A Generalized Software Framework for Accurate and Efficient Management of Performance Goals

Henry Hoffmann[1] Martina Maggio[2] Marco D. Santambrogio[3]
Alberto Leva[3] Anant Agarwal[4]
[1]University of Chicago, [2]Lund University, [3]Politecnico Di Milano, [4]MIT
hankhoffmann@cs.uchicago.edu, martina.maggio@control.lth.se, marco.santambrogio@polimi.it,
leva@elet.polimi.it, agarwal@csail.mit.edu

ABSTRACT

A number of techniques have been proposed to provide runtime performance guarantees while minimizing power consumption. One drawback of existing approaches is that they work only on a fixed set of components (or actuators) that must be specified at design time. If new components become available, these management systems must be redesigned and reimplemented. In this paper, we propose PTRADE, a novel performance management framework that is general with respect to the components it manages. PTRADE can be deployed to work on a new system with different components without redesign and reimplementation. PTRADE's generality is demonstrated through the management of performance goals for a variety of benchmarks on two different Linux/x86 systems and a simulated 128-core system, each with different components governing power and performance tradeoffs. Our experimental results show that PTRADE provides generality while meeting performance goals with low error and close to optimal power consumption.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes; D.4.8 [**Operating Systems**]: Performance—*Measurements, Monitors*; I.2.8 [**Problem Solving, Control Methods, and Search**]: Control Theory

General Terms

Performance, Design, Experimentation

Keywords

Adaptive Systems, Self-aware Computing, Power-aware Computing

This work was funded by the U.S. Government under the DARPA UHPC program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

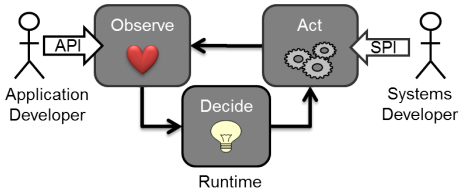


Figure 1: Overview of PTRADE.

1. INTRODUCTION

Researchers have proposed *performance management* techniques that control a single component in a computer system (*e.g.*, core allocation, DRAM, processor speed) in order to guarantee performance while minimizing power consumption. For example, systems have been built that manage dynamic voltage and frequency scaling (DVFS) [43], core throttling [45], core allocation [26], cache [2], DRAM [46], and disks [24].

It has been observed, however, that coordinated management of multiple components is more efficient than managing a single component [4, 13, 17, 29]. Thus, several researchers propose performance management frameworks that coordinate multiple components (*e.g.*, cores and processor speed [6, 27], processor and DRAM [25], cores, cache, and bandwidth [35]) in order to achieve greater power savings for the same performance targets.

Although these coordinated management frameworks provide increased power savings, they consider only a fixed set of components. However, different computer systems expose different sets of components for tuning. Therefore, each performance management framework needs to be redesigned and reimplemented to port to a new system. We would like to achieve efficient performance management on new systems without the burden of constant redesign, raising the question: *Is it possible to build a generalized performance management framework that provides performance guarantees and power savings comparable to those of a framework designed for a particular, fixed, set of components?*

We address this question by developing PTRADE, a multicore performance management framework that coordinates a general and flexible set of components. PTRADE can deploy on new platforms with new sets of components without redesign or reimplementation. PTRADE consists of two interfaces and a runtime system; which together comprise a closed-loop system, as illustrated in Figure 1. The first interface (for use by application developers) communicates an application's performance goals and current performance. The second interface (for use by systems devel-

opers) describes the system components available for management. The runtime system actively monitors application performance and tunes the available components to meet the application-specified performance goal while minimizing system power consumption. This tuning is accomplished in two phases. First, a generalized control system produces a generic control signal, independent of any specific components. Second, a translator converts this generic signal into specific settings for a particular system's available components. The control system keeps performance at the desired level, while the translator selects the combination of component settings that achieve this goal with minimal power consumption. PTRADE's controller and translator are adaptive and model application and component behavior with a series of Kalman filters [40]. Through adaptation, PTRADE dynamically tailors power consumption to meet the application's goals while reacting to the performance needs of a specific input or even phases within an input.

We evaluate PTRADE's ability to generalize performance management by deploying on three different platforms with different sets of components. Specifically, we use two real Linux/x86 systems and one simulated manycore based on the Graphite simulator [30]. On the Linux/x86 systems, we use the PARSEC benchmarks [3] plus two others designed to stress the system. On the simulated manycore, we use four SPLASH2 benchmarks [42]. We evaluate PTRADE's ability to meet performance goals by measuring *relative error*, or the percentage difference between the goal and the achieved performance. We evaluate power reduction by comparing the measured power consumption to that provided by an *oracle* that represents true optimal power consumption for a management framework that acts with perfect knowledge of the future and without overhead. We find that, on average across all benchmarks, PTRADE meets performance goals accurately (2.9% error on the real systems, $< 1\%$ error on the simulator). Furthermore, despite its generality, PTRADE is efficient and achieves close to optimal power consumption. On the real systems, PTRADE's average power consumption is within 6% of the oracle. On the simulated system, PTRADE's average power consumption is within 3% of the oracle.

This paper makes the following contributions:

- It proposes a generalized performance management framework that coordinates multiple components to control application performance while reducing power consumption.
- It designs a generalized, adaptive control system which ensures performance goals are met using a generic control signal, independent of the available system components.
- It designs a generalized, adaptive translator which transforms the generic control signal into specific component settings while minimizing power consumption.
- It demonstrates that a single, generalized performance management system can meet performance goals while providing close to optimal power consumption on different systems, both real and simulated.

The rest of this paper is organized as follows. Section 2 describes the PTRADE system. Section 3 presents the applications and systems used to evaluate PTRADE. The results of the evaluation are presented in Section 4. Section 5 discusses related work before concluding in Section 6.

2. PTRADE DESIGN

A generalized performance management system must work with a variety of applications and a variety of different systems components. PTRADE achieves this generality by asking application developers to specify performance goals and asking systems developers to specify components affecting performance/power tradeoffs. PTRADE's runtime system takes these specifications and tunes component usage to meet goals while reducing power consumption. Table 1 shows the responsibilities of each of these three roles in PTRADE development. A key feature of PTRADE is its separation of concerns; application developers do not need to understand system-level components and systems developers do not have to infer application-level performance. This section describes PTRADE's support for specification of goals, specification of components, and runtime management.

2.1 Specifying Application Performance

PTRADE uses the Application Heartbeats API [14] to specify application goals and progress. The API's key abstraction is a heartbeat; applications 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 original Heartbeats API includes functions for performance monitoring only. An API function allows other applications to examine the heartbeat data for an application. This data structure contains a log of all heartbeats with corresponding timestamps as well as various performance statistics, including the average heart rate since the beginning of program execution and the average heart rate over some application specified time window. PTRADE augments this data structure by inserting power and energy statistics. This additional data allows PTRADE's runtime system to observe the power consumption and energy for individual heartbeats and over the life of the application.

This modification of the Heartbeats API requires no additional work from application developers beyond inserting heartbeats into the application. Application developers are not required to interface with a power monitor, this is handled automatically by the implementation. The runtime is also insulated from the power monitor, as it retrieves information only from the Heartbeats API. Thus, the runtime is agnostic about whether power data comes from an on-chip meter [34], an off-chip meter [1], or a model [19].

2.2 Specifying System Components

PTRADE provides a separate, system programmer interface (SPI) for specifying components available for management. This interface is summarized in Table 2. The key abstraction in the SPI is a *control panel* populated with *actuators*. The PTRADE runtime exports a control panel and systems developers use the SPI to register new actuators. The actuator data structure includes: a name, a list of allowable settings, a function which changes the setting, the performance benefits of each setting, and the power costs of each setting. These costs and benefits are listed as multipliers over a nominal setting, whose costs and benefits are unity. Each actuator specifies a *delay*, or the time between when it is set and when its effects can be observed. Finally, each actuator specifies whether it works on only the application that registered it or if it works on all applications. This last feature allows applications to register application

Table 1: Roles and Responsibilities in PTRADE development.

Phase	Applications Developer	Systems Developer	PTRADE Runtime
Application Development	Specify goals and performance	-	
Application Execution	-	-	Manage component usage
Systems Development	-	Specify components	

Table 2: PTRADE SPI listing

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

specific actuators with the control panel.

