MIT-CSAIL-TR-2010-049 October 13, 2010

SEEC: A Framework for Self-aware Computing

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

SEEC: A 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
Massachusetts Institute of Technology, Cambridge, MA 02139
{mmaggio, hank, santambr}@mit.edu {agarwal}@csail.mit.edu

[2]Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
{maggio, santambr, leva}@elet.polimi.it

ABSTRACT

As the complexity of computing systems increases, application programmers must be experts in their application domain and have the systems knowledge required to address the problems that arise from parallelism, power, energy, and reliability concerns. One approach to relieving this burden is to make use of self-aware computing systems, which automatically adjust their behavior to help applications achieve their goals. This paper presents the SEEC framework, a unified computational model designed to enable self-aware computing in both applications and system software. In the SEEC model, applications specify goals, system software specifies possible actions, and the SEEC framework is responsible for deciding how to use the available actions to meet the application-specified goals. The SEEC framework is built around a general and extensible control system which provides predictable behavior and allows SEEC to make decisions that achieve goals while optimizing resource utilization. To demonstrate the applicability of the SEEC framework, this paper presents five different self-aware systems built using SEEC. Case studies demonstrate how these systems can control the performance of the PARSEC benchmarks, optimize performance per Watt for a video encoder, and respond to unexpected changes in the underlying environment. In general these studies demonstrate that systems built using the SEEC framework are goal-oriented, predictable, adaptive, and extensible.

1. INTRODUCTION

The growing complexity of modern computing systems is increasing the burden on application developers. In addition to correctness and performance issues, contemporary application development must handle parallelism, energy efficiency, reliability and predictability issues. Furthermore, these concerns must be addressed even when the execution environment cannot be characterized a priori. Given these issues, it is no longer practical for an average applications programmer to be an expert in their problem domain, have the systems knowledge necessary to manage these additional constraints, and produce an application that performs well on a variety of machines, in a variety of situations. One approach to simplifying the application programmer's task is the use of *self-aware* hardware [20, 9] and software [32, 17]. Researchers have variously called such systems *adaptive*, *autonomic*, *self-**, *goal-oriented*, *adaptive*, etc, and have used these techniques in mobile computing [27], grid and cloud computing [10, 40], networks [3], operating systems [11, 22, 29, 21], the web [34, 31], multicore resources managers [20, 9, 35, 23], and adaptive and dynamic compilation environments [39, 4, 6, 33].

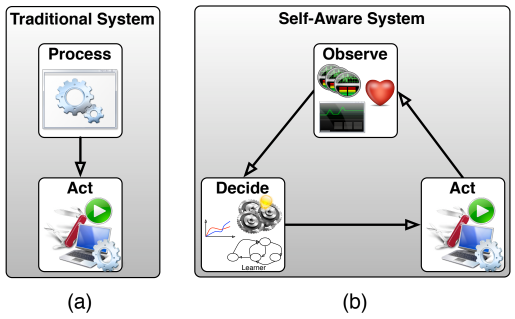


Figure 1: Comparison of traditional and self-aware systems.

Figure 1 illustrates the difference between traditional systems and self-aware systems. Traditional, non-adaptive systems, run in an open loop; application programmers design software given the characteristics of the target platform and assumptions about future inputs, power budgets, system load, etc. Then the application is deployed without the flexibility to change its behavior. This design process has two drawbacks: 1) it forces the designer to be an expert in the application domain and the target platform and 2) it lacks the flexibility to dynamically modify decisions if the original assumptions change. In contrast, self-aware systems run in a closed loop. In such a system all layers – including hardware, compilers, operating systems, and applications themselves – may be capable of observing their environment, altering their decisions, and changing their policies at run time.

This paper presents the SElf-awarE Computing (SEEC) framework, a new computational model supporting the implementation of observe-decide-act (ODA) loops involving both applications and system software. In the SEEC model applications specify their goals, system software specifies a set of possible actions, and the SEEC framework is responsible for deciding how best to use the available actions to meet application goals. For example, a video application might specify a performance target of 30 frames per second, while the system scheduler might specify a set of actions that allocate cores to applications. Given these specifications, the SEEC framework then implements an ODA loop, wherein it observes application performance and uses these observations as input to a control-theoretic decision engine. SEEC's control system decides which of the specified actions should be taken given its observations. Continuing the example, the SEEC framework would observe the video application's performance and change its allocation of cores in order to meet its goal of 30 frames per second using the minimum number of cores.

The SEEC framework provides several benefits for developing self-aware systems. First, it is goal-oriented as it directly incorporates application goals and measures progress towards those goals. Second, SEEC systems have predictable behavior because the decision making process is grounded in control theory allowing known, mathematical characterization of the system's response to stimuli. Third, SEEC is adaptive in that it continually monitors progress and alters decisions allowing it to respond to unforeseen or changing circumstances. Finally, SEEC is extensible in that it is designed to work with a wide variety of applications and system software. SEEC uses the Application Heartbeats API, which provides a general method for applications to indicate performance goals and progress [16]. SEEC's control framework is easily customized by specifying a set of actions, and SEEC is general enough to work with any set of actions which affect application performance.

To demonstrate the benefits of the SEEC framework, this paper presents several case studies. In these studies, the PARSEC benchmarks [8] are modified to use the Heartbeats API and register goals and performance. Five separate systems are then developed to control the behavior of these applications. Results show that these SEEC systems can predictably achieve their goals. Additionally, a SEEC-based system is shown to optimize performance per Watt for a video encoder across a range of input videos, each with differing compute demands. For many videos, this dynamic system is able to adapt its behavior and exceed the performance per Watt of the best static allocation of resources. Finally, several experiments demonstrate how SEEC systems can adapt their behavior to maintain performance in the face of environmental changes like clock frequency changes or core failures.

Self-aware computing has been used to meet many of the challenges in modern computing systems. In addition several systems have been built using self-aware techniques to allocate multicore resources [20, 9], dynamically manage application power [33, 6] and build adaptive operating systems [22, 29]. The SEEC framework presented in this paper has several distinguishing characteristics compared to previous work in self-aware computing. First, the framework is designed to be general and to make it easy to apply self-aware computation to a variety of applications and system software. Second, SEEC incorporates the needs of both applications and system software in self-aware systems. Using a general method for specifying application goals allows SEEC to use a general and widely applicable method for decision making and, thus, SEEC is easily extended to incorporate new actions. Third, SEEC incorporates application goals and feedback to directly measure its effects and avoid having to infer application progress from low-level metrics. Finally, SEEC is grounded in control theory which provides guarantees of the system's response to stimuli. While control theory has been used in computing systems [15, 21], the SEEC approach is unique in that its controller is not built for a specific purpose, but rather provides a general and broadly applicable framework which is easily customized for specific systems. This approach is easily extensible and allows systems developers to benefit from control-theoretic techniques without understanding the details of designing control systems.

This paper makes the following contributions:

- It presents the SEEC framework, a general computational model for enabling applications and system software to cooperate in a self-aware manner. In the SEEC model application specify goals, systems specify possible actions, and the SEEC framework decides what actions to take to meet goals.
- It describes the SEEC control system, which grounds the decision making processes of SEEC-based systems in control theory. This control system is distinguished by its generality as it provides a framework that is easily customized and extended to develop new systems. These properties are demonstrated by developing five separate self-aware system services.
- It presents several case studies, illustrating how systems implemented with SEEC can predictably control application performance, minimize power-consumption and resource utilization, self-optimize, and respond to fluctuations in the underlying hardware system.
- It demonstrates the applicability of the SEEC framework by modifying the PARSEC benchmarks to take advantage of self-aware computation and showing how the SEEC control framework can regulate these benchmarks.

The rest of the paper is organized as follows. Section 2 presents an example illustrating how the SEEC framework can minimize the power consumption of a video encoder while maintaining target performance. Section 3 describes the SEEC framework including the feedback mechanism applications use to specify goals and progress as well as the decision making support and the coordination mechanisms used to provide self-aware system services. Section 4 describes the development of five separate self-aware systems in the SEEC framework. Section 5 presents an experimental evaluation of the SEEC framework and the systems developed in the previous section. Section 6 discusses related work and the paper concludes with Section 7.

