Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-046 November 7, 2011

SEEC: A General and Extensible Framework for Self-Aware Computing

Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

SEEC: A General and Extensible Framework for Self-Aware Computing

Henry Hoffmann[1] Martina Maggio[1,2] Marco D. Santambrogio[1,2] Alberto Leva[2]
Anant Agarwal[1]
[1]Computer Science and Artificial Intelligence Laboratory MIT
[2]Dipartimento di Elettronica e Informazione, Politecnico di Milano

Abstract

Modern systems require applications to balance competing goals, e.g. achieving high performance and low power. Achieving this balance places an unrealistic burden on application programmers who must understand the power and performance implications of a variety of application and system *actions* (e.g. changing algorithms or allocating cores). To address this problem, we propose the Self-aware Computing framework, or SEEC. SEEC automatically and dynamically schedules actions to meet application specified goals. 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 observations and actions, according to their expertise. SEEC's runtime decision engine observes the system and schedules actions automatically, reducing programmer burden. This general and extensible decision engine employs both control theory and machine learning to reason about previously unseen applications and actions while automatically adapting to changes in both application and system models. This paper describes the SEEC framework and evaluates it in several case studies. SEEC is used to build an adaptive system that optimizes performance per Watt for the PARSEC benchmarks on multiple machines, achieving results as least $1.65\times$ better than a classical control system. Additional studies show how SEEC can learn optimal resource allocation online and respond to fluctuations in the underlying hardware while managing multiple applications.

1. Introduction

Where once application optimization generally meant performance maximization; now it is increasingly a process of balancing competing goals. For example, as energy and power limits turn into primary concerns, optimization may mean meeting performance targets within a given power-envelope or energy budget. This optimization process can be viewed as a problem of *action scheduling*. Both the application and system (OS, hardware, etc.) support various actions (e.g. changing or assignment of resources) and the application programmer must schedule actions that meet performance goals with minimal cost. Doing so requires both application domain expertise and a deep systems knowledge in order to understand the costs and benefits of a range of possible actions. Additionally, modern systems are dynamic with dramatically changing application workloads and unreliable and failure prone system resources that require online rescheduling of actions in response to environmental fluctuations. It is unrealistic to expect that application programmers acquire the necessary systems knowledge to optimally schedule actions in a fluctuating environment.

Self-adaptive or *autonomic* computing techniques have been proposed to address this problem [23, 26]. Self-adaptive systems are characterized by an *observe-decide-act* (ODA) loop wherein the system monitors itself and its environment making decisions about how to adapt behavior using a set of available actions. These systems have been implemented in both hardware and software, but recent survey papers point out several challenges in their implementation [22, 32, 37]. In this work, we focus on the challenge (described by Salehie & Tahvildari [37]) of creating a general and extensible self-optimizing system and note two common limitations to the generality of existing implementations. First, these implementations tend to focus on a fixed set of actions defined at design time and characterized for a limited domain (e.g. web servers). Second, these approaches focus on a single system layer (e.g. application only) and do not allow coordination across layers.

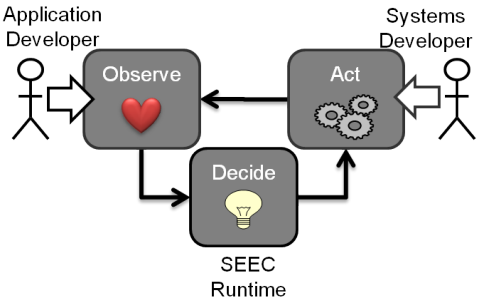


Figure 1. SEEC's decoupled observe-decide-act loop.

This paper proposes the *SElf-awarE Computing* (SEEC) framework to address the problem of automatically and dynamically scheduling actions while balancing competing goals in a fluctuating environment. The SEEC approach overcomes current limitations to generality by flexibly incorporating new actions and coordinating across multiple system layers. SEEC is based on two insights: 1) *decoupling* the ODA loop implementation and 2) supporting a general and extensible decision mechanism – itself an adaptive system. In the decoupled approach, applications specify goals and feedback (and optionally application-level actions), while system hardware and software separately specify system-level actions as shown in Figure 1. The SEEC decision engine automatically schedules actions to meet goals efficiently while responding to environmental fluctuations. SEEC can handle an array of new applications and system actions by automatically building and updating its internal models online. SEEC's decision mechanism is completely general and extensible, and can easily handle new applications and actions without redesign and re-implementation.

SEEC's runtime decision mechanism has four novel features supporting the generalized decoupled approach:

- SEEC directly incorporates application goals and feedback.

- SEEC uses *adaptive* control to respond quickly to new applications or phases within an application.
- SEEC's decision engine implements *adaptive action scheduling* to determine when to *race-to-idle* versus allocating resources *proportional* to need.
- SEEC's runtime combines feedback control with reinforcement learning to adapt its internal system-level models dynamically. It initially functions as a learner and determines the available actions' true costs and benefits, becoming a control system as the learned values converge to the true values.

SEEC is implemented as a set of libraries and runtime system for Linux/x86 which we test in various scenarios. First, we modify the PARSEC benchmark suite [6] to emit goals and progress towards those goals. Separately, we specify several actions on two separate machines that allow SEEC to change the allocation of cores and memory resources to applications, change the clock speed of cores, and change an application itself. We then demonstrate that the SEEC decision engine adapts to meet goals. Specifically, using the available actions on two different machines, SEEC optimizes performance per Watt for the PARSEC benchmarks. When moving from one machine to another the SEEC decision engine is unchanged; however, in both cases SEEC is over $1.65\times$ better than a classical control system. Additional experiments show that, compared to allocating for worst case execution time, SEEC produces $1.44\times$ better performance per Watt for a video encoder across a range of different inputs. Further experiments show that SEEC can learn to manage memory-bound applications without over-provisioning resources and simultaneously manage multiple applications in response to loss of compute power.

