capable of running at different voltage and frequency levels will optimize them for applications with limited energy budgets and time varying processing loads.

Technology scaling is fueling integration of larger on chip caches in processor design (e.g.,up to 50 MB [73]). In order to enable ultra-low power consumption, Angstrom cores need to feature voltage-scalable SRAMs. Conventional SRAMs cannot work at low-voltage levels due to stability problems. Thus, recent work has focused on implementing different bit-cell topologies [19, 13] and peripheral assist circuits [50, 80] to enable operation down to sub-VT levels.

Reconfiguration of the local caches is shown to reduce power consumption for the same performance [7]. Disabling unnecessary parts of the Angstrom caches (sets and ways) will help SEEC to optimize power and performance trade-offs. This adaptation can be beneficial both for applications with small working sets and applications with large working sets that do not achieve much locality on their data.

Inter-core Adaptations Angstrom supports dynamic adaptation of the on-chip network by enabling software and hardware to interact in achieving goal-driven trade-

offs between performance and efficiency. This is accomplished with three architecture features: express virtual channels (EVC) [20], bandwidth-adaptive networks (BAN) [22], and application-aware oblivious routing (AOR) [51].

Angstrom also supports adaptation of the cache-coherence protocol used between cores. For some applications, directory-based cache-coherence provides the best performance and energy consumption [32]. However, for other applications it is more efficient to use a shared-NUCA (non-uniform cache access) protocols because it provides for a large shared cache capacity and reduces the total number of off-chip accesses [49]. The ARCc architecture has shown that combining these protocols and adaptively selecting the best on a per application basis can improve performance and energy efficiency [48]. Angstrom adopts the ARCc approach of providing multiple coherence protocols and exposes these adaptations to SEEC for management.

Decision

Although self-aware optimizations are capable of dramatically improving the behavior of applications, they do not come for free. Some resources must be devoted to making runtime decisions to have a dynamic, adaptive system. To help reduce the costs of runtime decision making the Angstrom processor contains specialized, low-power cores called *partner cores*, which we describe below. More detail is available in [56].

Each main core in the Angstrom design has a partner core associated with it. These two cores are tightly integrated so that the partner core can inspect and manipulate state (including performance counters and configuration registers) within the main core. The partner core also has access to the event queues fed by event probes. The partner core targets a lower performance point than the main core and is designed to take much less area and energy. It has a simplified pipeline, smaller caches and fewer functional units. It is designed to run at lower clock frequencies and makes heavy use of low-power circuit techniques, requiring less energy per operation, and making it more efficient to run dynamic optimization code on the partner core than the main core. We estimate that each partner core will consume about 10% of the area and 10% of the power of a main core.

7.1.3 Distributing SEEC

SEEC's current runtime implementation is designed for existing multicore platforms and will need to be parallelized to work with future large scale multicores. There are two bottlenecks that will need to be addressed as the number of cores and the number of tunable parameters scales. The first bottleneck is the speed at which information can be transmitted from the applications to the SEEC decision engine, and the second is the speed with which SEEC can schedule actions over upcoming time quanta.

We propose to address both of these issues by developing a hierarchical decision engine for SEEC. In this hierarchical approach, regions of cores in the multicore will be broken up into *pods*. The size of the pod will be determined by the latency with which data can be communicated within the pod. Pods will be responsible for making quick decisions at a local level. Each pod will then communicate with a centralized decision engine which will be responsible for making longer term decisions at a slower rate.

There is an additional scaling challenge that has to be addressed to take SEEC to a multi-machine scenario. In such a case, we anticipate extending the hierarchical scheme to allow multiple levels of hierarchy. At least one of these levels will cover goals and constraints for the entire collection of machines under control. Another level of hierarchy will correspond to a single chip level. There may need to be several levels of additional hierarchy as well.

It is also possible that the economic models described in Section 7.1.1 could provide an alternative approach to scaling SEEC up to larger machines. By assigning economic value to actions and achievements, it may be possible to design systems that act independently but in such a way to optimize overall behavior.

7.2 Conclusions

This work describes the SEEC system, a novel runtime and accompanying interfaces, designed to manage power/performance/precision tradeoffs. SEEC drives the system to application-specified goals by tuning actuators provided by individual system com-

ponents. SEEC meets these goals accurately and efficiently using a combination of classical control theory, adaptive feedback control, adaptive actuator selection, and machine learning. A key contribution of SEEC is its use of separate Observation and Actuator interfaces which allow a corresponding separation of concerns. Application programmers use the Heartbeats API to specify application-level goals and progress, while systems developers use the Actuator interface to specify components that affect the performance/power/precision tradeoff space. The SEEC approach has been evaluated in a number of case studies including managing system resources on different machines and managing application adaptation. The SEEC runtime meets goals with low error and close to optimal behavior. SEEC is an example of an emerging class of management systems which will help application and system developers navigate the complicated tradeoff spaces brought in to being by the necessity of managing multiple, competing goals.