A systems developer writes a program to register an actuator. This program first calls ACT_attach_control_panel to connect to the PTRADE control panel. It then calls the ACT_register_actuator function providing both the name of a text file and a name for the actuator. The text file has an enumeration of the attributes of the actuator (settings, costs and benefits, delay). Specifying these values in text aids portability as the same program can be used on different systems by changing the file. For example, the same program can register a DVFS actuator on machines with different clock speeds by simply changing the file. If the systems developer wants to disable an actuator, the ACT_delete_actuator function removes it from the control panel. Two query functions are used (primarily by PTRADE) to query the number of actuators and the different actuators available. These functions allow multiple systems developers to register actuators independently. The costs and benefits for actuators only serve as initial estimates and PTRADE's adaptive features overcome errors in the values specified by the systems developer. PTRADE allows these models to be specified to provide maximum responsiveness in the case where they are accurate.

2.3 Runtime

The PTRADE runtime automatically and dynamically sets actuators to meet application performance goals while reducing power consumption. The runtime is designed to handle general purpose environments and it will often make decisions about actuators and applications with which it has no prior experience. In addition, the runtime must react quickly to changes in application workload. To meet these requirements, PTRADE uses feedback control, but splits the control problem into two phases. In the first phase, a generalized *controller* produces a generic control signal. This control signal is independent of actuators in the system and represents how much the application should speed up at the current time t. In the second phase, a *translator* finds actuator settings which achieve the control signal while minimizing power consumption. The controller is responsible for ensuring goals are met, while the translator works to minimize power consumption. Critically for generality, both the controller and the translator are themselves *adaptive* and dynamically adjust internal models of application behavior and system performance/power tradeoffs online.

2.3.1 The Controller

PTRADE drives performance to a goal g by reading the current heart rate $h(t)$ and computing a control signal $s(t)$, where t represents time. $s(t)$ represents the speedup necessary at time t. $s(t)$ is defined relative to the application *workload* w, which represents the time between two heartbeats when all actuators are turned to their minimal settings. For example, if $s(t) = 1.1$ the control system is telling the translator to increase speed by 10% over the baseline.

PTRADE computes $s(t)$, by modeling the application heart rate $h(t)$ at time t:

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

where $\delta h_i(t)$ represents an exogenous disturbance in performance. PTRADE eliminates the *error* $e(t)$ between the performance goal g and the observed heart rate $h(t)$:

$$e(t) = g - h(t) \tag{2}$$

As PTRADE uses a discrete time model, we analyze its behavior in the Z-domain by examining the transfer function from the error to the control signal:

$$\frac{S(z)}{E(z)} = \frac{w \cdot z}{z - 1} \tag{3}$$

From this transfer function, the controller is synthesized following a standard procedure [12] and $s(t)$ is calculated as:

$$s(t) \quad = \quad s(t-1) + w \cdot e(t) \tag{4}$$

PTRADE's generalized controller acts on speedup rather than directly managing a specific actuator (*e.g.*, clock speed). This distinction means the control signal is separate from the mechanism used to achieve the control, which is essential for generality as it allows PTRADE to work with different sets of components (or actuators) that are available on different systems. PTRADE's approach contrasts with prior control-based management frameworks like METE [35] and ControlWare [44], which directly incorporate the actuator into the control equations. Thus, if METE is ported to a new system with different actuators, METE's control has to be reformulated and re-implemented. Of course, PTRADE's flexibility comes at a cost of added complexity, as the speedup signal must be converted to actuator settings by the translator as described in Section 2.3.2.

Adapting Control.

As the controller is based on classical control techniques, it will meet the performance goal provided that w is a reasonable approximation of the actual workload [27]. In a generalized setting, however, it is not possible to know the workloads of every application beforehand. Furthermore, we

expect applications to exhibit phases with different workloads. Thus, assumption of a time-invariant w is not sufficient. PTRADE would like to know $w(t)$, the workload at the current time, but it is not possible to measure this value without turning all actuators to their lowest setting and, thus, failing to meet the performance goal. Therefore, PTRADE treats $w(t)$ as a hidden value and estimates it with a one dimensional Kalman filter [40]; *i.e.*, it models the true workload at time t as $w(t) \in \mathbb{R}$:

$$\begin{aligned} w_i(t) &= w_i(t-1) + \delta w_i(t) \\ h_i(t) &= \frac{s_i(t-1)}{w_i(t-1)} + \delta h_i(t) \end{aligned} \tag{5}$$

where $\delta w_i(t)$ and $\delta h_i(t)$ represent time varying noise in the true workload and heart rate, respectively. The estimate of this true workload is denoted as $\hat{w}_i(t)$ and calculated as:

$$\begin{aligned} \hat{x}^-(t) &= \hat{x}(t-1) \\ p^-(t) &= p(t-1) + q(t) \\ k(t) &= \frac{p^-(t)s(t-1)}{[s(t)]^2 p^-(t) + o} \\ \hat{x}(t) &= \hat{x}^-(t) + k(t)[h(t) - s(t-1)\hat{x}^-(t)] \\ p(t) &= [1 - k(t)s(t-1)]p^-(t) \\ \hat{w}(t) &= \frac{1}{\hat{x}(t)} \end{aligned} \tag{6}$$

Where $q(t)$ and o represent the application and measurement variance, respectively. The application variance $q(t)$ is the variance in the heart rate signal since the last filter update. PTRADE assumes that o is a small fixed value as heartbeats have been shown to be a low-noise measurement technique [14]. $h(t)$ is the measured heart rate at time t and $s(t)$ is the applied speedup (according to Equation 4). $\hat{x}(t)$ and $\hat{x}(t)^-$ represent the *a posteriori* and *a priori* estimate of the inverse of workload at time t. $p(t)$ and $p^-(t)$ represent the *a posteriori* and *a priori* estimate error variance, respectively. $k(t)$ is the *Kalman gain* at time t.

PTRADE adapts control online by first updating Equation 6. $\hat{w}(t)$ is then substituted in place of the fixed value w when calculating the control signal using Equation 4.

We note that control theory is, in general, a discipline for building adaptive systems. Classical control techniques (like those used to formulate Equation 4) make the system under control adaptive. However, *adaptive control* implies something stronger, meaning the controller itself is updated (in this case by Equation 6). Therefore, a classical (non-adaptive) control scheme is simply a static set of equations whose coefficients do not change over time. An adaptive control scheme modifies itself in order to generalize and cope with possible non-linearities or unmodeled dynamics in the controlled system.

2.3.2 The Translator

Adaptive control produces a generic speedup signal $s(t)$ which must be translated into specific actuator settings. PTRADE does this by scheduling over a time window of τ heartbeats (referred to as a "decision period"). Given a set $A = \{a\}$ of configurations for multiple actuators with speedups s_a and power costs c_a, the runtime would like to schedule each configuration for $\tau_a \leq \tau$ heartbeats so that the desired speedup is met and the total cost is minimized:

$$\begin{aligned} \text{minimize}(\tau_{idle} c_{idle} + \tfrac{1}{\tau}\textstyle\sum_{a \in A}(\tau_a c_a)) \quad \text{s.t.} \\ \tfrac{1}{\tau}\textstyle\sum_{a \in A} \tau_a s_a &= s_i(t) \\ \tau_{idle} + \textstyle\sum_{a \in A} \tau_a &= \tau \\ \tau_a, \tau_{idle} &\geq 0, \ \forall a \end{aligned} \tag{7}$$

Note the *idle* action, which idles the system paying a cost of c_{idle} and achieving no speedup ($s_{idle} = 0$). Solutions to this system will vary with platforms and having a flexible set of solutions is essential for achieving generality.

Linear optimization provides a solution for some power management problems, but it does not scale to large sets of actuators [41]. Therefore, PTRADE approximates the solution to Equation 7 by selecting three separate candidate solutions, computing their costs (power consumptions), and then selecting the best of these candidates. Specifically, PTRADE considers: *race-to-idle*, *proportional*, and *hybrid*.

First, PTRADE considers *race-to-idle*; *i.e.*, setting actuators to achieve 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 $c_{race} = \tau_{max} \cdot c_{max} + \tau_{idle} \cdot c_{idle}$.