This paper makes the following contributions:

- *It proposes and illustrates a decoupled approach to adaptive system implementation.* This approach allows application and systems programmers to specify either observations or actions, according to their expertise. SEEC's runtime decision engine observes the system and schedules actions automatically; ultimately reducing programmer burden. To our knowledge, SEEC provides the first implementation of such a decoupled adaptive system.
- *It addresses the challenge of creating a general and extensible decision mechanism for self-optimizing systems.* Without redesign or re-implementation, SEEC's decision engine supports the decoupled approach by managing a wide range of new applications and system actions.
- *It demonstrates the combination of control and machine learning to manage behavior of a computer system.*
- *It presents a thorough evaluation of the SEEC implementation controlling performance of the PARSEC benchmarks on multiple machines using several different sets of available actions.* This evaluation compares the SEEC decision engine to some existing approaches and demonstrates the benefits of incorporating both control and learning.
- *It presents results and discussions describing some tradeoffs between control theory and machine learning applied to system optimization problems.*

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the SEEC model and runtime system. Section 4 describes the applications and actions used to test the SEEC decision engine. Section 5 presents several case studies evaluating SEEC's decision mechanism. The paper concludes in Section 6.

2. Related Work

A self-aware, adaptive, or autonomic computing system is able to alter its behavior in some beneficial way without human intervention [19, 23, 26, 37]. Self-aware systems have been implemented in both hardware [2, 7, 12] and software [37]. Some example systems include those that that manage resource allocation in multicore chips [7], schedule asymmetric processing resources [36, 40], optimize for power [25], and manage cache allocation online to avoid resource conflicts[45]. In addition, languages and compilers have been developed to support adapting application implementation for performance [3, 42], power [4, 39], or both [17]. Adaptive techniques have been built to provide performance [5, 28, 34, 38, 46] and reliability [9] in web servers. Real-time schedulers have been augmented with adaptive computing [8, 14, 29]. Operating systems are also a natural fit for self-aware computation [10, 21, 24, 33].

One challenge facing researchers is the development of general and extensible self-aware implementations. We distinguish the concept of a general *implementation* from a general *technique* and note that many general techniques have been identified. For example, reinforcement learning [41] and control theory [15] are both general techniques which can be used to create a variety of adaptive systems. However, many techniques lose generality in their implementation; i.e. the implementation addresses a specific problem and new problems require redesign and re-implementation. Other implementations lose generality by fixing a set of actions and failing to extend to new action sets. In this section, we focus on prior work in creating general implementations.

Researchers have developed several frameworks that can be customized for a specific application. These approaches include: ControlWare [46], Agilos [28], SWiFT [13], the *tunability interface* [11], AutoPilot [35], and Active Harmony [18]. 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). Additionally, 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.

Other researchers have developed self-aware approaches to adapt system level actions and handle a variety of previously unseen applications. Such system-level approaches include machine learning hardware for managing a memory controller [20], a neural network approach to managing on-chip resources in multicores [7], a hill-climbing technique for managing resources in a simultaneous multithreaded architecture [12], techniques for adapting the behavior of super-scalar processors [2], and several operating systems with adaptive features [10, 21, 24, 33]. 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 [2]) or high-level metrics

[1] The paper mentions adaptive control can be supported, but the tested implementations use classical control techniques.

Table 1. Comparison of several self-aware approaches.

	ControlWare [46]	Tunability Interface [11]	Agilos [28]	Choi & Yeung [12]	Bitirgen et al. [7]	SEEC
Observation	Application	System	System	System	System	Application & System
Decision	Control[1]	Classifier	Fuzzy 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

Table 2. 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

(e.g. total system throughput [7]), 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 neural network from [7] will have to be redesigned, re-implemented, and retrained. In contrast, SEEC can combine actions specified by different developers and learn models for these new combinations of actions online.

Table 1 highlights the differences between SEEC and some representative prior adaptive implementations. The table includes general approaches for specifying application level adaptation and approaches for specifying system level adaptation for resource management. 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 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 handle previously unseen applications and incorporate new actions without redesign of its decision engine.

3. SEEC Framework

A key novelty of SEEC is its decoupling of ODA loop implementation, which leads to three distinct roles in development: application developer, system developer, and the SEEC runtime decision infrastructure. Table 2 shows the responsibilities of each of these three entities for the three phases of ODA execution: observation, decision, and action. The application developer indicates the application's goals and current progress toward those goals. The systems developer indicates a set of actions and a function which implements these actions. The SEEC runtime system uses a generalized and extensible decision engine to coordinates actions and meet goals. We note that in practice, roles can overlap: application developers can supply actions at the application level and systems developers can provide system level observations.

The SEEC system block diagram is illustrated in Figure 2. One difficulty implementing a decoupled adaptive system is designing a decision engine which can support a wide range of applications and

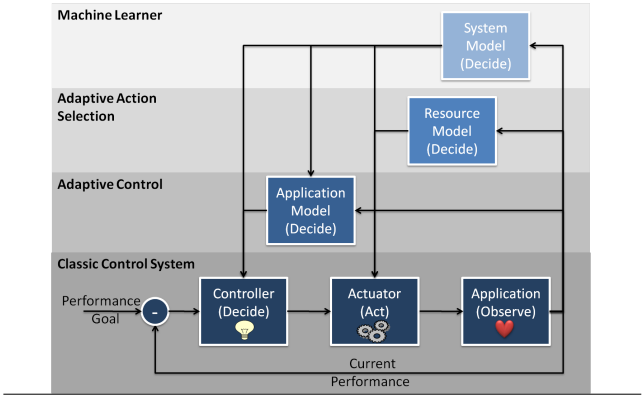


Figure 2. SEEC block diagram.

actions. Given that difficulty, the majority of this section focuses on SEEC's decision engine which is based on 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. SEEC implements an *adaptive action scheduling* scheme to support race-to-idle and proportional resource allocation, as well as a hybrid of the two approaches. Finally, SEEC incorporates a *machine learning engine* used to determine the true costs and benefits of each action online.

This section discusses each of the observation, action, and decision phases in greater detail, with a focus on the decision phase. We note observation and action require developer input, but SEEC's runtime handles decisions without requiring additional programmer involvement.

3.1 Observe

The SEEC model uses the Application Heartbeats API [16] 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 performance goals in terms of a target heart rate or a target latency between specially tagged heartbeats.