2. MOTIVATING EXAMPLE

To illustrate self-aware computing with the SEEC framework, we present an example of both an application and separate system software running on a multicore system. The application is x264, an open-source implementation of the H.264 video codec [41] (which is also part of the PARSEC benchmarks). The system software is a resource allocator that manages both processor speed and available cores to meet application performance goals while minimizing power consumption. Maintaining performance in the video encoder is complicated by the fact that videos can vary in their complexity, meaning some inputs require more compute resources to meet the same performance. For example, videos with few differences from frame to frame (e.g., typical video conferencing) require fewer operations to encode, while more complicated videos (e.g., sports) require more operations. We would like the simple videos to consume fewer resources and less power than the more complicated videos which we expect to require more resources and thus more power. Furthermore, we would like to separate the concern of managing resources from that of writing the video encoder.

Using traditional methods, an application developer produces a power-optimal video encoder by understanding both the video domain and how resource utilization within the target platform affects power. While possible, making the video programmer responsible for power management places greater burden on the programmer and may result in an ad hoc solution. Such a solution can produce optimal resource utilization for the video encoder on a specific hardware platform, but is likely unsuitable for other applications or even the same application running on different hardware.

Alternatively, using the SEEC framework the encoder developer specifies goals and current performance allowing the resource allocator to allot the minimal amount of resources required to meet those goals. This solution does not require the application program-

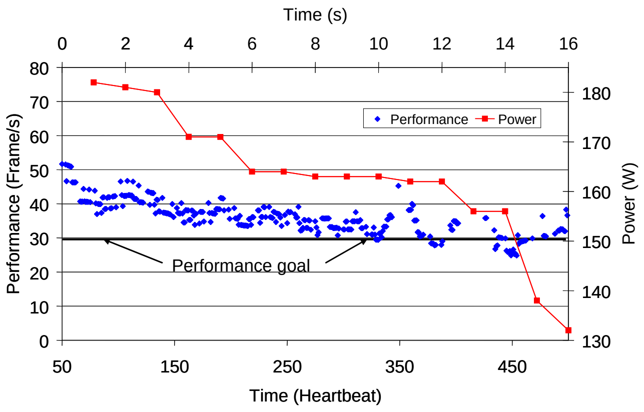


Figure 2: x264 power and performance with SEEC.

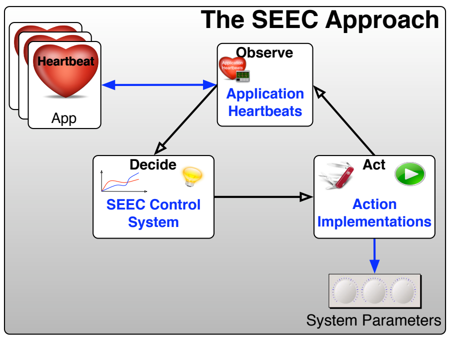


Figure 3: The SElf-Aware Computing (SEEC) Framework.

mer to understand the power/performance tradeoffs in the underlying hardware and operating system. The SEEC solution also clearly separates the application development from system optimization, which results in a modular solution and increases portability. Additionally, by using the SEEC framework the encoder and the system both have the flexibility to adapt to unanticipated events, such as videos that are easier or harder to encode, changes in the underlying hardware, or even changes in application goals.

To make use of the SEEC system the application developer simply instruments the code using the Application Heartbeats API [16]. For x264, this instrumentation requires adding five lines of code. These additional lines include adding a header file, declaring a heartbeat data structure, initializing and finalizing that data structure, and a call to signify the heartbeat. The initialization function is used to specify the desired heart rate. Since we want our encoder to maintain thirty frames per second, we specify a desired heart rate of thirty beats per second and add the call to signify the heartbeat each time a frame is encoded. Having made this small set of changes, the application is prepared to benefit from system services written in the SEEC framework.

To meet performance goals, the encoder relies on a separately developed resource allocator built by customizing SEEC's control system. This control system uses the application's heartbeat information as a feedback mechanism and adjusts its actions to achieve the desired heart rate. SEEC provides a skeleton of a generic controller that system software developers can specialize into different classes of control systems to suit their particular needs. To customize the controller, a system software developer specifies a set of actions, the speedup associated with a given action, and a function that executes an action.

For this example, the systems developer must specify a set of actions which affect power. We note that the power in a multicore chip is related to the number of cores in use and the clock speed of those cores. Using dual quad-core Intel Xeon E5530 processors running Linux and cpufrequtils, the system has 56 actions (allocating up to seven speeds and one to eight cores) available to the resource allocator. Thus the developer defines a function that executes an action by 1) changing clock speed using cpufrequtils and 2) changing the number of available cores by manipulating affinity masks. Given this information, the generic SEEC control system will find the state with the lowest speedup (and thus power) required to meet the application's performance goals.

At this point we have an example self-aware system created using SEEC. We have an application (x264) which emits heartbeats as an indicator of performance and system software that reads those heartbeats and adjusts its policies (available cores and clock speed) to meet application goals (thirty frames per second). The behavior of this system is illustrated in Figure 2. The figure shows the performance and power consumption of x264 over time. Time is shown on the x-axis while performance (heartbeats, or frames, per second) is shown on the left y-axis and total system power (measured with a Wattsup power meter [1]) is shown on the right y-axis.

As shown in Figure 2, the resource allocator initially assigns all available resources to x264. This over-provisioning results in a performance of over 50 frames per second and a power consumption of over 180 W. The resource allocator detects that the encoder exceeds its target performance and reduces the amount of available cores and the clock-speed of those cores until performance is at the desired level of thirty frames per second. This reduction in resources produces a corresponding reduction in power; as the encoder completes it is consuming just over 130 W.

This example demonstrates several of the benefits of self-aware computing with the SEEC framework. First, it shows how self-aware computation can help relieve application programmer burden; in this case the video developer could ignore power optimization knowing that the system would adapt to meet application goals. Second, it illustrates the kind of benefits that can be gained by building self-aware systems; in this case a system power savings of 50 W without missing application deadlines.

3. SEEC FRAMEWORK

The SEEC framework uses input from applications and systems developers to implement a closed-loop system with three distinct phases: *Observation*, *Decision*, and *Action*. This ODA loop is characteristic of control systems; during the observation phase the system collects information, which is fed to the decision phase. During the decision phase the system determines whether recent observations warrant a change in behavior and, if needed, what form this change should take. If adaptation is desired, the action phase implements the adaptation dictated by the decision process. The SEEC framework, as shown in Figure 3, supports this form of closed-loop execution by generalizing the observation and decision phases, providing standard techniques that work with a broad range of actions.

In the SEEC model there are three distinct participants, or roles: application developer, system software developer, and the SEEC framework itself. Table 1 shows the responsibilities of each of these three entities for each of the three phases of observation, decision, and action. The application developer's only responsibility is to indicate the application's goals and current progress toward those goals (see Section 3.1 for more details). The systems developer needs to indicate a set of actions and a function which implements these actions (see Section 3.3 for more details). All responsibilities

Table 1: Roles and Responsibilities in SEEC Development

Phase	Applications Developer	Systems Developer	SEEC Framework
Observation	Specify application goals and performance	-	Read goals and performance
Decision	-	-	Determine how much to speed up the application
Action	-	Specify actions and a function that performs actions	Initiate actions based on result of decision phase

listed under the SEEC framework are handled automatically by the system as described later in this section. In brief, the SEEC framework reads the application goals and performance, uses this data as input to a control system, and the control system decides what actions to take (see Section 3.2 for details on the control system).

3.1 Observing Application in SEEC

A key feature of the SEEC framework is its direct incorporation of application-specified goals and performance feedback into its decision making progress. Toward this end, SEEC uses the Application Heartbeats interface to make an application's goals and progress known to the rest of the system [16]. Using this interface applications emit *heartbeats* at some important place in the code by making calls to the Heartbeat API. For example, in a video encoder a heartbeat might be registered for every encoded frame. Additional functions in the interface allow applications to specify their goals in terms of a desired heart rate or a desired time between specially tagged heartbeats. The Application Heartbeats interface contains additional API functions that allow the SEEC framework to query an application's heartbeat and goals. In comparison with a standard control system, the heartbeat serves as a sensor that provides information about an application's execution.