PTRADE then considers *proportional* scheduling by selecting from A, an actuator setting j with the smallest speedup s_j such that $s_j \geq s_i(t)$ and an actuator setting k such that $s_k < s_j$. Given these two settings, PTRADE configures the system at 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 PTRADE considers is a *hybrid*, where PTRADE finds a setting j as in the proportional approach. Again, s_j is the smallest speedup such that $s_j \geq s_i(t)$; however, PTRADE considers only actuator settings 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 PTRADE runtime system solves Equation 7 by finding the minimum of c_{race}, c_{prop}, and c_{hybrid} and and using the actuator settings corresponding to this minimum cost.

Adapting Translation.

The translator provides good solutions to Equation 7 assuming that the values of s_a and c_a are known for all combinations of actuators $a \in A$. Unfortunately, the true values of s_a and c_a are application dependent and not directly measurable online, so they need to be estimated dynamically. This situation is analogous to estimating the true value of application workload in the controller. Thus, PTRADE adopts the same approach, estimating s_a and c_a online as $\hat{s}_a(t)$ and $\hat{c}_a(t)$ using one dimensional Kalman filters. The specification of these filters is analogous to that presented in Equation 6, so it is omitted for brevity. Similar to the control system, the translator adapts its behavior by first estimating $\hat{s}_a(t)$ and $\hat{c}_a(t)$ online using the value of a selected for the prior decision period, and then substituting these estimates for the values in Equation 7.

3. USING PTRADE

This section describes the applications and platforms used to evaluate PTRADE.

3.1 Applications

We use two sets of benchmark applications. One set is run on the real machines. On the simulated machine, we are limited by the number of benchmarks ported to the system and modified to run on more than 100 cores. Therefore, we

use a restricted set of benchmarks. We discuss each set in more detail below.

Our benchmarks for the real system consist of the 13 PARSECs [3] plus dijkstra and STREAM [28]. PARSEC has a mix of important, emerging multicore workloads. dijkstra is a parallel implementation of single-source shortest paths on a large, dense graph developed for this paper. STREAM tests memory performance. These last two benchmarks test PTRADE's ability to handle resource limited applications. dijkstra has some parallelism, but does not scale beyond 4 cores, and it is more efficient to increase processor speed than allocate more cores. STREAM is generally memory bandwidth limited, but needs sufficient compute resources to saturate the available memory bandwidth.

Our benchmarks for the simulated system consist of four of the SPLASH2 benchmarks [42]: barnes, ocean non-contiguous, raytrace, and water spatial. The ocean benchmark runs on up to 128 cores but slows down as the number of cores increases beyond 8. Ocean benefits most from additional cache resources. In contrast, the other applications scale well with increasing numbers of cores but receive only marginal benefit from increasing cache size.

Although PTRADE's runtime can work with any performance feedback metric that increases with increasing performance (*e.g.*, instructions per second, etc), in this paper we use direct feedback from the application in the form of heartbeats [14]. Each application has been modified to emit heartbeats at important locations in the code. Note that these are not necessarily regular signals. In fact, these applications have a range of performance behaviors, so this suite tests PTRADE's ability to handle applications with both high and low variance in performance feedback. Table 3 illustrates this range. The variance data is gathered by running each benchmark, measuring the reported heart rate at each heartbeat, and computing the variance in heart rate signal. Benchmarks with regular performance have low variance, while benchmarks with irregular performance (some iterations much harder/easier than others) have high variance. Six of the 13 PARSECs (bold in the table) have high variance. To manage these benchmarks, PTRADE will adapt its internal models to the characteristics of each application including phases and variance within a single application.

The PARSEC benchmarks contain thirteen diverse, modern multicore workloads [3]. Interestingly, perhaps indicative of a modern emphasis on interactive applications, twelve of the thirteen might be deployed with soft real-time performance goals or quality-of-service requirements including examples from financial analysis (blackscholes, swaptions), media search (ferret), graphics (facesim, fluidanimate, raytrace), image/video processing (bodytrack, vips, x264), and streaming data analytics (freqmine, streamcluster). In addition, for applications with externally defined performance goals, reducing power consumption will reduce energy as well, leading to reduced cost or extended battery life. These benchmarks, therefore, represent good targets for testing PTRADE's ability to meet performance goals while minimizing power consumption.

3.2 Importance of Application Feedback

As noted in Section 2, PTRADE incorporates application level feedback in the form of heartbeats. This distinction is critical for applications that execute data dependent code; *i.e.*, where the processing changes based on the input data.

Table 3: Variance in Application Performance.

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

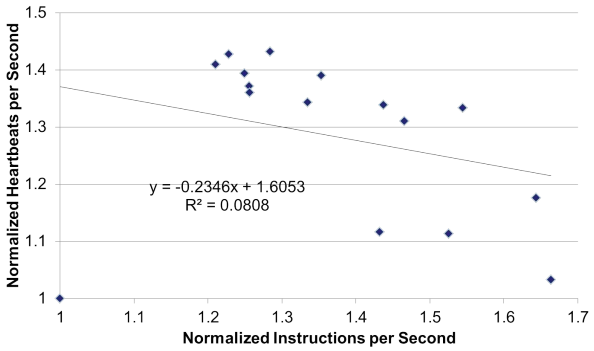


Figure 2: Instructions per second vs. heart rate.

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

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

3.3 Hardware Platforms

To demonstrate the generality of PTRADE, we deploy it on real Linux/x86 systems and a simulated many core system, summarized in Table 4. All three platforms have different components governing power and performance tradeoffs, summarized in Table 5. For each platform, this table lists the components available, as well as the maximum speedup and maximum increase in power consumption measured across all benchmarks. These values represent the ranges of each individual component when only that component is varied and all other components are held constant.

Table 4: Hardware platforms used in evaluation.

Name	Processor	Cores	Memory Controllers	Speeds (GHz)	TurboBoost	Variable Cache Size	Idle State
Machine 1	Intel Xeon X5460	8	1	2.000–3.160	no	no	yes
Machine 2	Intel Xeon E5520	8	2	1.596–2.395	yes	no	yes
Simulator	Graphite	128	1	0.100–0.500	no	yes	no

Table 5: Power/Performance Tradeoffs.

	Actuator	Max Speedup	Max Power Increase
Machine 1	Cores	7.95	1.57
	DVFS	1.59	1.56
	Idle Time	1.00	1.00
Machine 2	Cores	8.02	2.05
	DVFS	1.50	1.21
	Idle Time	1.00	1.00
	TurboBoost	1.07	1.15
	Mem. Cont.	1.81	1.27
Simulator	Cores	136.20	84.70
	DVFS	4.50	5.67
	Cache	2.10	2.02

As shown in Table 4, both Linux/x86 systems are 8-core Xeon servers. Both machines support clock-gating of inactive cores, so assignment of application threads to cores (through affinity masks) affects power/performance tradeoffs. Additionally, both machines support DVFS (accessed through `cpufrequtils`); Machine 1 supports four settings, while Machine 2 supports 8. Finally, both machines can suspend the current application and idle the processor. Machine 1's idle power consumes 94% of the lowest measured active power (200 W idle, 213 W lowest active power). In contrast, Machine 2 supports a low-power idle state that consumes 69% of the lowest measured active power (90 W idle, 131 W lowest active power). In addition to these shared components, Machine 2 supports assignment of memory controllers to an application (through `libnuma`) and TurboBoost. The power and performance gains available through these actuators are summarized in Table 5. We note that idling the application never increases performance or power consumption, but lowers both. To measure power consumption on these platforms, both are connected to WattsUp? power meters which report total system power consumption at 1 second intervals. We note that PTRADE makes decisions at speeds much faster than 1 Hz. When querying power at finer-granularity, the heartbeats interface uses a model to interpolate power consumption since the last measurement. The models are updated online every time new power feedback is available.