The SEEC model assumes that increasing application performance generally comes at some cost (e.g., an increase in power consumption). Therefore, we modify the Application Heartbeats API so application developers can indicate preferred tradeoffs. Currently SEEC supports two tradeoff spaces: *application-level* tradeoffs and *system-level* tradeoffs. Indicating a preference for one over another directs SEEC to exhaust all actions affecting the preferred tradeoff space before attempting any actions in the second. For example, if system-level tradeoffs are preferred, then the SEEC run-

time will only use actions specified at the application level if the target performance cannot be met by any combination of system-level actions. This interface is extensible so more tradeoffs can be specified as more tradeoff spaces are explored.

3.2 Act

The SEEC model provides a separate, system programmers interface for specifying actions that can be taken in the system. A set of actions is defined by the following features: an identifier for each action, a function which implements the corresponding action, and an array of the estimated costs and benefits in the tradeoff space. The performance benefits of an action are listed as speedups, while the costs are listed by type. We currently support two types of costs: POWER and ACCURACY, but the interface can be extended to support additional costs in the future. In addition to specifying the type of the cost, developers can specify a function that SEEC can use to measure cost. This allows the developer flexibility to measure costs where available or provide a model if measurement is not practical on a given system.

By convention, the action with identifier 0 is considered to be the one with a speedup of 1 and a cost of 1; the speedup 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 eight core system.

For example, to specify the allocation of cores to a process in an 8 core system, the developer indicates 8 actions with identifiers $i \in \{0, ..., 7\}$ 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 effect the given application.

SEEC combines n sets of actions $A^0, ..., A^{n-1}$ defined by (possibly) different developers using the following procedure. First, SEEC creates a new set of actions where each action in the set is defined by the n-tuple $< a_i^0, a_j^1, ..., a_k^{n-1} >$, and corresponds to taking the ith action from set A^0, the jth action from set A^1, etc. The speedup 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.

3.3 Decide

SEEC's runtime system automatically and dynamically selects actions to meet the goals with minimal cost. 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.

3.3.1 Classical Control System

The SEEC runtime system augments a classical, model-based feedback control system [15], which complements and generalizes the control system described in [30]. The controller reads the performance goal g_i for application i, collects the heart rate $h_i(t)$ of application i at time t, and computes a speedup $s_i(t)$ to apply to application i at time t.

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

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

Where w is the *workload*, or the expected time between two subsequent heartbeats when the system is in the state that provides the lowest speedup. Using classical techniques, it is assumed that the workload is not time variant and any noise or variation in the system is modeled with the term $\delta h_i(t)$, representing a time varying exogenous disturbance. This assumption of constant workload is a major drawback for a general purpose system and will be addressed by incorporating adaptive control (in the next section)[2].

The control system works to eliminate the *error* $e_i(t)$ between the heart rate goal g_i and the observed heart rate $h_i(t)$ where $e_i(t) = g_i - h_i(t)$. Error is reduced by controlling the speedup $s_i(t)$ applied to application i at time t. Since SEEC employs a discrete time system, we follow standard practice [27, p17] and analyze its transient behavior in the Z-domain:

$$F_i(z) = \frac{1}{z} \tag{2}$$

where $F_i(z)$ is the Z-transform of the closed-loop transfer function for application i (labeled "Classic Control System" in Figure 2). The gain of this function is 1, so $e_i(t)$ is guaranteed to reach 0 for all applications (i.e., the system will converge to the desired heartrate). From Equation 2, the classic controller is synthesized following a standard procedure [27, p281] and $s_i(t)$ is calculated as:

$$s_i(t) \quad = \quad s_i(t-1) + w \cdot e_i(t) \tag{3}$$

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 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 [44].

SEEC's represents the true workload for application i at time t as $w_i(t) \in \mathbb{R}$ and models this workload as:

$$\begin{aligned} w_i(t) &= w_i(t-1) + \delta w_i(t) \\ h_i(t) &= \frac{s_i(t-1)}{w_i(t-1)} + \delta h_i(t) \end{aligned} \tag{4}$$

[2] The assumption of constant workload may not be a drawback for application-specific systems, which model the specific controlled application before deployment.

Table 3. 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 Action Scheduling	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

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:

$$\begin{aligned} \hat{x}_i^-(t) &= \hat{x}_i(t-1) \\ p_i^-(t) &= p_i(t-1) + q_i(t) \\ k_i(t) &= \frac{p_i^-(t)s_i(t-1)}{[s_i(t)]^2 p_i^-(t) + o_i} \\ \hat{x}_i(t) &= \hat{x}_i^-(t) + k_i(t)[h_i(t) - s_i(t-1)\hat{x}_i^-(t)] \\ p_i(t) &= [1 - k_i(t)s_i(t-1)]p_i^-(t) \\ \hat{w}_i(t) &= \frac{1}{\hat{x}_i(t)} \end{aligned} \tag{5}$$

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 [16]. $h_i(t)$ is the measured heart rate for application i at time t and $s_i(t)$ is the applied speedup (according to Equation 3). $\hat{x}_i(t)$ 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 1 and 3 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.

3.3.3 Adaptive Action Scheduling

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:

$$\begin{aligned} \text{minimize}(\tau_{idle}c_{idle} + \tfrac{1}{\tau}\textstyle\sum_{a\in A}(\tau_a c_a)) \quad \text{s.t.} & \\ \tfrac{1}{\tau}\textstyle\sum_{a\in A}\tau_a s_a &= s_i(t) \\ \tau_{idle} + \textstyle\sum_{a\in A}\tau_a &= \tau \\ \tau_a, \tau_{idle} &\geq 0, \ \forall a \end{aligned} \tag{6}$$

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 6 by finding the minimum of c_{race}, c_{prop}, and c_{hybrid} and and using the set of actions corresponding to this minimum cost.