The ability to control an application's heartbeat will be influenced by the *soundness* of a particular heartbeat signal. The SEEC framework uses past application measures to obtain a model of future behavior (discussed further in the next section). The higher the correlation between the recent past and the near future, the more accurate the control. For example, if a video encoder's frame rate over the last second is a good indicator of its frame rate in the next second, the system will allocate resources more efficiently. The SEEC framework does not require, or expect, that this soundness condition always hold (and in fact, a change in heartbeat data often drives a change in the decision making process). The control system discussed in the next section uses a simple adaptation mechanism to estimate the workload in different phases of the application and may decide to adjust its actions based on these estimates. This adaptation allows the control system to adapt to underlying changes in the environment.

We note that for application developers, adding Heartbeats is the only requirement for taking advantage of SEEC (as shown in Table 1). Typically, this addition requires adding less than six lines of code to the application and linking against the publicly available implementation of the Application Heartbeats API. Application developers who want to create adaptive applications can, of course, use the SEEC control system within an application to drive adaptation.

3.2 Making Decisions in SEEC

The use of the Heartbeats API means that all applications in the system have a standard method for indicating their current performance and there is a standard method for observing this behavior. This standardization of observation makes SEEC's control system possible. SEEC provides a general decision making framework for system services designed to control the performance of Heartbeat-enabled applications. The SEEC control system takes a series of heartbeat observations as input and produces a series of desired speedups which are then used to determine what actions the system should take.

As the Heartbeats framework abstracts applications into a heartbeat, the SEEC control system abstracts decisions into reasoning about speedup. The controller determines when, and by how much, to speedup an application in order to achieve a desired heart rate (or latency between heartbeats). By reasoning about speedup, rather than specific decisions, the SEEC controller can be used as a decision making engine for a wide range of potential actions including resource allocation, algorithmic modifications, policy modifications, etc. The SEEC control system can serve as a decision engine for any system that affects application performance.

We note that the SEEC control system automatically provides decision making capabilities as part of the framework; neither the application nor systems developer is responsible for implementing the decision making process or understanding control theory. Thus, the SEEC framework allows self-aware systems to benefit from a well-founded decision making process without understanding the details. The controller is general, however, so developers can customize its behavior if desired.

3.2.1 The SEEC Control System

This section provides an overview of the SEEC control system, which complements and generalizes previous work on controlling heartbeat-enabled applications [26].

The SEEC system block diagram is illustrated in Figure 4. The controller observes the heartbeat data of the application and models application performance at the k-th heartbeat as

$$r(k) = \frac{s(k)}{w(k)} + \delta r(k) \tag{1}$$

Where $r(k)$ is the heart *rate* of the application at time k, $s(k)$ is the relative speedup applied to the application between time $k-1$ and time k, and $w(k)$ is the *workload* of the application. Workload is defined as the expected time between two subsequent heartbeats when the system is in the state that provides the lowest speedup. For example, in an adaptive resource allocator the workload corresponds to the application performance with the minimal amount of the resource under control. For control purposes we assume that the workload is not time variant and its variations, which in principle cannot be predicted, are modeled with the term $\delta r(k)$, representing an exogenous disturbance.

Using the Heartbeats API an application specifies its desired heart rate $\bar{r}$. The SEEC controller uses that value as a target, or set point and the controller is designed to be as general as possible while converging to the set point without committing errors. Additionally, the controller is parameterized, allowing system developers to customize transient behavior to suit the needs of a specific implementation. Examples of these customizable options include the time for the controller to converge to the set point, the existence

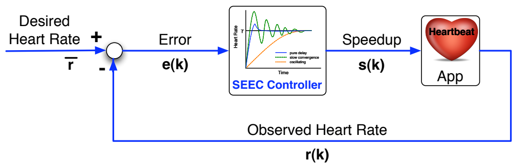


Figure 4: Block diagram of the SEEC control system.

of oscillations around the set point, and whether over- or under-shooting the set point is allowed.

Specifically, SEEC supports three classes of control systems and three possible trajectories from an observed heart rate to a desired heart rate as illustrated in Figure 5. The first, a *pure delay* controller, has the shortest time for the observed heart rate to converge to the desired heart rate, but it is also sensitive to noisy or unsound feedback from the application. The second, a *slow convergence* controller, takes longer to converge but is the least sensitive to noise or disturbance in the heart rate observations. The third, an *oscillating* controller, quickly achieves an average heart rate that is equal to the desired value, but does so by alternately over- and undershooting the the target heart rate before finally converging.

SEEC's controller is based on a generic closed-loop transfer function, i.e., the function that produces the heartrate r given the target $\bar{r}$. In general, a transfer function is a mathematical representation of the relation between the input and the output of a linear time invariant system. In Figure 4, the input is the desired heart rate and the output is the observed heart rate. SEEC's controller provides a guarantee that, the observed heart rate converges to the desired heart rate provided a set of actions that produce sufficient speedup. In addition, the controller is generic in that system developers are free to customize the parameters of the generic transfer function, and thus the *trajectory* of the observed heart rate as it converges to the desired value.

Selecting the behavior of the control system as it converges to the set point is equivalent to selecting the shape of the system in the frequency domain. As the SEEC framework uses a discrete time model, we use the Z-transform to obtain its frequency domain representation [24, p17]. Defining $\bar{R}(z)$ as the transfer function of the desired heart rate and $R(z)$ as the transfer function of the observed heart rate, we set the relation between the output and the input

$$\frac{R(z)}{\bar{R}(z)} = \frac{(1-p_1)(1-p_2)}{1-z_1}\,\frac{z-z_1}{(z-p_1)(z-p_2)} \tag{2}$$

where z^{-1} is the delay operator and $\{z_1, p_1, p_2\}$ are a set of customizable parameters which alter the transient behavior (or trajectory as described in Section 3.2). The gain of this function is 1, so the system will reach the set point or the desired heart rate assuming sound feedback. From Equation 2, the generic SEEC controller is synthesized following a classical control procedure [24, p281]. As shown in Equation 1, SEEC must determine $s(k)$, the speedup to apply at time k. This value is calculated thusly:

$$\begin{aligned} e(k) &= \bar{r} - r(k) \\ s(k) &= F \cdot [\, A\, s(k-1) + B\, s(k-2) + \\ &\quad\; C\, e(k)\, w(k) + D\, e(k-1)\, w(k-1)\,] \end{aligned} \tag{3}$$

where $e(k)$ is the *error* between the current heart rate and the desired heart rate at time k as shown in Figure 4. And the values of $\{A, B, C, D, F\}$ come from the controller synthesis:

$$\begin{aligned} A &= - && [-p_1 z_1 - p_2 z_1 + p_1 p_2] \\ B &= - && [p_2 z_1 + p_1 z_1 - z_1 - p_1 p_2] \\ C &= + && [p_2 - p_1 p_2 + p_1 - 1] \\ D &= + && [p_1 p_2 - p_2 - p_1 + 1]\, z_1 \\ F &= + && [z_1 - 1]^{-1} \end{aligned} \tag{4}$$

To customize the generic controller for a specific trajectory, the values $\{z_1, p_1, p_2\}$ must be specified. For the overall system to be stable the absolute value of p_1 and p_2 needs to be less than one. In the pured delay case, no transient behavior is desired, and $z_1 = z_2 = p_1 = 0$ specify that the system should reach $\bar{r}$ as quickly as possible. Otherwise, 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 $p_1 \leq z_1 \leq p_2$, 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 behavior slowly converges to the desired value. $p_1 = 0.1$, $p_2 = 0.8$ and $z_1 = 0.7$ does not present any oscillations or overshoots but has a slow convergence.

SEEC's generic controller calculates a speedup using an equation that produces values in a set of real numbers $(1, \infty)$; however, this speedup must be realized in a computing system which is limited to a discrete set of actions. For example, consider a system that allocates cores in a dual-core computer. This system can achieve speedups of 1 and 2, but the controller decide on a speedup of 1.5. To convert the continuous speedup signal into a set of actions achievable in a discrete system, the SEEC framework computes a set of actions to take over a window such that the average speedup over that window is the desired speedup. In the example, SEEC will decide on a speedup of 2 for half the window and a speedup of 1 for the other half, thus achieving an average of 1.5.