As shown in Table 4, the third experimental platform supports a different set of components available on the Graphite simulator [30]. Specifically, Graphite is configured to simulate the Angstrom processor [17]. Angstrom is designed to support a wide range of power and performance tradeoffs in hardware and expose these tradeoffs to software. Specifically, Angstrom allows software to adapt the number of cores assigned to an application (from 1-128), the L2 cache size of each core (from 32-128 KB, by powers of 2), and the voltage and frequency (from 0.4 – 0.8 V, and 100 – 500 MHz) of those cores. To provide power feedback, Graphite integrates 32 nm cache and core power models from McPat [23], 32 nm DRAM models from CACTI [36], and 32 nm network power models from Orion 2.0 [21]. In the simulations, power feedback can be read on a cycle by cycle basis. In all other respects, we maintain the default Graphite configuration.

We use the real systems to show PTRADE's ability to manage power goals on contemporary systems. The simulated system allows us to evaluate PTRADE on future many-core systems with larger configuration spaces. In addition, the simulator allows us to test PTRADE in an environment where fine-grain power feedback is available.

4. EVALUATION

We demonstrate PTRADE meeting performance goals and optimizing power. We first describe approaches to which we compare PTRADE and present the metrics used in evaluation. We then evaluate PTRADE on the Linux/x86 platforms and the simulated manycore.

4.1 Points of Comparison

We compare PTRADE's management to the following.

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

Uncoordinated Adaptation: In this approach, components are tuned individually, without coordination. All available actuators are used, but each is tuned by an independent instance of PTRADE. Comparing to this approach demonstrates the need for coordinated management.

Classical Control: This approach implements the control and translation system from Section 2 without using adaptation; *i.e.*, the control system computes the control signal using Equation 4, and the translator schedules components according to Equation 7. In this case, the controller does not update its model of workload and the translator does not update its models of performance and power. This approach is limited by the fact that no one model can capture the interactions of all components across a range of applications. Comparing to this approach highlights the benefits of PTRADE's adaptive control and translation.

Dynamic Oracle: This oracle represents ideal behavior for a system that can act with knowledge of the future and without overhead. This oracle is calculated by running the application in every possible configuration and recording the heartbeat logs. We then select the best possible configuration for each heartbeat based on the recorded data. Thus, the dynamic oracle is a good proxy for the best possible performance management framework that could be built and tailored to a specific machine and application. Comparing to the dynamic oracle demonstrates the loss PTRADE's generalized approach incurs compared to an idealized solution built for a specific set of components[1]. We believe a less general system that is optimized for a particular machine will achieve power savings close to the dynamic oracle.

4.2 Metrics

[1]Our use of static and dynamic oracles is motivated by a similar approach used to evaluate the efficacy of managing per core DVFS [20]

We evaluate the *accuracy* of each approach (*i.e.*, its ability to meet the performance goals), by determining the *error*, calculated as $(g - \min(g, h))/\ g$, where g is the performance goal, and h is the achieved performance (see Equation 1). This metric penalizes systems for not achieving the performance goal, but provides no reward or penalty for exceeding it. Note that exceeding the performance goal will likely cause greater than optimal power consumption, and this will be reflected in the power efficiency.

To evaluate the *efficiency* of each approach (*i.e.*, its ability to minimize power while meeting the performance goal), we compute *normalized power*, which measures average power consumption when controlling a performance goal. Power is normalized to that achieved by the static oracle. For this metric lower values are better; normalized power consumption less than unity indicates a savings over the best possible non-adaptive strategy. We compute power by measuring total system power and subtracting out idle power.

4.3 Linux/x86 Results

4.3.1 Overhead

To evaluate PTRADE's overhead, we run all benchmark applications with PTRADE, but disable its ability to actually change component settings. Thus, the runtime executes and consumes resources, but cannot have any positive effect. We compare execution time and power consumption to the application running without PTRADE. For most applications, there is no measurable difference. The largest difference in performance is 2.2%, measured for the fluidanimate benchmark, while the power consumption is within the run to run variation for all benchmarks. We find this overhead to be acceptable and note that all measurements in this section include the overhead and impact of the runtime system.

Additionally, we account for overhead by measuring the rate at which the runtime makes decisions when not limited by waiting for feedback. We collect traces of application performance and power, feed them into the runtime,and measure the maximum achievable decisions per second. We measure a rate of 10.3 million decisions per second, confirming the above overhead measurement and demonstrating that the runtime is not a bottleneck in the system.

4.3.2 Results

We launch each benchmark on the target machine and evaluate each approach's accuracy and efficiency. Each benchmark requests a performance equal to half the maximum achievable on Machine 1. We use this target because it is hardest to hit while minimizing power consumption. Targets at the extremes can be achieved by only a few configurations, giving the management system fewer possibilities. Targets in the middle can be met by the most possible combinations, making it harder to find the one that minimizes power.

For each benchmark, we compute the performance error and normalized power. Figure 3 shows the results. In all cases, the x-axis shows the benchmarks (and the average for all benchmarks). In Figures 3(a) and 3(c), the y-axes show performance error on the two different machines. In Figures 3(b) and 3(d), the y-axes show the normalized power for each machine (lower is better).

The results in Figures 3(a) and 3(c) indicate that, on average, PTRADE is more accurate for performance targets than either uncoordinated adaptation or classical control. The average performance error on Machine 1 is 10.8% for uncoordinated control, 21.2% for classical control, and 3.6% for PTRADE. The average performance error on Machine 2 is 11.7% for uncoordinated control, 35.8% for classical control, and 2.2% for PTRADE. The lower performance error shows that PTRADE does a better job of meeting goals than uncoordinated or non-adaptive approaches.

Figures 3(b) and 3(d) show that PTRADE is more efficient than uncoordinated adaptation; *i.e.*provides lower power consumption for the given performance targets. In some cases, power consumption is slightly lower than the dynamic oracle. These cases correspond to times when the performance target was missed, so these small additional savings are coming at a cost of not meeting the performance goal, whereas the dynamic oracle is 100% accurate. On machine 1, the normalized power is 1.22 for uncoordinated adaptation, 1.18 for classical control, 0.80 for PTRADE and 0.77 for the dynamic oracle. Both uncoordinated and classical control are worse than the static oracle, while PTRADE is better. Uncoordinated control consumes 58% more power than the dynamic oracle while PTRADE exceeds optimal by 4.1%. On machine 2, the normalized power is 0.94 for uncoordinated adaptation, 0.96 for classical control, 0.84 for PTRADE and 0.79 for the dynamic oracle. In this case, all approaches improve on the static oracle, but PTRADE is closest to optimal. On machine 2, uncoordinated control consumes 19% more power than the dynamic oracle while PTRADE exceeds optimal by only 6.3%.