3.3.4 Machine Learning Engine

The use of adaptive control and adaptive action scheduling 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, consider a set of actions which change processor frequency and assume the systems programmer specifies that speedup is linear with a linear increase in frequency. This model works 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 using the workload estimate from Equation 5 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 (using a function supplied by the systems developer, Section 3.2) of taking the action a for τ_a time units and idling for the remaining $\tau_{idle} = \tau - \tau_a$ time units. Given the reward signal, SEEC updates its estimate of the utility function Q_a by calculating:

$$\hat{Q}_a(t) = \hat{Q}_a(t-1) + \alpha(r(t) - \hat{Q}_a(t-1)) \tag{7}$$

[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.

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 $A' = \{b | b \in A,\ \hat{s}_b \geq s\}$
  select $a' \in A'$ s.t. $\hat{Q}_{a'}(t) \geq \hat{Q}_b,\ \forall b \in A'$
end if
return a' and ϵ

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

$$\begin{aligned} \hat{h}_a(t) &= \hat{h}_a(t-1) + \alpha(h(t) - \hat{h}_a(t-1)) \\ \hat{s}_a(t) &= \frac{\hat{h}_a(t)}{\hat{h}_0(t)} \\ \hat{c}_a(t) &= \hat{c}_a(t-1) + \alpha(cost(t) - \hat{c}_a(t-1)) \end{aligned} \quad (8)$$

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 Equations 7 and 8 and then selects an action using Algorithm 1. This algorithm employs Value-Difference Based Exploration (VDBE) [43] to balance exploration and exploitation. As shown in the algorithm listing, SEEC keeps track of a parameter, ϵ ($0 \leq \epsilon \leq 1$) used to balance the tradeoff between exploration and exploitation. When selecting an action to meet the desired speedup, a random number r ($0 \leq r < 1$) is generated. If $r < \epsilon$, the algorithm randomly selects an action. Otherwise, the algorithm selects the lowest cost action that meets the desired speedup. ϵ is initialized to 1 and 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 far from the true value, the algorithm explores available actions. As the estimates converge to the true values, the algorithm exploits the best solution found so far. The value of σ can be used to tune the tradeoff between exploration and exploitation[5].

Having selected and action a', SEEC executes that action and waits until τ heartbeats have been completed (idling itself during this time). If the heartbeats complete sooner than desired for the given value of s, then $\hat{s}_{a'}$ was larger than necessary, 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 7 and a new action is selected using Algorithm 1. 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.

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

[5] For all experiments we set $\sigma = 5$.

3.3.5 Controlling Multiple Applications

When working with multiple applications, the control system may request speedups which create resource conflicts (e.g., in an eight core system, the assignment of 5 cores to one application and 4 to another). SEEC's actuator resolves conflicting actions using a priority scheme. Higher priority applications get first choice amongst any set of actions which govern finite resources. Actions scheduled for high priority applications are removed from consideration for lower priority applications.

For equal priority applications, SEEC resolves conflicts using a *centroid technique*. 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 resources 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 equal priority 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 actions. Similarly, systems developers specify available adaptations without knowing how to monitor an application. Both application and systems developers rely on SEEC's general and extensible decision engine to coordinate the adaptations specified by multiple developers. The decoupled approach makes it easier to develop adaptive systems which can optimize themselves dynamically.

Each of the adaptation levels in SEEC's runtime decision mechanism (Figure 2) builds on adaptations from the previous level and each has its tradeoffs, summarized in Table 3. 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 confident that the systems models (costs and benefits of actions) are accurate, then SEEC will work best using adaptive action selection. These benefits and costs only need to be accurate relative to one another because the adaptive control system can compensate for absolute errors if the relative values are correct. 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 and the response to actions varies, then it is unlikely that the models provided by the developer will be accurate for all applications. In this case, SEEC's reinforcement learning can keep the control system from over-provisioning resources for little added gain. Experiments demonstrating these tradeoffs are described in Section 5.

SEEC can support two types of application goals. If an application requests a performance 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 (near) infinite 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.

Table 4. Variance in heart rate signal for application benchmarks.

Benchmark	Min Resources	Max Resources
blackscholes	3.03E-03	1.90E-01
bodytrack	3.03E-03	2.32E-01
canneal	7.01E+01	2.40E+09
dedup	1.73E+06	1.10E+10
facesim	6.28E-05	3.51E-03
ferret	2.53E+07	2.27E+07
fluidanimate	7.74E-04	1.29E-01
freqmine	3.07E+07	1.17E+09
raytrace	6.46E-04	9.55E-02
streamcluster	8.24E-04	7.41E-03
swaptions	1.35E+02	9.23E+07
vips	2.97E+05	4.93E+09
x264	5.68E+00	4.94E+02
STREAM	4.42E-02	1.93E-01
dijkstra	3.18E-01	2.50E+01

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. To make use of this flexibility, 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.

4. Using the SEEC Model

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

4.1 Applications

We use fifteen different benchmarks to test SEEC's ability to manage a variety of applications. Thirteen of these come from PARSEC [6] version 2.1. The additional benchmarks are STREAM [31] and dijkstra, a benchmark created for this paper.

The PARSEC benchmarks represent a variety of important, emerging multicore workloads [6], and they tend to scale well with increasing processing resources. We modify these benchmarks to emit heartbeats as described in [16]. In general, these benchmarks have some outer loop (in one case, freqmine, control is governed by a recursive function call) and this is where we insert the heartbeats. 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. To manage the goals of these benchmarks, SEEC will have to adapt its internal models to the characteristics of each application including phases and variance within a single application. Each PARSEC benchmark is launched with 8 threads using the "native" input parameters.

Unlike the PARSEC benchmarks, the STREAM benchmark does not scale well with increasing compute resources [31]. 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 paper 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 overprovision resources for a benchmark that cannot make use of them.

4.2 Adaptations

This section describes the adaptations we make available to SEEC's runtime decision engine. To emphasize the generality and extensibility of the SEEC framework, we examine three distinct combinations of adaptations as summarized in Table 5. The first two focus on system level actions, while the third includes both system and application level actions. SEEC's decision engine can manage all three distinct combinations of adaptations without redesign or reimplementation.

All actions are implemented on one of the two different machines described in Table 6. Both machines are hooked up to Watts Up? power meters [1], which measure power consumption over an interval, the smallest supported being 1 second. Acting as systems developers, we provide SEEC with a function to measure cost which accesses the power meter. If SEEC's requests come less than a second apart, the function interpolates power, otherwise it returns the power measurement directly.