Given a target speedup $\bar{s}$ determined by Equation 3 and a discrete set of actions a_i with associated speedups s_i, the SEEC framework determines which actions to take over the next time window using the following algorithm. First, the framework finds an action a_{i+1} such that s_{i+1} is the smallest speedup for which $s_{i+1} \geq \bar{s}$. Thus, $s_i < \bar{s}$. For a window of τ time units, SEEC computes the amount of time τ_{i+1} to apply speedup s_{i+1} and the amount τ_i to apply speedup s_i as

$$\begin{aligned} \tau_{i+1} &= \tau\,\frac{s_{i+1} - \bar{s}}{s_{i+1} - s_i} \\ \tau_i &= \tau - \tau_{i+1} \end{aligned} \tag{5}$$

Thus, SEEC takes the desired heart rate and observed heartbeat data from an application and uses it to decide what speedup to apply at time k to meet the application's goal. Having done so, SEEC then translates this speedup into a set of actions that can be realized by the discrete computing system. SEEC's control framework is general, allowing system's developers to customize it for specific behavior as desired.

We note that the details described in this section are implemented automatically by the SEEC framework. System software developers do not need to understand how the control system works to make use of SEEC, although they can optionally customize the control behavior by setting the values of $\{p_1, p_2, z_1\}$ as described above.

3.3 Taking Action in SEEC

The SEEC control system determines a series of speedups to apply to modify an application's heart rate. Affecting these speedups requires that the system take some action. Thus the SEEC frame-

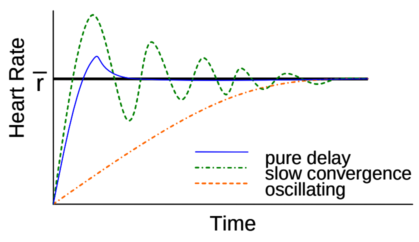


Figure 5: Possible trajectories of SEEC controllers.

work needs three inputs from the systems developer: a set of possible actions, the speedups associated with these actions, and a function that can take a specified action. For example, to implement a controller allocating cores in a dual-core system, the systems developer would specify two actions (allocating one or two cores with associated speedups of one and two). In addition, the system's developer might specify a function that manipulates affinity masks to affect the assignment of cores to an application.

Using this information, the SEEC framework is responsible for mapping speedups into actions and calling the function provided by the system's developer to realize those actions. Consider the core allocator from the previous paragraph and suppose the SEEC controller determines that a speedup of 1.5 is desired over some time interval. The SEEC framework will achieve this speedup by calling the provided function to assign two cores for half the time interval and then assigning one core for the second half. Using the action set provided by the systems developer, the SEEC framework automatically maps the decisions of the control system into actions.

We note that the system software developer is responsible for specifying a set of actions, the speedup of these actions, and a function to implement the actions. Additionally, the systems developer may choose to customize the trajectory of the controller as discussed above. The developer provides this information to the SEEC framework, which automatically monitors applications, uses its control system to make decisions, and uses the provided function to convert those decisions into actions. Section 4 provides five examples developing self-aware system software using SEEC.

4. USING THE SEEC FRAMEWORK

To illustrate the development of system software using SEEC, we describe five different examples. To demonstrate the broad applicability of SEEC, these example systems operate on a range of mechanisms including: processor speed, allocation of processor cores, access to DRAM, system power, and algorithm changes. Of course, this is a small subset of the many possible types of adaptive systems that can be built with SEEC, which supports any system that can take action to influence application speed.

To use SEEC in a given system, the systems software developer must specify a set of actions that can be taken, a speedup associated with each action, and a function that realizes the desired action. Table 2 summarizes these values for each of the five example systems. We note that each action often represents a tradeoff. For example, changing the processor speed may increase application performance, but it will also increase system power. Therefore, the table includes a summary of the tradeoffs inherent in each controller's actions.

Given that actions represent tradeoffs, system developers would like to ensure that SEEC can make the optimal tradeoff. For example, we would like the adaptive video encoder to maximize the quality when achieving a given speedup. As described in Section 3, SEEC selects the action that has the minimum speedup required to meet a goal. If the set of actions presented to SEEC represent the Pareto-optimal subset of all possible actions, then SEEC's control system will select the action that meets the target performance with optimal trade off. For example, consider a video encoder that can execute three possible actions: the default algorithm with speedup 1 and quality 1, algorithm A with speedup 1.5 and quality .8, and algorithm B with speedup 2 and quality .9. In this case, A is not Pareto-optimal and should not be in the set of actions that the systems developer specifies for SEEC; instead, the SEEC controller should only decide between the default algorithm and B.

4.1 Frequency Scaler

The first controller uses dynamic voltage and frequency scaling (DVFS) to adjust clock speed. As described in Table 2 the controller acts by adjusting the clock speed to one of the possible settings. On the experimental platform used in the next section there are seven possible settings. The lowest setting is assumed to provide speedup of 1 and all other setting are assumed to provide speedup proportional to their difference in clock speed. We implement the actuator using the `cpufrequtils` package in Linux. Increasing clock-speed increases performance, while increasing system power. For the frequency scaler all possible states are assumed to be Pareto-optimal.

4.2 Core Allocator

Our second control system allocates cores in a multicore processor. This controller acts by changing the affinity mask of processes (or threads in a process) thus altering the number of cores on which the process is permitted to run. Binding a process to a single core is assumed to provide a speedup of one. We assume that additional cores increase speedup according to $s_n = n^{7/8}$ where s_n is the speedup of using n cores. For this system we do not assume linear scaling with the number of cores. Empirically we find that few systems achieve truly linear scaling and the core allocator achieves better results in practice with this model. Increasing the number of cores in use increases application performance while increasing system power. For the core allocator, all states are again assumed to be Pareto-optimal in the target system.

4.3 DRAM Allocator

The first two controllers in the system alter the amount of compute resources available to applications; however, the framework also supports development of controllers for other needs. Thus, we examine is designed to alter the amount of available memory bandwidth by changing an application's NUMA mappings and allowing access to additional DRAM controllers. This system affects change by suspending the processes and threads in an application, allocating new memory from a new set of DRAM controllers, copying the contents of old to new, and freeing the old memory. We assume that the speedup using a single memory controller is 1 and that $s_n = n$, where n is the number of DRAM controllers used, which provides good results for memory bound applications. Increasing the number of DRAM controllers in use will increase application performance at a cost of increased system power.

4.4 Power Manager

All three of the previous controllers alter the number of resources assigned to an application and affect the system power. Therefore, it is possible to develop a single controller which allocates power by managing all three resources simultaneously. For this controller the possible actions are simultaneously adjusting the number of DRAM controllers, the number of cores, and the frequency of these cores. In our experimental platform (described in Section 5) this

Table 2: Summary of SEEC Controllers

Controller	Action Set	Actuation Function	Tradeoffs
Frequency Scaler	CPU Speeds	Change CPU speed	Power vs Speed
Core Allocator	Number of available cores	Change affinity masks	Power vs Speed
DRAM Allocator	Number of available DRAM controllers	Change NUMA page allocation	Power vs Speed
Power Manager	CPU speed and in-use cores	Change CPU speed and affinity masks	Power vs Speed
Adaptive Video Encoder	Encoding Parameters and Algorithm	Change parameters, use different algorithms	Video Quality vs Speed

results in 112 possible actions. However, in profiling the power and performance of this system we find that only 50 of these possible actions are Pareto-optimal, thus SEEC reasons only about these 50 states. To affect an action the controller changes the clock speed, affinity masks, and NUMA mappings as described above. To eliminate unnecessary memory management in CPU bound applications we write the action function such that it will not take an action that allocates multiple memory controllers unless the observed cache miss rate is above 20%. This policy shows how the direct performance metrics provided by Application Heartbeats can be used in conjunction with low-level metrics gathered from performance counters to build more capable and efficient systems.