Bibliography

[1] Project Gutenberg. http://www.gutenberg.org/.

[2] Wattsup .net meter. http://www.wattsupmeters.com/.

[3] Xiph.org. http://xiph.org.

[4] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36:49–58, December 2003.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

[6] Woongki Baek and Trishul Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, June 2010.

[7] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[8] L.A. Barroso and U. Holzle. The case for energy-proportional computing. *COMPUTER-IEEE COMPUTER SOCIETY-*, 40(12):33, 2007.

[9] Muli Ben-Yehuda, David Breitgand, Michael Factor, Hillel Kolodner, Valentin Kravtsov, and Dan Pelleg. Nap: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, Oct 2008.

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

[12] Aaron Block, Björn Brandenburg, James H. Anderson, and Stephen Quint. An adaptive framework for multiprocessor real-time system. In *ECRTS*, 2008.

[13] B.H. Calhoun and A. Chandrakasan. A 256kb sub-threshold sram in 65nm cmos. In *ISSCC*, feb. 2006.

[14] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. Jagr: An autonomous self-recovering application server. *AMS*, 0, 2003.

[15] Liyu Cao and Howard M. Schwartz. Exponential convergence of the kalman filter based parameter estimation algorithm. *International Journal of Adaptive Control and Signal Processing*, 17(10):763–783, 2003.

[16] Michael Carbin and Martin Rinard. Automatically Identifying Critical Input Regions and Code in Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2010.

[17] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM J. Res. Dev.*, 50(2/3):239–248, 2006.

[18] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC*, 2000.

[19] L. Chang, D.M. Fried, J. Hergenrother, J.W. Sleight, R.H. Dennard, R.K. Montoye, L. Sekaric, S.J. McNab, A.W. Topol, C.D. Adams, K.W. Guarini, and W. Haensch. Stable sram cell design for the 32 nm node and beyond. In *Symposium on VLSI Technology*, june 2005.

[20] Chia-Hsin Owen Chen, Niket Agarwal, Tushar Krishna, Kyung-Hoae Koo, Li-Shiuan Peh, and Krisha C. Saraswat. Physical vs. Virtual Express Topologies with Low-Swing Links for Future Many-core NoCs. In *Fourth ACM/IEEE International Symposium on Networks-on-Chip*, 2010.

[21] Jian Chen and Lizy Kurian John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS*, 2011.

[22] Myong Hyon Cho, Miezko Lis, Keun Sup Shim, Michel Kinsy, Tina Wen, and Srini Devadas. Oblivious Routing in On-Chip Bandwidth-Adaptive Networks. In *PACT*, 2009.

[23] Seungryul Choi and Donald Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA*, 2006.

[24] J. Deutscher and I. Reid. Articulated body motion capture by stochastic search. *International Journal of Computer Vision*, 61(2):185–205, 2005.

[25] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO*, 2010.

[26] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart data structures: an online machine learning approach to multicore data structures. In *ICAC*, 2011.

[27] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *ICAC*, 2010.

[28] A. Gandhi, M. Harchol-Balter, R. Das, C. Lefurgy, and J. Kephart. Power capping via forced idleness. In *Workshop on Energy-Efficient Design*, June 2009.

[29] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[30] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. In *2nd USENIX Windows NT Symposium*, 1998.

[31] N. Goulding, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, J. Babb, M. B. Taylor, and S. Swanson. Greendroid: A mobile application processor for a future of dark silicon. In *Hot Chips*, 2010.

[32] Anoop Gupta, Wolf dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[33] H.264 reference implementation. http://iphome.hhi.de/suehring/tml/download/.

[34] Claude-J. Hamann, Michael Roitzsch, Lars Reuther, Jean Wolter, and Hermann Hartig. Probabilistic admission control to govern real-time systems under overload. In *ECRTS*, 2007.

[35] Joseph L. Hellerstein. Why feedback implementations fail: the importance of systematic testing. In *FeBID*, 2010.

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

[37] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, 2010.

[38] Henry Hoffmann, Jim Holt, George Kurian, Eric Lau, Martina Maggio, Jason E. Miller, Sabrina M. Neuman, Mahmut E. Sinangil, Yildiz Sinangil, Anant Agarwal, Anantha P. Chandrakasan, and Srinivas Devadas. Self-aware computing in the angstrom processor. In *DAC*, 2012.

[39] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[40] J.K. Hollingsworth and P.J. Keleher. Prediction and adaptation in active harmony. In *HPDC*, 1998.

[41] IBM Inc. IBM autonomic computing website. http://www.research.ibm.com/autonomic/, 2009.

[42] N. Ickes, Y. Sinangil, F. Pappalardo, E. Guidetti, and A.P. Chandrakasan. A 10 pj/cycle ultra-low-voltage 32-bit microprocessor system-on-chip. In *ESSCIRC*, sept. 2011.