4.2.1 Compute Adaptations

These adaptations allow allocation of CPUs and clock speed to running applications. Clock speed is changed by providing a function that manipulates the `cpufrequtils` package. Cores are allocated to an application by changing the affinity mask of processes. For both machines, we tell SEEC the model for speedup is linear for both clock and core changes, i.e., changing either resource by some factor changes speedup by the same factor. The initial power model for each set of actions on both machines is derived by measuring the power of an application that simply does a busy loop of floating point arithmetic. For clock frequency changes, we measure power using a single core. For core changes, we measure power at the highest frequency setting. Even for the compute bound applications in the PARSEC suite this model is overly optimistic as many of the PARSECs do not achieve linear speedup on our test machines. It is up to SEEC's decision engine to overcome this limitation.

4.2.2 Compute and Memory Adaptations

In this case three sets of actions are available to SEEC on Machine 1: the core and clock speed actions from the previous section and a set of two actions which assign memory controllers to the application. These actions change the binding of pages to memory controllers using the `numa` interface. The initial model provided to SEEC assumes that the application's speed increases linearly with the number of memory controllers. We create the model for power by running a separate memory bound application (which repeatedly copies a 1 GB array) and measuring the power consumption using all eight cores at the maximum clock speed while varying the number of memory controllers. We provide SEEC with an initial model which again assumes linear speedup for all resources; i.e. it assumes that doubling the number of memory controllers and the number of cores will quadruple the speed of the application. Such speedup is not close to possible for any application in our test suite. Using this set of adaptations efficiently will test SEEC's ability to learn the true costs and benefits of these actions online and tailor its response to individual applications.

Table 5. Summary of Adaptations

Adaptation	Action Sets	Tradeoffs
Compute Resources	Allocate CPUs & Clock Speed	Performance vs Power
Compute & Memory Resources	Allocate CPUs, Mem. controllers, & Clock Speed	Performance vs Power
Compute Resources & Application Behavior	Allocate CPUs & Clock Speed, Change application control parameters	Performance vs Accuracy

Table 6. Hardware platforms used in evaluation.

Name	Processor	Cores	Memory Controllers	Clock Speed (GHz) (Min/Max)	Available Speeds	Max Power (Watts)	Idle Power (Watts)
Machine 1	Intel Xeon E5520	8	2	1.596/2.394	7	220	90
Machine 2	Intel Xeon X5460	8	1	2.000/3.160	4	329	200

4.2.3 Compute and Application Adaptations

This case again makes three sets of actions available to SEEC on machine 1. The core and clock speed changes used in the previous two sections are reused here. Additionally, we modify x264 to specify 560 possible actions that alter the way it finds temporal redundancy between frames [17]. When x264 is running in the system, SEEC can alter its encoding algorithm using the specified actions to increase speed at a cost of reduced video encoding quality. In this case, the SEEC runtime must create two separate models. The first captures a general, non-adaptive application's response to cores and clock speed, while the second captures x264's response to both compute resources and algorithm changes. Additionally, x264 requests that SEEC favor system-level adaptations rather than application-level ones; i.e. SEEC will only change x264's algorithm if it cannot meet x264's goals with the maximum compute resources. This set of adaptations tests SEEC's ability to manage both system and application level actions and its ability to manage multiple applications at once.

5. Evaluation

This section evaluates the generality, applicability, and effectiveness of the SEEC approach through case studies that make use of the applications and actions described in Section 4. We compare SEEC's decision engine to other, less general, approaches for managing adaptive systems. We first describe these points of comparison and then show results of the case studies.

To evaluate the overhead of SEEC's runtime system, we run all benchmark applications with SEEC disabling its ability to take actions. Thus, SEEC's runtime observes and decides but cannot have any positive effect on the system. We compare execution time in this scenario to the execution time when the application runs without SEEC. For most applications, there was no measurable difference. The largest difference in performance was 2.2%, measured for the fluidanimate benchmark, while the power consumption was within the run to run variation for all benchmarks. These are small overheads and easily compensated for when SEEC is taking action.

5.1 Points of Comparison

We compare SEEC's decision engine to several other approaches: A static oracle, a heuristic system, a classical control system, and a worst case execution time (wcet) allocator.

Static Oracle: This approach schedules actions for an application once, at the beginning of its execution, but knows a priori the best allocation for each benchmark. The oracle is constructed by measuring the performance and power consumption for all benchmarks with all available sets of actions. Obviously, it is impossible to build a static oracle on a real system where the set of applications and inputs is not known ahead of time, but it provides an interesting comparison for active decision making.

Heuristic: This system simply measures the application's heart rate and allocates more resources if the heart rate is too low and fewer resources when the heart rate is too high. The approach is based on one originally used to test the Application Heartbeats framework as described in [16]. This mechanism generalizes to multiple applications, but new actions require redesign and re-implementation.

Classical Control: This is the system is described in Section 3.3.1. One difficulty implementing this approach is the specification of w in Equation 3. Ideally, we would determine w on a per application (or per input) basis, but these studies assume no a priori knowledge. Instead, we use a fixed value of $w = 0.5$ as recommended in [30]. Using classical control as a point of comparison allows us to show the benefits of SEEC's additional adaptive features.

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.