4.5 Adaptive Video Encoder

As our final example of a self-aware controller, we develop an adaptive application that alters its algorithm to meet a performance goal. Specifically, we improve the adaptive video encoder developed in [16] using the x264 source code [41]. The original system used heuristics to control its adaptation. Using SEEC's control system we build a controller that accomplishes the same goal but does so with predictable and well-founded methods. In this example, the actions alter the algorithms the encoder uses to find temporal redundancy in a raw input video. We define the set of algorithms specified by the default command line for the PARSEC native input to have a speedup of 1. To find additional actions and speedups we profile x264 using different algorithms to encode 4 inputs and measure different algorithms' effects on speedup and quality. Out of the 560 possible actions, only 54 are Pareto-optimal (many algorithms are not meant to be used in conjunction). x264 rechecks a set of control parameters for every frame; to realize an action dynamically, we modify the values of x264's internal control parameters which will change the algorithm used in the next frame.

5. EVALUATION

This section describes several experiments designed to evaluate the generality, applicability, and effectiveness of the SEEC framework. First, we describe the experimental platform. Then, we demonstrate the predictability of the SEEC framework by showing the five controllers (described in Section 4) achieving a performance goal using the desired trajectory. The next subsection illustrates the broad applicability of SEEC for a range of applications by controlling the performance and power consumption of the PARSEC benchmarks. Then, we show how SEEC can be used to adaptively perform constrained optimization, by maximizing a video encoder's performance per Watt over a range of inputs each with differing compute requirements. Finally, the adaptability of the SEEC framework is illustrated by showing how the system can maintain performance in the face of fluctuations in the underlying compute environment.

5.1 Experimental Platform

All experiments are run on a Dell PowerEdge R410 server with two quad-core Intel Xeon E5530 processors running Linux 2.6.26. The processors support seven power states with clock frequencies from 2.4 GHz to 1.6 GHz. The `cpufrequtils` package enables software control of the clock frequency (and thus the power state). Consumed power is measured by a WattsUp device which samples and stores power at 1 second intervals [1]. All benchmark applications run for significantly more than 1 second so the sampling interval should not affect results. The maximum and minimum measured power ranges from 210 watts (at full load) to 80 watts (idle), with a typical idle power consumption of approximately 90 watts.

To account for the overhead of the SEEC control system we measure the time it takes to make a new decision, which requires calculating a speedup given a new observation of the heartbeat data. Our SEEC implementation is able to make a decision in 20.09 nanoseconds, which is close to 50 million decisions per second. We conclude that SEEC can likely make decisions much faster than the system can take action, or even record heartbeats so we conclude that SEEC's decision engine is low-overhead. There are other overheads, but those will be implementation specific and include the overhead of registering a heartbeat and the overhead of taking an action.

5.2 Predictable Behavior

We begin with a study to demonstrate the behavior of the five example systems (described in Section 4) built using the SEEC framework. For the first three systems we show the behavior while managing the PARSEC swaptions benchmark. For the DRAM allocator we show the system managing the STREAM benchmark [28]. For the adaptive video encoder we show x264 managing itself.

For each controller we first measure and record the minimum and maximum performance available through static allocation of resources. We then instantiate three different versions of each controller: pure delay, slow convergence, and oscillating. For each system, we set the application to request a target performance that is average of the maximum and minimum. The goal is predictably achieve the target heart rate following the desired trajectory.

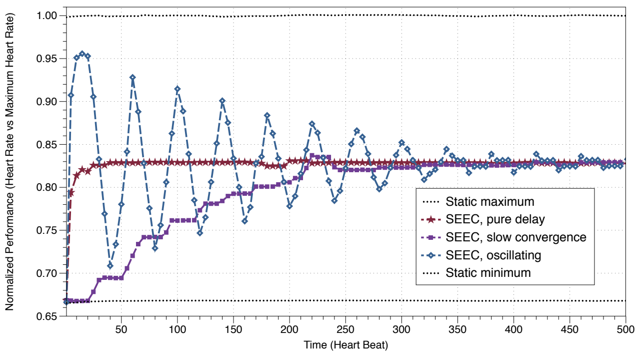


Figure 6: swaptions controlled with the SEEC frequency scaler.

The results of this study for all five systems are shown in Figures 6–10. For each chart, the x-axis shows time while the y-axis shows performance normalized to the maximum value. The min-

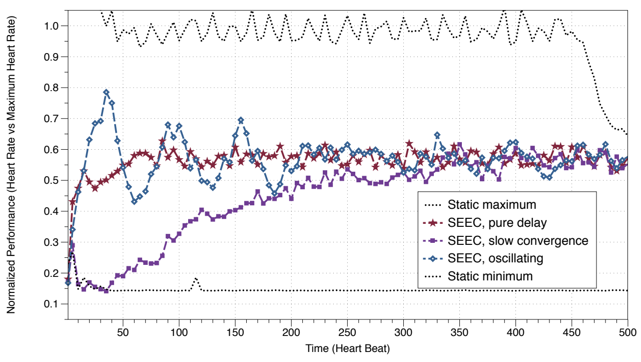


Figure 7: swaptions controlled with the SEEC core allocator.

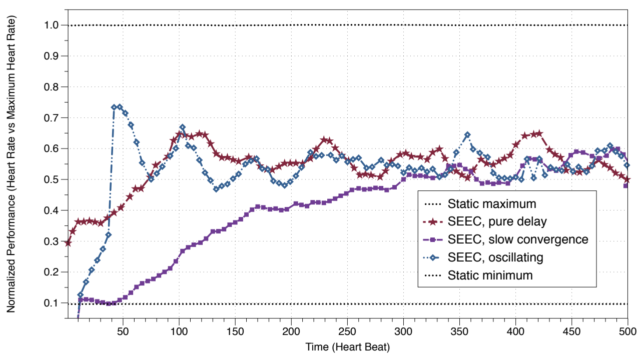


Figure 8: swaptions controlled with the SEEC power manager.

imum and maximum performance of the system are shown with dotted lines, while the behavior of the SEEC systems are shown with dashed lines.

Figure 6 shows the behavior of the frequency scaler managing swaptions. In this scenario, swaptions is run on a single core and provides perfectly sound feedback, which allows the controller to achieve the desired behavior exactly. Figure 7 shows the behavior of the core allocator, again managing swaptions. In this case, we use the parallel version of swaptions and the noise in the feedback system increases as we add cores. Despite this noise, the figure shows that all controllers converge to the desired performance, although the oscillating controller's curve is distorted. Figure 8 shows the behavior of the power manager controlling swaptions. Again all the curves converge to the desired performance with some distortion due to noise.

Figure 9 shows the behavior of the DRAM allocator managing the STREAM benchmark. In this case we use STREAM because none of the PARSEC benchmarks exhibit significant response to DRAM bandwidth changes on our system. For this experiment we increased the size of the vectors in the stream benchmark to 1.6MB and ran the benchmark for 1000 iterations. As shown in the figure all controllers converge to the desired behavior. This is notable because the DRAM allocator on our system has only two states, 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 10 shows the behavior of the adaptive x264 encoder using the PARSEC native input. Again, the controller is able to achieve the desired performance. In 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.

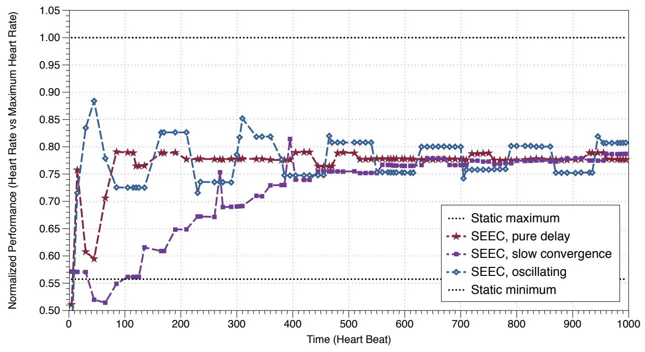


Figure 9: STREAM with the SEEC memory allocator.