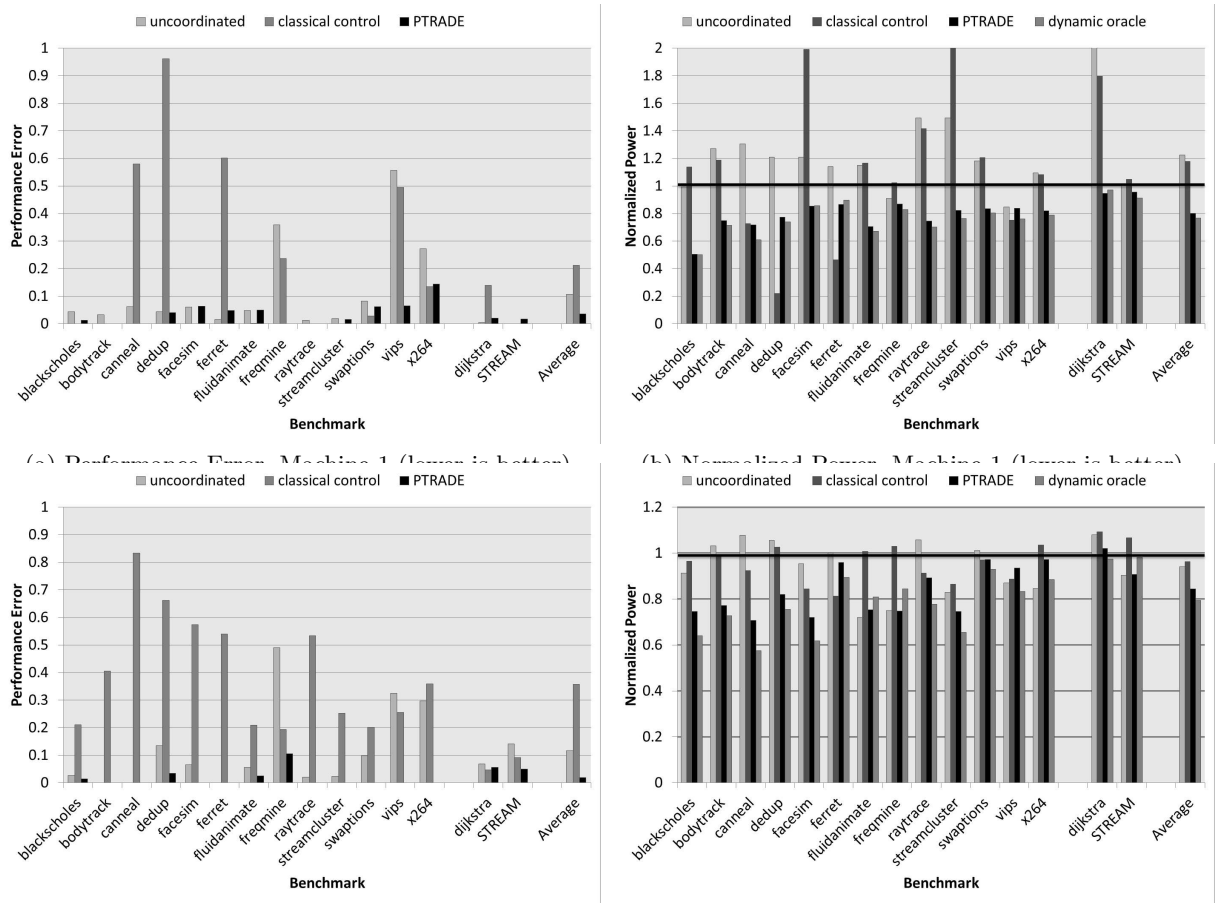
These results demonstrate the accuracy and efficiency of PTRADE, which achieves the performance goal with low error and close to optimal savings. In contrast, uncoordinated control is less accurate, less efficient, and on Machine 1 is actually worse than the static oracle on average. Furthermore, the results indicate the benefits of PTRADE's adaptive control and translation, as PTRADE is both more accurate and more efficient than classical control. These results illustrate that even when components are coordinated (which classical control does), accuracy and performance can suffer unless the management system adapts to the application (*e.g.*, using adaptive control) and system (*e.g.*, using adaptive translation) on which it is running.

This section demonstrates PTRADE's broad behavior across many applications. Additional results showing detailed behavior for several applications as a function of time are included in Appendix A. A detailed study of how PTRADE can benefit the x264 video encoder by adapting control to the needs of different video scenes is included in Appendix B.

4.4 Manycore Simulation Results

This section presents the results of running PTRADE on the simulated manycore described in Section 3. Deploying on the simulator allows us to test PTRADE on a new set of components and to gain insight into the scalability of the approach. In this section we omit results for uncoordinated adaptation due to limitations of simulation time. Again, the performance goal is set to half the maximum possible for each application as this stresses the management systems' ability to reduce power consumption.

Figure 4 shows the results. Performance error is shown in Figure 4(a) while normalized power is shown in Figure 4(b). The classic control system produces an extremely large error for ocean_non_contig, but not for any other applications. PTRADE produces a very small performance error for ocean



(a) Performance Error, Machine 1 (lower is better)

(b) Normalized Power, Machine 1 (lower is better)

(c) Performance Error, Machine 2 (lower is better)

(d) Normalized Power, Machine 2 (lower is better)

Figure 3: Controlling performance and minimizing power.

on this system and no error for the other benchmarks as it slightly exceeds goals for all applications. PTRADE is clearly more efficient than the classic control system. On average, PTRADE produces power consumptions only 3% higher than the dynamic oracle. In contrast, the classic control system's power consumption is quite bad on average because it performs so badly on ocean_non_contig. This benchmark actually slows down as core allocation increases beyond 8 cores; the most efficient configuration for this application is to use eight cores and maximum cache size and then idle for slack periods. PTRADE's adaptive translation learns to use this configuration through repeated updates of its internal models. The classical control formulation, however, cannot adjust its models on the fly and ends up allocating all resources on the chip and missing the performance goal while burning unnecessary power.

These results demonstrate that PTRADE can provide accuracy and efficiency for large scale multicores. They also demonstrate that PTRADE can work with systems that provide much finer grain power feedback than is available from the WattsUp? meter. Finally, these results further demonstrate the importance of incorporating adaptation into a generalized control system to manage the complexity of interactions between various components.

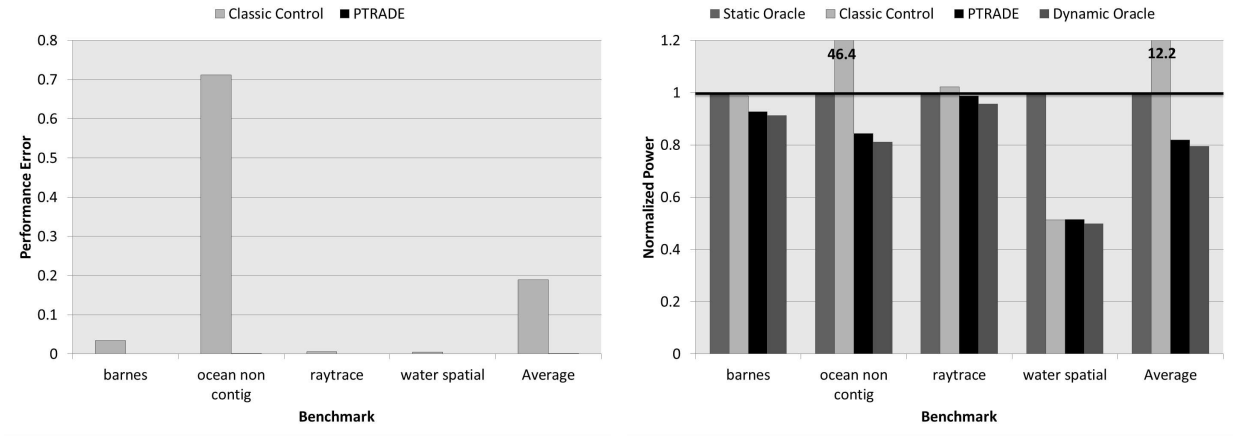
4.5 Summary

These results show PTRADE can achieve accurate and efficient performance management for a range of applications on different systems with different components. As shown in Table 3 the benchmarks include both high and low variance applications. The same PTRADE runtime is deployed on all platforms without redesign or re-implementation. The only thing that changes moving from one machine to the next is the specification of the different components on the different machines. Given an application and set of components, PTRADE's runtime manages the available components, automatically changing its own behavior based on feedback from the applications and machines. Overall, these results demonstrate that it is possible to build a generalized performance management system which approaches the performance of customized implementations. PTRADE's generalized approach allows applications to be ported to new platforms with new combinations of components governing performance versus power tradeoffs and still ensure that the goals are met while reducing corresponding power costs.

5. RELATED WORK

Multicore scalability is increasingly limited by power and thermal management [9, 37]. These limitations have inspired a number of techniques for managing power. These techniques can be distinguished by the guarantees they provide.

Some systems guarantee power consumption while maximizing performance subject to the power constraint. Cluster level solutions which guarantee power consumption include those proposed by Wang et al. [39] and Raghavendra et



(a) Performance Error (lower is better)

(b) Normalized Power (lower is better)

Figure 4: Controlling performance on simulated 128-core manycore.

al. [31]. These cluster-level solutions require some node-level power management scheme. Node-level systems for guaranteeing power consumption have been developed to manage DVFS for a processor [22], to manage per-core DVFS in a multicore [20], to manage processor idle-time [11], and to manage DRAM [7]. Other approaches provide power guarantees while increasing performance through coordinated management of multiple components, including processor and DRAM [5, 10], processors speed and core allocation [6, 33], and combining DVFS and thread scheduling [32, 41]. While important, these techniques do not provide any guarantees about performance, so they may not be usable with applications that require a certain level of performance to be maintained for correctness.