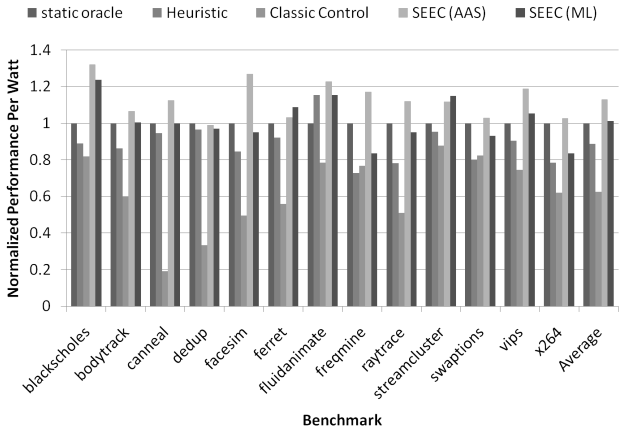
5.2 Optimizing the PARSEC Benchmarks

This study shows how SEEC can meet application goals while minimizing power consumption. SEEC manages the PARSEC benchmarks on both Machines 1 and 2 using the compute adaptations described in Section 4.2.1. Each PARSEC benchmark is launched on a single core set to the minimum clock speed and it requests a performance equal to half the maximum achievable with Machine 1. SEEC's runtime attempts to meet this application specified performance goal while minimizing power consumption. For each application, we measure the average performance and power consumption over the entire execution of the application. We then subtract out the idle power of the processor and compute performance per Watt as the minimum of the achieved and desired performance divided by the power beyond idle. We compute this metric for the static oracle, the heuristic allocator, the classical control system and for SEEC both with and without machine learning enabled.

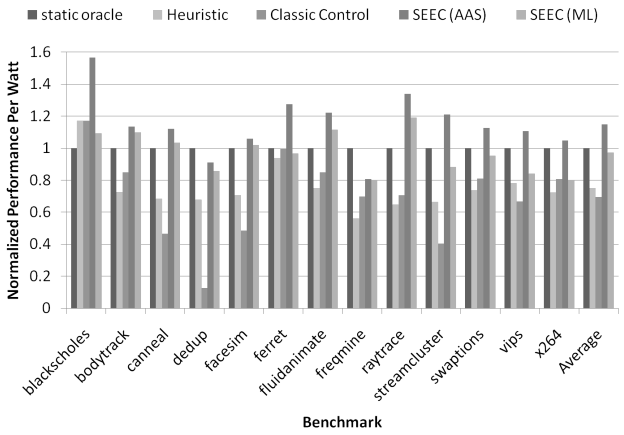
Figures 3(a) and 3(b) show the results of this experiment on Machine 1 and 2, respectively. The x-axis shows the benchmark (and the average for all benchmarks) while the y-axis shows the performance per Watt normalized to that of the static oracle. The bar labeled "SEEC (AAS)" shows the results for the SEEC system with adaptive control and action scheduling but without machine learning. The bar labeled "SEEC (ML)" shows the results for the full SEEC system with the machine learner enabled.

For both machines SEEC (AAS) is over $1.13\times$ better than the static oracle on average. On Machine 1, SEEC (AAS) outperforms the static oracle for all benchmarks but dedup; on Machine 2, it is better than the static oracle for all benchmarks but dedup and freqmine. SEEC (AAS) outperforms the heuristic solution by $1.27\times$ on Machine 1 and by $1.53\times$ on Machine 2. SEEC (AAS) outperforms the classical control system by $1.80\times$ on Machine 1 and by $1.65\times$ on Machine 2. For all benchmarks on both machines, SEEC is able to maintain at least 95% of the performance goal, so the improvement in performance per Watt directly translates into a power savings while meeting application goals.

SEEC (AAS) outperforms other approaches because of its multiple layers of adaptation. SEEC outperforms the classical control



(a) Machine 1



(b) Machine 2

Figure 3. Controlling PARSEC with SEEC.

system as adaptive control tailors response to specific applications online. The heuristic system handles multiple applications but it is slow to react, tends to over-provision resources, and often achieves average performance goals by oscillating around the target. Finally, SEEC outperforms the static oracle by adapting to phases within an application and tailoring resource usage appropriately.

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. The ML engine does outperform all other adaptive systems on Machine 1, achieving $1.01\times$ the performance of the oracle. On Machine 2, SEEC (ML) achieves 97% the performance of the oracle and outperforms the other systems that can be realized in practice. Part of SEEC (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.

SEEC's AAS has a benefit that is not immediately apparent from the figures. On Machine 1, SEEC (AAS) correctly decides to race-to-idle, repeatedly allocating all resources to each benchmark for some small number of heartbeats and then idling the machine for as long as possible (see Section 3.3.3). On Machine 1, adaptively racing-to-idle improves SEEC's performance per Watt by 15% compared to a system that is forced to use proportional

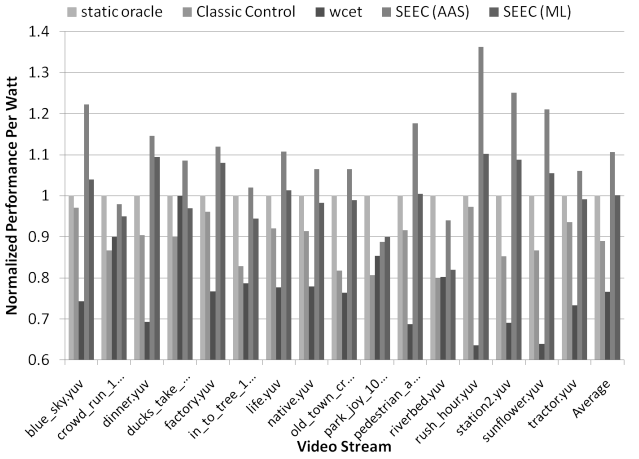


Figure 4. Controlling multiple x264 inputs with SEEC.

allocation. On Machine 2, SEEC determines that it should use the hybrid approach to action selection and improves performance per Watt by 14% compared to a system that is forced to race-to-idle.

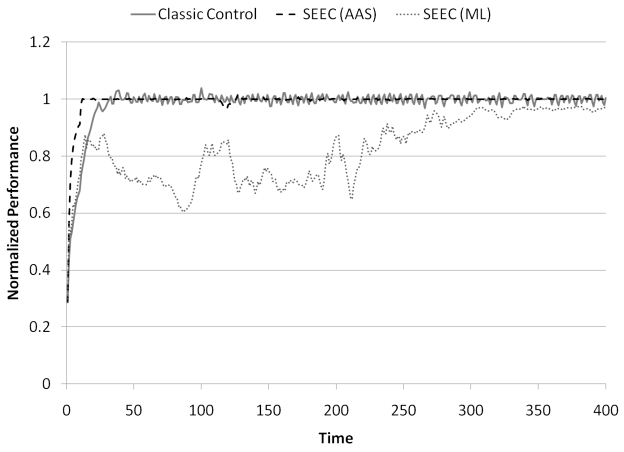
These experiments show that SEEC can handle a range of important multicore applications. As shown in Table 4 the PARSEC benchmarks have a wide range of variance, some have very little variance (e.g. raytrace) and it is not surprising that these can be controlled. Others (e.g. dedup and vips) have tremendous variance in heart rate signal, yet SEEC can still control these applications even without prior knowledge of their behavior. This study also shows how the same decision engine can be used to manage different machines without redesign or re-implementation. The only change going from machine 1 to machine 2 was informing SEEC that a different set of actions is available. SEEC's decision engine takes care of all other required adaptations.