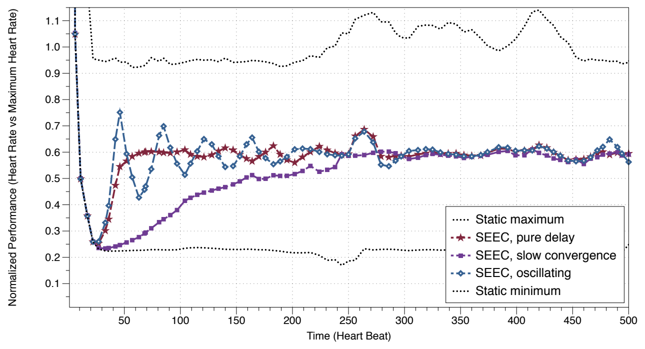


Figure 10: Adaptive video encoder built with SEEC.

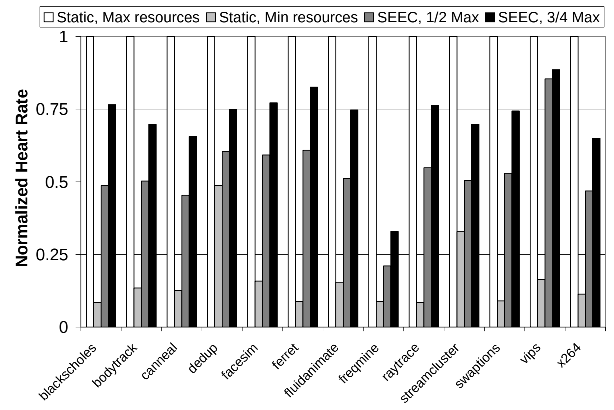
In summary, these results illustrate the generality and extensibility of the SEEC framework. While each system works with a different set of mechanisms, they are all built using the SEEC framework, which is shown to predictably converge to the set point and do so following the desired trajectories. This convergence is achieved despite the fact that the heartbeat signal is noisy for some systems. 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.

5.3 SEEC and the PARSEC Benchmarks

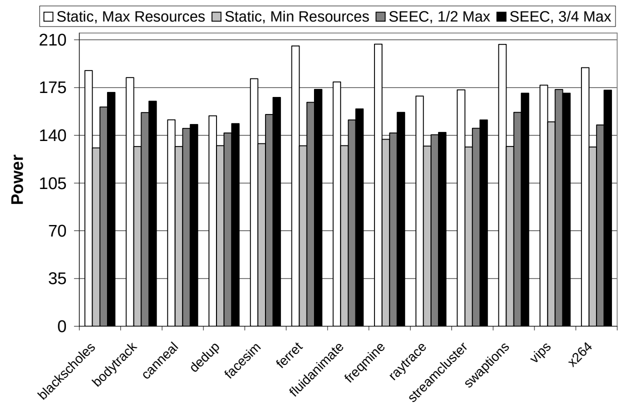
This section demonstrates the broad applicability of the SEEC framework and shows how its incorporation of application specified goals can be used to reduce resource consumption (in this case, power). For this study we instrument each of the PARSEC (v2.0) benchmarks using the Application Heartbeats API. We then measure the performance of each benchmark using its native input set when statically allocated eight cores set to the highest clock speed. Knowing the maximum achievable performance, we then run each benchmark with lower performance goals under the control of the self-aware power manager described in Section 4.4, recording the performance and total system power.

The results of this experiment are shown in Figures 11(a) and 11(b). Figure 11(a) shows the normalized performance of each benchmark when it is given the maximum resources, as well as the performance of the benchmark when requesting a heart rate of 50 and 75% of the maximum. Figure 11(b) shows the corresponding power consumption of each benchmark for each performance target.

This study demonstrates several characteristics of the SEEC framework. First, PARSEC consists of a variety of important multi-



(a) Performance



(b) Power

Figure 11: PARSEC benchmarks under SEEC.

core benchmarks, so SEEC's ability to control these benchmarks shows the framework is applicable to a broad range of applications. Adding Heartbeats to the PARSEC benchmarks is straightforward requiring each benchmark be augmented with the same five lines of code described in detail in Section 2. Second, this study demonstrates the predictability of SEEC's control system as eleven of the thirteen benchmarks are within 85% of their requested performance, indicating that the SEEC framework can control applications with predictable results. Finally, this study shows how the SEEC framework can be used to perform constrained optimization, allocating the minimal amount of resources required to meet the desired performance goal. For example, consider swaptions, which consumes 207 watts when allocated all resources in the machine. If the user only needs 75% of the maximum performance for swaptions, the self-aware resource manager can save 36 watts of total system power (17%). Alternatively, if only 50% of maximum performance is required, then the SEEC resource manager can save 50 Watts (24% of total system power). We note that each benchmark has a separate set of power/performance tradeoffs and the SEEC framework is capable of navigating these tradeoffs for each individual benchmark. The SEEC controller is able to make these tradeoffs without incorporating any prior knowledge of the benchmark itself; it only uses Heartbeat data observed during the application's execution.

The two benchmarks from this study which are more difficult to control are freqmine and vips, which do not provide sound feedback. For example, the bulk of processing in freqmine is done through recursive function calls. While it is easy to add heartbeats to this application, the time spent in each instance of the function call depends on the data processed at the leaves of the recursion. For freqmine we see performance vary by more than 10× from one heartbeat call to another. The vips benchmark is structured around a thread-pool and we insert a heartbeat each time a thread completes an assigned function; however, the time spent in these functions varies tremendously from one call to another. For these applications, Heartbeats can serve as a signal that the application is still alive, but the lack of sound feedback makes the control unpredictable. It is possible that an application expert could place the heartbeat calls into freqmine and vips in such a way as to provide sound feedback.

5.4 Maximizing Power/Performance Tradeoffs

This section presents an in-depth study of optimizing resource consumption while maintaining a performance goal. Specifically, we use the SEEC power manager (discussed in Section 4.4) to optimize performance per Watt for a video encoder. Furthermore, we want to achieve this optimization over a large range of inputs, each with differing compute requirements. This study uses fifteen different 1080p videos obtained from xiph.org, plus the native input from PARSEC.

To begin, x264 is modified to emit a heartbeat every frame, and it requests a heart rate of 30 frames per second. Additional modifications are made so that when the encoder detects a heart rate of less than 25 frames per second it "drops" the current frame, skipping its encoding and moving to the next frame. To evaluate the performance of the encoder, we measure the number of dropped frames for each input and record the fraction of frames which are encoded (not dropped). This performance metric rewards the system for reaching the target performance, but provides no extra benefit for exceeding the goal, as appropriate for systems with real-time goals like video encoding.

Given this performance metric, we run x264 without a resource manager, statically assigning it a set number of cores and a set clock-speed. We do this for all 56 possible combinations of cores and clock-speeds in the system. We then measure the power and performance for x264 on each of the 16 inputs and compute the performance per Watt, recording the best and worst static assignment of resources for each of the inputs. Having done so, we run x264 for each of the inputs using the SEEC power manager and again compute performance per Watt. This allows us to compare the performance per Watt under SEEC to the best and worse static assignments for each input. We note that no single static assignment is the best for all inputs. For example, with blue_sky.yuv the best static assignment of resources is 3 cores at maximum clock speed, while for ducks_take_off_1080p.yuv, the best static assignment is 6 cores at maximum clock speed. The fact that there is no single static assignment that is best for all inputs shows the difficulty of this optimization problem. Furthermore, this is a common problem for video encoders as they will routinely be confronted with previously unseen inputs.

Figure 12 shows the results of this 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 best static assignment. For each input, the first bar represents the worst static assignment, the middle bar represents the best static assignment, and the last bar represents the performance per Watt for x264 run with the SEEC power manager.

As shown in Figure 12 the SEEC-based power manager is able to adapt its behavior to individual inputs, delivering performance per Watt that is close to or better than the best static assignment

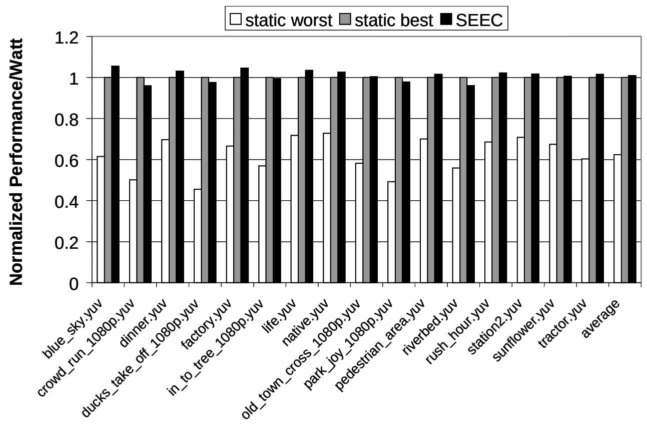


Figure 12: x264 performance per Watt.

of resources. The SEEC power manager is always within at least 95% of the best static assignment; however, for 10 of the 16 inputs, it outperforms the best static assignment by a small margin. In fact, on average the SEEC power manager provides slightly better performance per Watt than the best static assignment.