To address this need, other systems provide performance guarantees while minimizing power consumption. Examples exist at the cluster level [18, 38], and the node level. At the node level, proposed techniques provide performance and minimize power by managing DVFS in the processor [43], core throttling [45], assignment of cores to an application [26], caches [2], DRAM [46], and disks [24]. Researchers have shown that additional power savings can be achieved by coordinating multiple components within a node [29]. For example, Li et al. propose a method for managing memory and processor [25], while Dubach et al. demonstrate a method for coordinating a large collection of microarchitectural features [8]. Maggio et al. develop a classical (non-adaptive) control system for managing core allocation and clock speed [27]. Bitirgen et al. coordinate clockspeed, cache, and memory bandwidth [4]. Sharifi et al. develop an adaptive (but non-general) control scheme (called METE) for managing cores, caches and off-chip bandwidth [35]. While METE can adapt to different applications, it is not general with respect to actuators and requires the control system to be redesigned to work with new sets of components. All of these solutions provide performance guarantees by fixing a set of components and then tuning them to meet a performance target while minimizing power consumption; however, each of these approaches would require redesign and reimplementation to incorporate new components or to be ported to a new architecture with different components.

A recent approach, called PowerDial, uses a control system to manage tradeoffs between the performance of applications and the accuracy of their results [15]. Like PTRADE, PowerDial makes use of a generic control system, proposed by Maggio et al [26], to split management into separate control and translation phases. PowerDial, however, is limited by reliance on extensive offline profiling to create models of performance and accuracy tradeoffs. PowerDial has no capacity to update its internal models online; and thus, cannot adapt to either applications or system components which it has not previously profiled. PTRADE, in contrast, extends the idea of separate control and translation phases by making each adaptive and capable of modifying their behavior to continue to function even with previously unseen applications or system components.

PTRADE is another example of a node-level performance management framework based on the coordination of multiple components. PTRADE is distinguished by the fact that the components it manages are not known at design time. Instead, PTRADE is general and can be deployed to new platforms and new sets of components without redesign and reimplementation. Adding a new component for management simply requires a systems developer to register the component with PTRADE using the appropriate interface. In contrast, frameworks like those mentioned above, require the designers to re-architect their management system and reimplement it to take advantage of new components.

6. CONCLUSION

This paper proposes PTRADE, a novel performance management framework designed to be general with respect to the components and applications it manages. PTRADE is an example of a *self-aware computing* system [17, 16] that understands high-level user goals and automatically adjusts behavior to meet those goals optimally. PTRADE meets performance goals accurately and efficiently by breaking the management problem into two phases: a control phase, which determines how much to change performance at the current time, and a translation phase, which determines the most efficient configuration of components that will realize the desired performance. We evaluate PTRADE using a variety of benchmarks on both real and simulated machines. Across three different platforms with different characteristics, we find the PTRADE runtime meets goals with low error while providing close to optimal power consumption. We see PTRADE as an example of an emerging class of generalized management systems, which will help

navigate the complicated tradeoff spaces brought in to being by the necessity of managing both power and performance.

7. REFERENCES

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

[2] R. Balasubramonian et al. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[3] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, Oct 2008.

[4] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[5] J. Chen and L. K. John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS*, 2011.

[6] R. Cochran et al. Pack & cap: adaptive dvfs and thread packing under power caps. In *MICRO*, 2011.

[7] B. Diniz et al. Limiting the power consumption of main memory. In *ISCA*, 2007.

[8] C. Dubach et al. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO*, 2010.

[9] H. Esmaeilzadeh et al. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[10] W. Felter et al. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS*, 2005.

[11] A. Gandhi et al. Power capping via forced idleness. In *Workshop on Energy-Efficient Design*, 2009.

[12] J. L. Hellerstein et al. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[13] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[14] H. Hoffmann et al. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, 2010.

[15] H. Hoffmann et al. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

[16] H. Hoffmann et al. SEEC: A General and Extensible Framework for Self-Aware Comput ing. Technical Report MIT-CSAIL-TR-2011-046, MIT, November 2011.

[17] H. Hoffmann et al. Self-aware computing in the angstrom processor. In *DAC*, 2012.

[18] T. Horvath et al. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *Computers, IEEE Transactions on*, 56(4):444 –458, april 2007.

[19] M. E. A. Ibrahim et al. A precise high-level power consumption model for embedded systems software. *EURASIP J. Embedded Syst.*, 2011:1:1–1:14, Jan. 2011.

[20] C. Isci et al. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO*, 2006.

[21] A. B. Kahng et al. Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration. In *DATE*, 2009.

[22] C. Lefurgy et al. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.

[23] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[24] X. Li et al. Performance directed energy management for main memory and disks. *Trans. Storage*, 1(3):346–380, Aug. 2005.

[25] X. Li et al. Cross-component energy management: Joint adaptation of processor and memory. *ACM Trans. Archit. Code Optim.*, 4(3), Sept. 2007.

[26] M. Maggio et al. Controlling software applications via resource allocation within the heartbeats framework. In *CDC*, 2010.

[27] M. Maggio et al. Power optimization in embedded systems via feedback control of resource allocation. *IEEE Transactions on Control Systems Technology*, 21(1):239–246, 2013.

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

[29] D. Meisner et al. Power management of online data-intensive services. *ISCA*, 2011.

[30] J. E. Miller et al. Graphite: A distributed parallel simulator for multicores. In *HPCA*, 2010.

[31] R. Raghavendra et al. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS*, 2008.

[32] K. K. Rangan et al. Thread motion: fine-grained power management for multi-core systems. In *ISCA*, 2009.

[33] S. Reda et al. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 32(5):64–75, 2012.

[34] E. Rotem et al. Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge. In *Hot Chips*, Aug. 2011.

[35] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *SIGMETRICS*, 2011.

[36] S. Thoziyoor et al. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*, 2008.

[37] G. Venkatesh et al. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, 2010.

[38] A. Verma et al. Server workload analysis for power minimization using consolidation. In *USENIX Annual technical conference*, 2009.

[39] X. Wang et al. MIMO power control for high-density servers in an enclosure. *IEEE Transactions on Parallel and Distributed Systems*, 21(10):1412–1426, 2010.

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

[41] J. A. Winter et al. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *PACT*, 2010.

[42] S. C. Woo et al. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995.

[43] Q. Wu et al. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS*, 2004.

[44] R. Zhang et al. Controlware: A middleware architecture for feedback control of software performance. In *ICDCS*, 2002.

[45] X. Zhang et al. A flexible framework for throttling-enabled multicore management (temm). In *ICPP*, 2012.

[46] H. Zheng et al. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *MICRO*, 2008.

APPENDIX

A. DETAILED RESULTS FOR X86

This section presents some supplemental results showing the behavior of three applications under PTRADE's control. This data demonstrates: 1) how PTRADE handles the same benchmarks differently on two different machines, 2) how PTRADE handles a low-variance benchmark (facesim), 3) how PTRADE handles a high-variance benchmark (swaptions), and 4) how adaptive control and translation allows PTRADE to adopt new strategies dynamically and not over-allocate resources for STREAM, which is the most challenging benchmark for the system to handle. Controlling STREAM requires PTRADE to adapt to non-linearities in the interaction between the system components.

Figures 5–7 show detailed behavior for facesim, swaptions and STREAM managed by both PTRADE and by the static oracle. Each figure consists of six sub-figures (a-f) illustrating behavior in time; (a-c) show the data for Machine 1, while (d-f) show the data for Machine 2. Sub-figures (a) and (d) show performance (normalized to the goal). Sub-figures (b) and (e) show power consumption (subtracting out idle power). Sub-figures (c) and (f) show system component usage as the proportion of the component in use (*e.g.*, using four of eight cores would be depicted as 0.5).

Figure 5 shows detailed results for the facesim benchmark. For both machines, PTRADE's performance is very close to that of the static oracle which is close to the desired performance. This is not surprising since facesim is a very regular benchmark; however, PTRADE saves power compared to the static oracle in both cases. On machine 1, PTRADE's average performance is 98% of the target performance while its average power consumption is 27% less than the static oracle. On this machine, PTRADE's translator periodically allocates all resources and then idles the machine for large portions of time, as illustrated in Figure 5(c). On machine 2, PTRADE consumes 7% less power than the static oracle by allocating the minimal amount of resource required to meet goals and idling for short amounts of time when there is slack in the schedule, as shown in Figure 5(f). For this benchmark, PTRADE holds performance steady at the desired level. These examples show that there is some oscillation in the power, however, but we note that power consumption is not under control. Instead, PTRADE is minimizing average power consumption, and has determined that the best way to do so for the given performance target is to alternate component configurations.