5.3 Optimizing Video Encoding

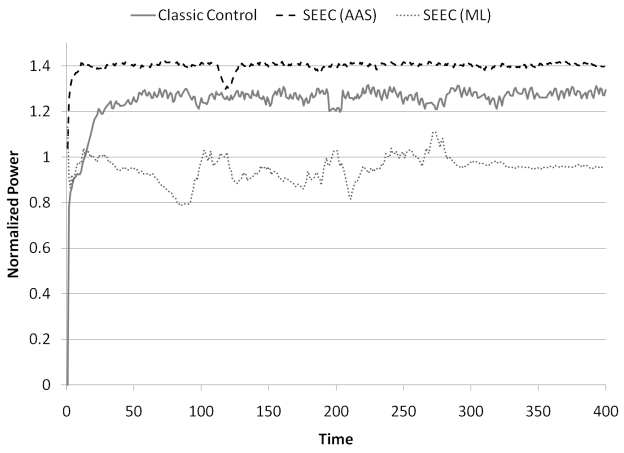
In this experiment, SEEC's decision engine maintains desired performance for the x264 video encoder across a range of different inputs, each with differing compute demands. In addition to the PARSEC native input, we obtain fifteen 1080p videos from xiph.org. We then alter x264's command line parameters 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. 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 4 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.

On average, SEEC (AAS) outperforms the static oracle by $1.1\times$, the classical control system by $1.25\times$, and the wcet allocator by $1.44\times$. SEEC (AAS) bests these alternatives because its adaptive control system allows it to tailor its response to particular videos and even phases within a video. Additionally, SEEC can adaptively race-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 in this case the difference is just 10%. As in the previous study, the system models used here assume linear speedup and that

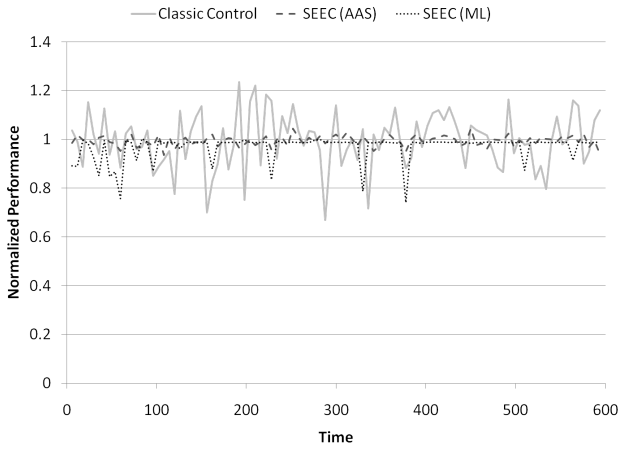


(a) Performance

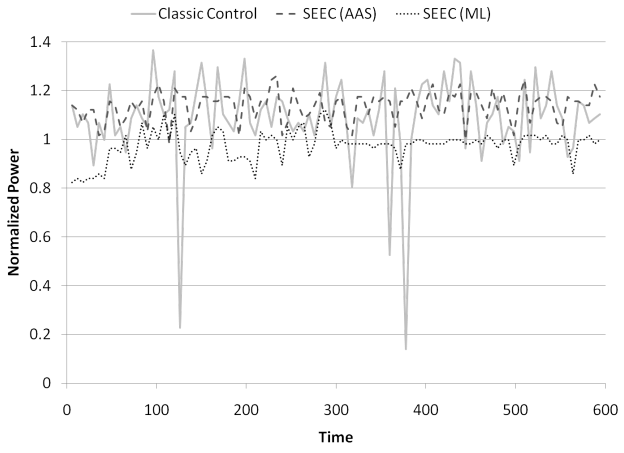


(b) Power

Figure 5. Learning System Models for STREAM.



(a) Performance



(b) Power

Figure 6. Learning System Models for dijkstra.

is good enough for SEEC (AAS) to meet the needs of this application. Using ML, SEEC explores actions to try to learn the true system models on a per input basis, but the exploration causes some 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.

5.4 Learning System 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 set of compute and memory adaptations described in Section 4.2.2. 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).

The results of this case study are shown in Figures 5 and 6. 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.

Even though the two applications have very different characteristics, 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. For dijkstra, SEEC converges to the performance goal while achieving the same power consumption as the static oracle.

These experiments demonstrate the tradeoffs inherent in 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 to the desired performance, 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.

5.5 Managing Multiple Applications

This experiment demonstrates how SEEC can use both system and application level actions to manage multiple applications in response to a fluctuation in system resources. In this scenario, SEEC uses the actions described in Section 4.2.3 to simultaneously control bodytrack and x264. As discussed in Section 4.2.3 this implementation of x264 is, itself, adaptive and its adaptations are managed by SEEC's runtime system. 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 take application level actions if it has exhausted all system level ones.

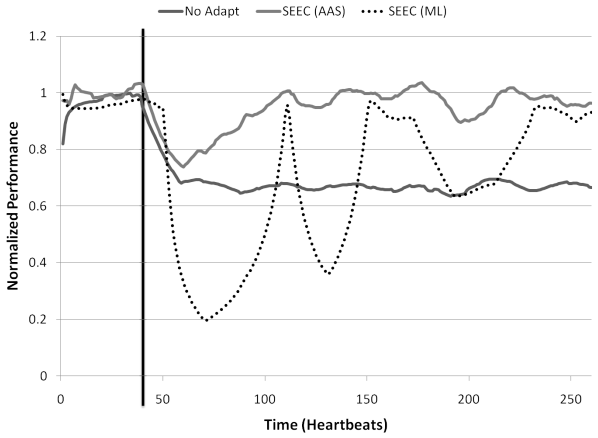
Approximately 10% of the way through execution we simulate a thermal emergency as might occur if chip is in danger of overheating. In response, the hardware lowers the processor frequency to its lowest setting. To simulate this situation, we force Machine 1's clock speed to its minimum and disable the 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.