We note that in practice it would be impossible to achieve the best static assignment for an input without an oracle that can provide the correct assignment for unseen videos. In contrast, the SEEC power manager is able to achieve performance similar to the best possible without requiring an oracle, because it adapts behavior based on feedback from the application. In practice, many real-time systems simply allocate resources for the worst case scenario. Therefore, we expect SEEC to provide an even greater advantage in practice because worst case execution will result in wasted resources for most of the inputs.

SEEC out-performs static allocations by adapting its behavior to inputs whose needs vary during execution. For example, the middle section of the PARSEC native input is easier to encode than the beginning and the end. During this section, the SEEC control system detects an increase in performance and is able to reduce the amount of resources assigned to the encoder while still meeting its goal. This behavior is illustrated in Figure 13, which shows the power consumption of x264 encoding the PARSEC native input running with both its best static allocation and SEEC. Such an adaptation allows the SEEC to save power over even the best static assignment which lacks flexibility to adjust.

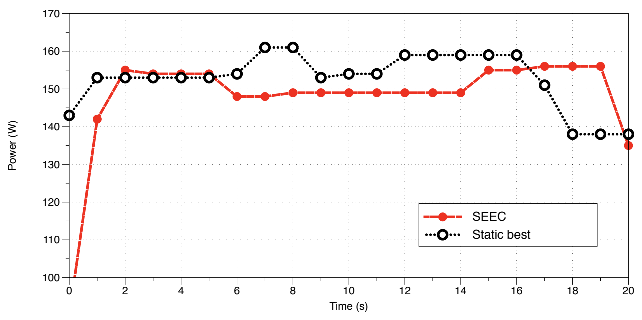


Figure 13: Power consumption of native input.

These results show several advantages of self-aware computing using the SEEC framework. First, they demonstrate how programmer burden can be reduced. With SEEC, the video programmer only has to use the Heartbeat interface and then SEEC-based systems can optimize the behavior of the program. With static assignments, the video programmer is responsible for the laborious task of profiling and understanding resource management within the system. Second, this study shows the optimality of the SEEC framework. Even though no single static assignment is best for all videos, the SEEC manager is able to adapt its behavior to find an assignment that is close to or better than the best static assignment for each input. Finally, this experiment shows the adaptability of the SEEC framework as videos with multiple regions with differing needs can be allocated the optimal amount of resources for each region. The next section explores this adaptability further.

5.5 Adapting to Hardware Changes

This section illustrates how the SEEC framework can adapt behavior to maintain application performance in fluctuating hardware environments. Toward this end, we present two scenarios: the first shows the core allocator reacting to a change in processor speed and the second shows the adaptive x264 reacting to a change in the number of available cores.

We begin with the PARSEC swaptions benchmark, running it on a single core at the maximum clock speed (2.4 GHz) and swaptions requests a performance that is achievable at this speed. Approximately one-quarter of the way through the execution of the program the clock speed is dropped to its lowest setting (1.6 GHz). This might occur if the system detects that the chip is too hot and reduces clock-speed to reduce power. We measure the performance of swaptions both when run by itself and with the core allocator described in Section 4. We note that meeting this performance represents a particular challenge for the core allocator as, ideally, it would allocate 1.5 cores to the application to make up for the loss in clock frequency. As such an allocation is not possible, SEEC will closely manage the number of cores in order to meet, but not exceed, the desired performance.

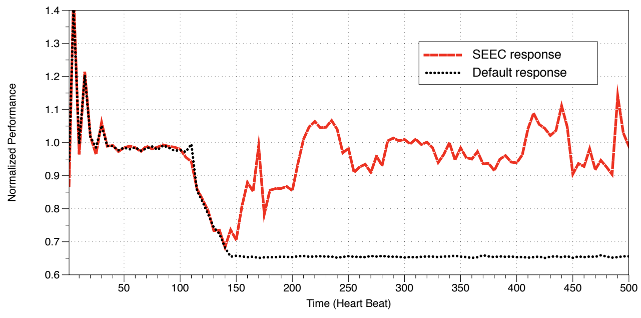
In the second scenario we explore the adaptive video encoder described in Section 4.5. We begin with the encoder running on eight cores with the default PARSEC command line and a performance goal that requires all eight cores. Approximately one-quarter of the way through execution we use the Linux `taskset` utility to force the encoder onto five cores. This simulates a loss of compute resources that may occur due to core failure or an unforeseen spike in system load. We measure the reaction of both the original x264 code and the SEEC version.

The results of these two experiments are shown in Figures 14(a) and 14(b). These charts show time on the x-axis (measured in heartbeats) and normalized performance on the y-axis. The behavior of the applications without SEEC (labeled "default response") is represented with a dotted line, while the behavior of the SEEC framework (labeled "SEEC response") is represented with a solid line.

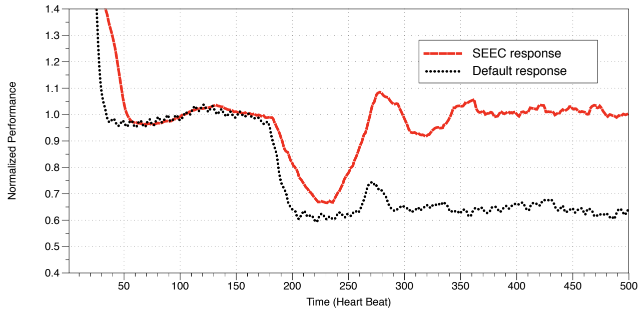
In the frequency scaling scenario, Figure 14(a) shows that, the default response without adaptation does nothing and swaptions' performance falls to two-thirds of its original value. In contrast, the core allocator uses the SEEC control system to detect the drop in swaptions' heart rate and adjusts the required speedup and number of cores to keep performance at the desired, original value.

In the core loss scenario, Figure 14(b) shows that the default response without adaptation does nothing and x264's performance falls. In contrast, the SEEC control system which detects the drop in performance and adjusts x264's algorithms to keep performance at the desired, original value.

The SEEC framework adapts without directly detecting the frequency change or the change in number of cores. Instead, these systems use their direct observations of application performance reflected in the heart rate. The SEEC control system uses this change to drive a decision in the amount of speedup to apply to an applica-



(a) swaptions with clock frequency changes



(b) x264 with simulated core failures

Figure 14: Adapting to hardware changes with SEEC.

tion. The control system then takes an action to deliver the desired speedup returning performance to the desired level.

The generality of the SEEC approach means that the system is not limited by an ability to detect specific conditions or events but can respond to arbitrary environmental fluctuations. Any event that alters performance will be detected by the SEEC framework and be fed as input to the control-based decision making process, allowing the system to adapt its behavior. Thus, the SEEC framework can aid fault tolerance and detection by providing a general way to detect changes in application performance and a general way to decide on what actions to take in the face of such performance fluctuations.

6. RELATED WORK

A self-aware, adaptive, or autonomic computing system is able to alter its behavior in some beneficial way without the need for human intervention [32, 2, 17]. Such a system can observe its behavior, make decisions, and take actions to meet desired goals at runtime. This ability to adapt promises to reduce the burden modern computing systems place on application developers; however it also rises new challenges for the creation and the usage of such systems [12]. Some example systems that adapt behavior include multicore chips that manage resource allocation [9], provide resources for critical sections [35], optimize for power [23], and assemble heterogeneous cores from many small cores[19]. In addition, languages and compilers have been developed to support adapting application implementation for performance [39, 4] or power [6, 33]. Operating systems are also a natural fit for self-aware computation [11, 22, 29, 21]. Self-aware techniques are also prominent in industry; companies such as IBM [17] (e.g., IBM Touchpoint Simulator, the K42 Operating System [22]), Oracle (e.g., Oracle Automatic Workload Repository [30]), and Intel (e.g., Intel RAS Technologies for Enterprise [18]).