Figure 6 shows how PTRADE handles swaptions on the two different machines. First, we see that PTRADE holds average performance much closer to the desired level than the static oracle. On machine 1, PTRADE achieves 94% of the target performance while consuming 10% less average power. On this machine PTRADE allocates the minimal amount of resources required to meet goals and idles for very short amounts of time when there is slack in the schedule, as shown in Figure 6(f). On machine 2, PTRADE achieves 99% of the target performance while consuming 8.6% less average power than the static oracle. For this machine, PTRADE adopts a different approach, allocating all resources and then idling the machine for large portions of time, as illustrated in Figure 6(c). We note that the performance for this benchmark oscillates due to the high variance in the performance signal Table 3. The power consumption also oscillates, but as in the case of facesim, this is desired as PTRADE has determined that oscillation lowers the average power consumption.

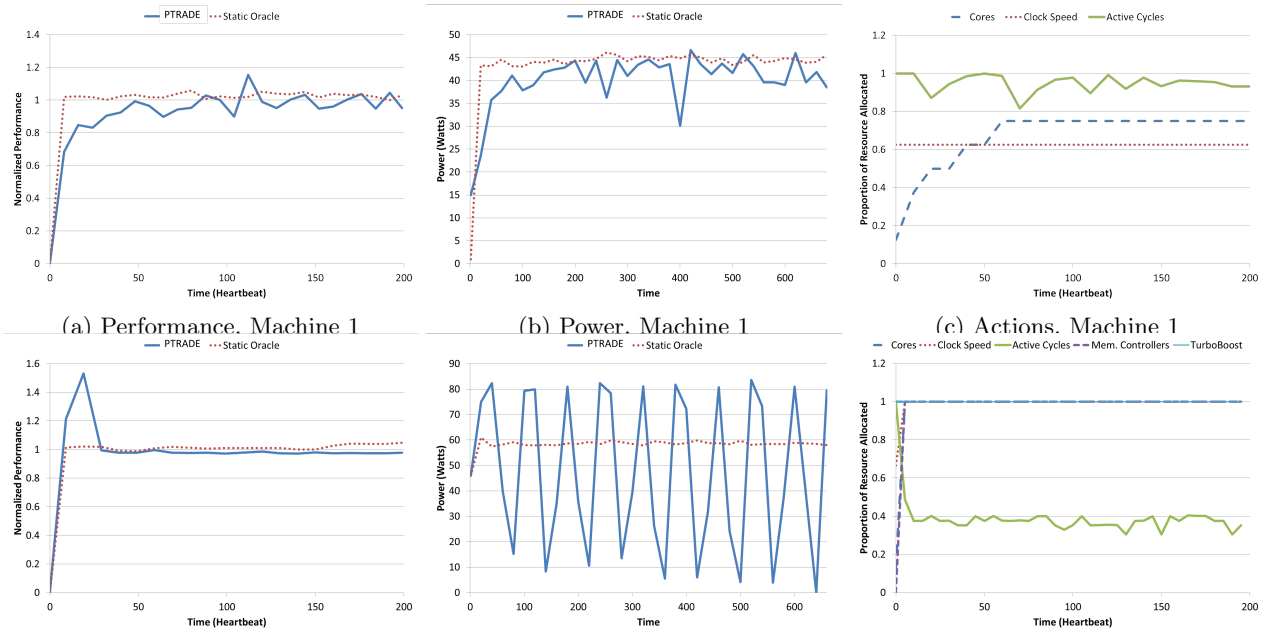
Figure 7 shows how PTRADE handles STREAM on the two different machines. STREAM forces PTRADE to adapt its strategy in order not to over-provision resources to this memory bound application. PTRADE holds average performance close to the desired level; achieving the target performance on Machine 1 and 97% of the target on machine 2. On both machines, PTRADE reduces power consumption by almost 10% compared to the static oracle. For this benchmark, the most interesting thing is how these results are achieved. In both cases, PTRADE explores several different strategies using adaptive translation. After some initial exploration, the Kalman filters converge and the system settles to a desired operating point. On Machine 1, PTRADE allocates between 1 and 4 cores while keeping clock speed at the lowest setting. On Machine 2, PTRADE allocates all the memory controllers, and all the cores, but turns TurboBoost off and sets the clock to its slowest speed. These resource allocations are efficient for the memory bound application, but are achieved without any prior knowledge of the properties of the application itself. For this application, it takes a small amount of time to achieve the desired performance, and during this time, PTRADE is exploring a number of different component configurations. Once the Kalman filters have converged, performance converges as well.

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

B. MORE VIDEO ENCODING RESULTS

This experiment highlights PTRADE's ability to adapt to different application inputs, each with differing compute demands. Specifically, we show how PTRADE manages the x264 video encoder while adapting to different video inputs. We use fifteen 1080p videos from xiph.org and the PARSEC native input. We alter x264's command line parameters to maintain an average performance of thirty frames per second on the most difficult video using all compute resources available on Machine 1. x264 requests a heart rate of 30 beat/s corresponding to a desired encoding rate of 30 frame/s. Each video is encoded separately. We measure the performance per Watt for each input when controlled by a system that allocates the same resources to all videos to ensure that the performance goal is met in all cases (*i.e.*, it allocates for worst case execution time, or *wcet*), and PTRADE. Figure 8 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 for Machine 1, PTRADE outperforms the static oracle by 1.1× and the wcet allocator by 1.28×, while achieving 97% of the desired performance. On average for Machine 2, PTRADE outperforms the static oracle by 1.1× and the wcet allocator by 1.44×, while achieving 99% of the desired performance. PTRADE is meeting the perfor-



(a) Performance, Machine 1 (b) Power, Machine 1 (c) Actions, Machine 1

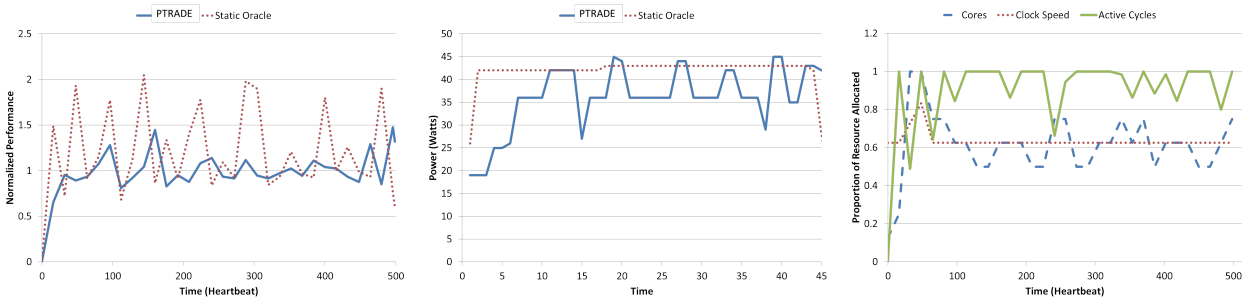
(d) Performance, Machine 2 (e) Power, Machine 2 (f) Actions, Machine 2

Figure 5: Details of PTRADE controlling facesim on two different machines.

mance goals, so these gains in performance per Watt reflect considerable potential power savings compared to allocating for the worst case input. Additional results showing detailed behavior for several inputs as a function of time are included in Appendix B.

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

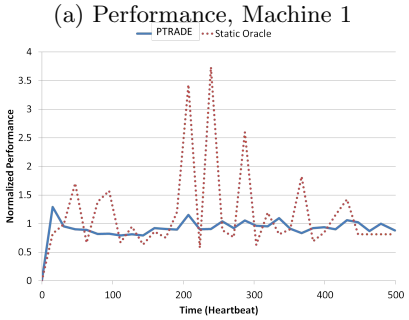
Figure 9 presents the results when PTRADE controls x264 encoding the concatenated video. The x-axes show time, measured in heartbeats, and the y-axes show performance (Figure 9(a)) measured in frames per second and power (Figure 9(b)). Results are shown for both PTRADE and the worst-case-execution-time (wcet) allocator. As shown in the figures, the first phase causes PTRADE to work hard and there are some sections for which neither PTRADE nor wcet can maintain the target goal. In the second phase, PTRADE quickly adjusts to the ease in difficulty and maintains the target performance while wcet has reserved over twice as many resources as needed, consuming unnecessary power. In the final phase, PTRADE is able to closely maintain the target performance and save power despite the noise evident in this portion of the video.