[43] Intel Inc. Reliability, availability, and serviceability for the always-on enterprise. www.intel.com/assets/pdf/whitepaper/ras.pdf, 2005.

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

[45] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *HotOS*, Berkeley, CA, USA, 2005.

[46] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[47] J.O. Kephart. Research challenges of autonomic computing. In *ICSE*, 2005.

[48] Omer Khan, Henry Hoffmann, Mieszko Lis, Farrukh Hijaz, Anant Agarwal, and Srinivas Devadas. ARCc: A case for an architecturally redundant cache-coherence architecture for large multicores. In *Computer Design, International Conference on*, pages 411–418, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[49] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 211–222, New York, NY, USA, 2002. ACM.

[50] Tae-Hyoung Kim, J. Liu, J. Keane, and C.H. Kim. A high-density subthreshold sram with data-independent bitline leakage and virtual ground replica scheme. In *ISSCC*, feb. 2007.

[51] Michel Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srini Devadas. Application-Aware Deadlock-Free Oblivious Routing. In *ISCA*, 2009.

[52] O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski J. W., Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys*, 2006.

[53] R. Kumar, K. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Processor power reduction via single-isa heterogeneous multi-core architectures. *Computer Architecture Letters*, 2(1):2–2, Jan-Dec 2003.

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

[55] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization* , CGO, Palo Alto, California, March 2004.

[56] Eric Lau, Jason E. Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. Multicore performance optimization using partner cores. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[57] W.S. Levine. *The control handbook*. CRC Press, 2005.

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

[59] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE TPDS*, 17(9):1014–1027, September 2006.

[60] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *RTSS*, 1999.

[61] Martina Maggio, Henry Hoffmann, Anant Agarwal, and Alberto Leva. Control-theoretical cpu allocation: Design and implementation with feedback control. In *FeBID*, 2011.

[62] Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. Controlling software applications via resource allocation within the heartbeats framework. In *CDC*, 2010.

[63] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel. Performance measures for information extraction. In *Broadcast News Workshop'99 Proceedings*, page 249. Morgan Kaufmann Pub, 1999.

[64] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, pages 19–25, December 1995.

[65] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. Composing adaptive software. *Computer*, 37(7), july 2004.

[66] R. Merritt. ARM CTO: power surge could create 'dark silicon'. *EE Times*, 2009.

[67] C. Middleton and R. Baeza-Yates. A comparison of open source search engines. Technical report, Universitat Pompeu Fabra, Department of Technologies, October 2007.

[68] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[69] Simon Oberthür, Carsten Böke, and Björn Griese. Dynamic online reconfiguration for customizable and self-optimizing operating systems. In *EMSOFT*, 2005.

[70] Oracle Corp. Automatic Workload Repository (AWR) in Oracle Database 10g. http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php.

[71] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[72] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: adaptive control of distributed applications. In *HPDC*, 1998.

[73] R.J. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski. A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers. In *ISSCC*, pages 84 –86, feb. 2011.

[74] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM New York, NY, USA, 2006.

[75] Efi Rotem, Alon Naveh, Doron Rajwan amd Avinash Ananthakrishnan, and Eli Weissmann. Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge. In *Hot Chips*, August 2011.

[76] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

[77] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[78] Lui Sha, Xue Liu, Uiuc Ying Lu, and Tarek Abdelzaher. Queueing model based network server performance control. In *RTSS*, 2002.

[79] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *SIGMETRICS*, 2011.

[80] M.E. Sinangil, H. Mair, and A.P. Chandrakasan. A 28nm high-density 6t sram with optimized peripheral-assist circuits for operation down to 0.6v. In *ISSCC*, feb. 2011.

[81] Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: operating system level performance management via self-adaptive computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 856–865, New York, NY, USA, 2012. ACM.

[82] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys*, 2007.

[83] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

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

[85] SWISH++. `http://swishplusplus.sourceforge.net/`.

[86] M. Tanelli, D. Ardagna, and M. Lovera. LPV model identification for power management of web service systems. In *MSC*, 2008.

[87] PAPI Team. Online document, `http://icl.cs.utk.edu/papi/`.

[88] G. Tesauro. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11:22–30, January 2007.

[89] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP*, 2005.

[90] Michel Tokic. Adaptive ϵ-greedy exploration in reinforcement learning based on value differences. In *KI*. 2010.

[91] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. *Mobile Computing*, pages 449–471, 1996.

[92] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical Report TR 95-041, UNC Chapel Hill, Department of Computer Science.

[93] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. In *PACT*, 2010.

[94] x264. Online document, http://www.videolan.org/x264.html.

[95] R. Zhang, C. Lu, T.F. Abdelzaher, and J.A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*, 2002.