While self-aware computing has been used to build a variety of systems, we find that there are several common trends in implementations. First, while these systems are designed to help applications meet their goals (e.g., the PEM [5] technology in the IBM K42 Operating System [22, 7]), many rely on inferring application behavior from low-level metrics like performance counters (e.g. Performance API (PAPI) project [38]) or high-level metrics like system throughput (e.g. the adaptive memory controller in [20]). An exception is Green which uses application goals for the quality of the answer, but not application speed [6]. In addition, many existing adaptive systems have the adaptation mechanism tightly coupled with the specific system being built. For example, the multicore processor described in [9] can adaptively assign resources to applications and does so under the control of a neural network and this decision engine is tightly coupled with the specific architecture. Finally, many adaptive systems rely on either machine learning (e.g. [20, 9, 13]) or system specific decision engines (e.g. [35, 33]).

The SEEC framework presented here differs from existing self-aware approaches in several respects. SEEC is designed as a general framework that supports applications and system software working together in a self-aware manner. In the SEEC model applications directly specify their goals, while system software specifies a set of possible actions. This approach is general in that it uses Application Heartbeats [16] allowing any application to specify goals, and it uses a simple framework wherein systems specify actions. In addition, SEEC uses a generic control system as its decision engine, while this control system is easily extended to new systems it is also capable of provided predictable behavior and guaranteed convergence to an application specified goal.

As self-aware systems mature, they will likely use a combination of these techniques. For example, application performance must be inferred when applications provide no feedback or when the feedback is too noisy to be useful; however, when direct feedback and goals are available they open up a new range of possibilities. In addition future systems will likely benefit from a combination of machine learning and control theoretic techniques. When applications require predictable performance and the system response can be adequately modeled, control theory provides known responses to stimuli. For systems where the interaction is too complicated to accurately model machine learning is an alternative.

Hellerstein et al [15] and Karamanolis et al [21] have both suggested that control systems can be used as “off-the-shelf” solutions for managing the complexity of modern systems, especially multi-tiered web-applications. While these authors suggest that existing control solutions can be used, the development of such systems still requires identification of a feedback mechanism and translating an existing control model into software. This leads to solutions that address a specific computing problem using control theory, but do not generalize [37, 36, 25]. For example, in [14] the specific problem of building a controller for a .NET thread pool is addressed.

One approach to generalizing control is ControlWare, middleware designed to create a generic control interface for managing Quality of Service in Internet applications [42]. Both ControlWare and SEEC turn a user desired service level into a set point for a control system; however, in the case of ControlWare, the system uses a difference equation model based on performance traces rather than direct feedback from sensors. In our proposal, the system is completely modeled and the translation into different actuator actions is characterized based on the feedback provide by Application Heartbeats.

In summary, the SEEC control system differs from existing approaches to using control in computing because SEEC does not solve a specific problem, but provides a general control strategy using a widely applicable feedback mechanism. SEEC's control strategy provides a common decision making infrastructure which is easily customized when applications specify goals and system software specifies actions.

7. CONCLUSION

This paper has presented the SEEC framework for self-aware computing. SEEC enables a new computational model where applications specify their goals, system software specifies possible actions, and the framework observes the system using a decision engine to select and execute actions as necessary to reach the desired goals. SEEC has three distinguishing features: it addresses the needs of both applications and system software in a unified framework, it directly incorporates application goals, and it uses a general control-theoretic decision engine that is easily customized for specific needs. We have implemented several self-aware systems with SEEC and found that it predictably achieves application-specified goals. Furthermore, the SEEC framework is easily extended to a wide range of systems operating on different mechanisms. Finally, by constantly observing and re-evaluating its decisions, SEEC is able to adapt its behavior. This flexibility to change decisions and take new actions allows SEEC to minimize power consumption while maintaining performance as well as respond to changes in the available compute resources. My predictably managing resource usage and response to unforeseen events, we believe the SEEC framework has the potential to reduce some of the application programmer's burden.

8. REFERENCES

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

[2] Organic Computing Initiative OCI. http://www.sra.uni-hannover.de/orgcomp/, 2010.

[3] ANA, European Union Funded Project. Autonomic network architecture. http://www.ana-project.org, 2009.

[4] 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 *Conf. on Programming Language Design and Implementation*, Jun 2009.

[5] R. Azimi, C. Cascaval, E. Duesterwald, M. Hauswirth, K. Sudeep, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for whole-system characterization and optimization. pages 15–24, 2004.

[6] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2010.

[7] A. Baumann, D. D. Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT-2008: Proceedings of the 17th Inter. Conf. on Parallel Architectures and Compilation Techniques*, Oct 2008.

[9] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[10] J. Buisson, F. André, and J. L. Pazat. Dynamic adaptation for grid computing. *Lecture Notes in Computer Science. Advances in Grid Computing - EGC*, pages 538–547, 2005.

[11] 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.

[12] P. Dini, W. Gentzsch, M. Potts, A. Clemm, M. Yousif, and A. Polze. Internet, GRID, self-adaptability and beyond: Are we ready? Aug 2004.

[13] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *ICAC '10: Proceeding of the 7th international conference on Autonomic computing*, pages 215–224, New York, NY, USA, 2010. ACM.

[14] J. Hellerstein, V. Morrison, and E. Eilebrecht. Applying control theory in the real world: Experience with building a controller for the .net thread pool. *Sigmetrics Performance Evaluation Review*, pages 38–42, 2009.

[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 '10: Proceeding of the 7th international conference on Autonomic computing*, pages 79–88, New York, NY, USA, 2010. ACM.

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

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

[19] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, 2007.

[20] E. Ipek, O. Mutlu, J. F. MartŠnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch.*, 2008.

[21] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 9–15, Berkeley, CA, USA, 2005. USENIX Association.

[22] 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 '06: Proc. of the 1st ACM SIGOPS/EuroSys Euro. Conf. on Computer Systems*, 2006.

[23] 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.

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

[25] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):1014–1027, September 2006.

[26] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. Controlling software applications via resource allocation within the heartbeats framework. In *Proceeding of the 49th international conference on decision and control*, 2010.

[27] M. M. Masters. Exploring usability in mobile autonomic networks. In *MobileHCI '08: Proceedings of the 10th international conference on Human computer interaction with mobile devices and services*, pages 549–550, New York, NY, USA, 2008. ACM.

[28] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[29] S. Oberthür, C. Böke, and B. Griese. Dynamic online reconfiguration for customizable and self-optimizing operating systems. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 335–338, New York, NY, USA, 2005. ACM.

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

[31] P. Reinecke and K. Wolter. Adaptivity metric and performance for restart strategies in web services reliable messaging. In *WOSP '08: Proceedings of the 7th International Workshop on Software and Performance*, pages 201–212. ACM, 2008.

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

[33] 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*, pages 161–174, 2007.

[34] J. Strassner, S.-S. Kim, and J. W.-K. Hong. The design of an autonomic communication element to manage future internet services. In C. S. Hong, T. Tonouchi, Y. Ma, and C.-S. Chao, editors, *APNOMS*, volume 5787 of *Lecture Notes in Computer Science*, pages 122–132. Springer, 2009.

[35] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, pages 253–264, 2009.

[36] Q. Sun, G. Dai, and W. Pan. LPV model and its application in web server performance control. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, volume 3, pages 486–489, Washington, DC, USA, December 2008. IEEE Computer Society.

[37] M. Tanelli, D. Ardagna, and M. Lovera. LPV model identification for power management of web service systems. In *Proceedings of the 2008 IEEE Multi-conference on Systems and Control*, pages 1171–1176, Boston, MA, 2008. IEEE Control Systems Society.

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

[39] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 277–288, New York, NY, USA, 2005. ACM.

[40] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurr. Comput. : Pract. Exper.*, 17(2-4):235–257, 2005.

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

[42] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Proceedings of the 22nd International conference on Distributed Computing Systems*. IEEE computer society, 2002.