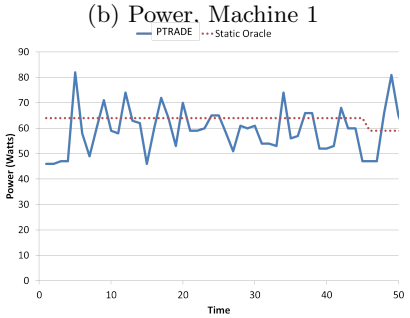
(a) Performance, Machine 1

(b) Power, Machine 1

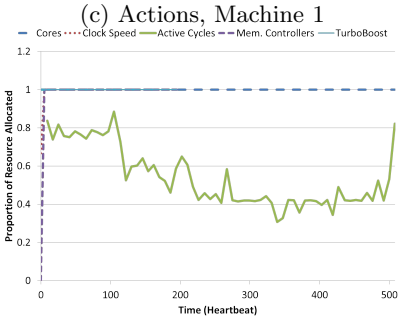
(c) Actions, Machine 1



(d) Performance, Machine 2

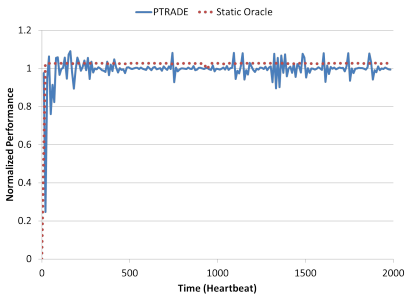


(e) Power, Machine 2

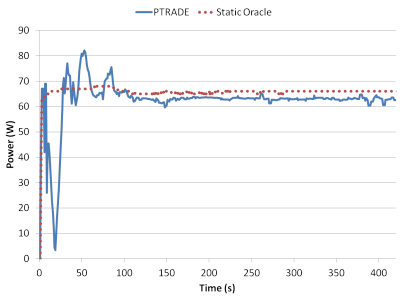


(f) Actions, Machine 2

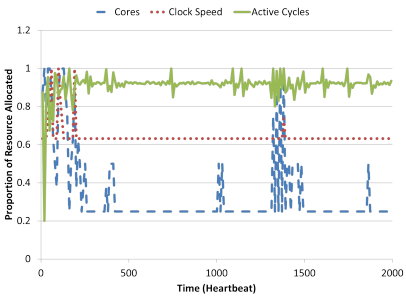
Figure 6: Details of PTRADE controlling swaptions on two different machines.



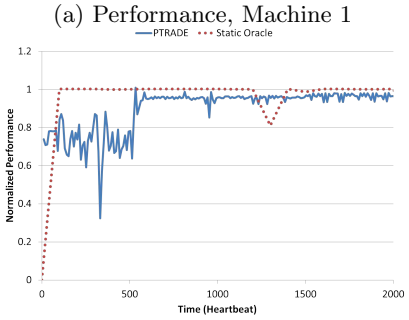
(a) Performance, Machine 1



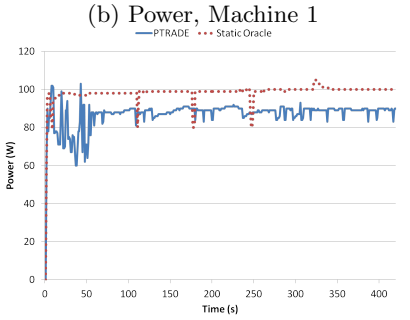
(b) Power, Machine 1



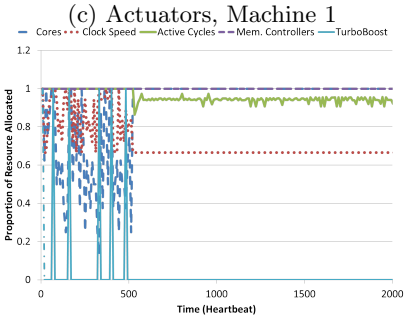
(c) Actuators, Machine 1



(d) Performance, Machine 2

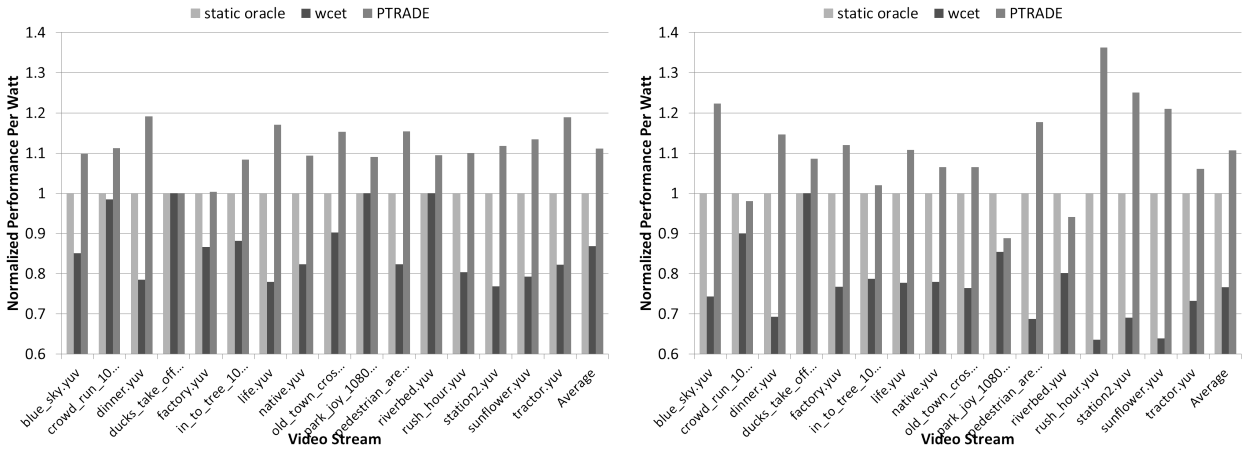


(e) Power, Machine 2



(f) Actuators, Machine 2

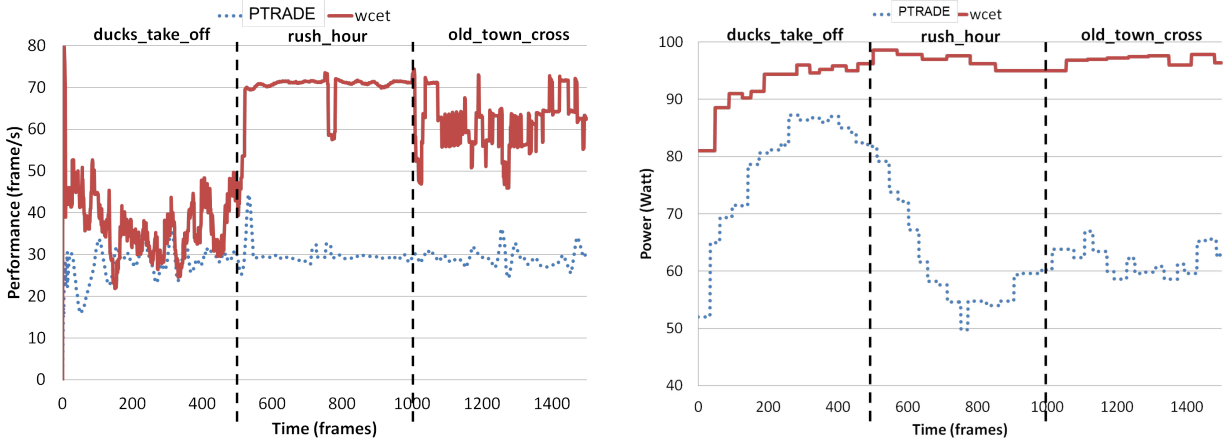
Figure 7: Details of PTRADE controlling STREAM on two different machines.



(a) Machine 1

(b) Machine 2

Figure 8: Performance/Watt for x264 with different inputs (higher is better).



(a) Performance

(b) Power

Figure 9: PTRADE controlling x264.