Figures 7(a) and 7(b) illustrate the behavior of SEEC (AAS) in this scenario, where Figure 7(a) depicts bodytrack's response and Figure 7(b) shows that of x264. Both figures show performance (normalized to the target performance) on the 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.

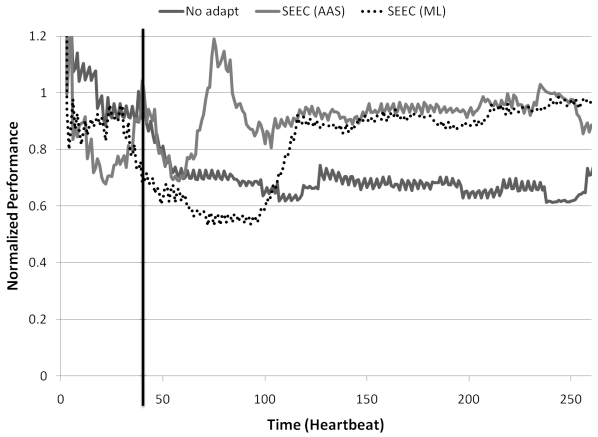
Figure 7(a) shows that SEEC (AAS) maintains bodytrack's performance despite the loss in compute power. SEEC observes the clock speed loss as a reduction in heart rate and deallocates two cores from the lower priority x264, assigning them to bodytrack. Without SEEC bodytrack would only achieve 65% of 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 7(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 that if x264 had been the higher 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 how the SEEC runtime system can control 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, this study shows how SEEC can automatically adapt to fluctuations in the environment by directly observing application performance and goals. SEEC does not detect the clock frequency



(a) bodytrack



(b) x264

Figure 7. SEEC responding to clock speed changes.

change directly, but instead detects a change in the applications' heart rates. Therefore, SEEC can respond to any change that alters the performance of the component applications.

5.6 Discussion

This section describes several studies illustrating the capabilities of the SEEC decision engine and results have demonstrated the tradeoffs between the AAS approach and the ML approach. While both outperform existing mechanisms, neither is ideal for all scenarios. SEEC AAS without machine learning can provide higher performance per Watt when the system models are accurate. When the system models are inaccurate, SEEC AAS quickly brings performance to the desired level (as seen for the STREAM and dijkstra benchmarks), but it does so at a cost of sub-optimal resource allocation. In contrast, SEEC's ML engine learns system models online and adjusts to provide optimal resource allocation even when the models are wrong. This flexibility comes at a cost of reducing SEEC's response time and occasionally exploring actions that do not meet the performance goals.

We note that SEEC's ML engine can be turned on and off as desired and we offer the following general guidelines. If performance is paramount, do not use the ML engine. If SEEC is deployed in a specialized setting where the applications to be run are known beforehand, the ML engine is probably not necessary. If SEEC is deployed in a general purpose setting running a mix of applica-

tions, or a setting where the systems programmers are unsure of the models, then the ML engine should be turned on to save power. In addition, the longer the applications run, the less impact the ML engine will have on average performance because it will eventually converge to the desired value using an optimal resource allocation. For long running jobs ML might provide most of the benefits with negligible drawbacks.

6. Conclusion

This paper proposed the Self-aware computing framework, or SEEC, which automatically schedules actions to meet goals. SEEC helps reduce the burden of programming complex modern systems by adopting a general and extensible approach to adaptive system design. Key to this generality is the decoupling that allows application and systems developers to specify observations and actions independently. This decoupling lets developers focus on their area of expertise while creating a system that can adapt across system layers and coordinate actions specified by multiple developers. To schedule actions, SEEC employs a runtime decision engine that is, itself, an adaptive system. This unique runtime incorporates adaptive control and reinforcement learning to adapt both application and system models automatically and dynamically, ensuring that the action scheduling problem is solved with minimal cost. We have presented a thorough evaluation of the SEEC framework illustrating the benefits of the approach. In addition, this evaluation highlights some of the tradeoffs inherent in the use of control and machine learning techniques.

References

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

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36:49–58, December 2003.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, 2009.

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

[5] M. Ben-Yehuda, D. Breitgand, M. Factor, H. Kolodner, V. Kravtsov, and D. Pelleg. Nap: a building block for remediating performance bottlenecks via black box network analysis. In *ICAC*, 2009.

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

[7] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[8] A. Block, B. Brandenburg, J. H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *ECRTS*, 2008.

[9] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: An autonomous self-recovering application server. *AMS*, 0, 2003.

[10] 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.

[11] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC*, 2000.

[12] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *ISCA*, 2006.

[13] A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. In *2nd USENIX Windows NT Symposium*, 1998.

[14] C.-J. Hamann, M. Roitzsch, L. Reuther, J. Wolter, and H. Hartig. Probabilistic admission control to govern real-time systems under overload. In *ECRTS*, 2007.

[15] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[16] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, 2010.

[17] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

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

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

[20] E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

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

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

[23] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

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

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

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

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

[28] B. 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, Sept. 1999.

[29] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *RTSS*, 1999.

[30] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *CDC*, 2010.

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

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

[33] S. Oberthür, C. Böke, and B. Griese. Dynamic online reconfiguration for customizable and self-optimizing operating systems. In *EMSOFT*, 2005.

[34] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

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

[36] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

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

[38] L. Sha, X. Liu, U. Y. Lu, and T. Abdelzaher. Queueing model based network server performance control. In *RTSS*, 2002.

[39] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys*, 2007.

[40] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

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

[42] 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.

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

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

[45] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. In *PACT*, 2010.

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