Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs

Jeffrey O. Kephart, Hoi Chan, Rajarshi Das, David W. Levine,
Gerald Tesauro, Freeman Rawson, and Charles Lefurgy
IBM Research
{kephart, hychan, rajarshi, dwl, gtesauro, frawson, lefurgy}@us.ibm.com

Abstract

Getting multiple autonomic managers to work together towards a common goal is a significant architectural and algorithmic challenge, as noted in the ICAC 2006 panel discussion regarding "Can we build effective multi-vendor autonomic systems?" We address this challenge in a real small-scale system that processes web transactions. An administrator uses a utility function to define a set of power and performance objectives. Rather than creating a central controller to manage performance and power simultaneously, we use two existing IBM products, one that manages performance and one that manages power by controlling clock frequency. We demonstrate that, with good architectural and algorithmic choices established through trial and error, the two managers can indeed work together to act in accordance with a flexible set of power-performance objectives and tradeoffs, resulting in power savings of approximately 10%. Key elements of our approach include a) a feedback controller that establishes a power cap (a limit on consumed power) by manipulating clock frequency and b) reinforcement learning, which adaptively learns models of the dependence of performance and power consumption on workload intensity and the powercap.

1 Introduction

Energy consumption is a major and growing concern for customers, data centers, server vendors, government regulators and non-governmental organizations concerned with energy and environmental matters. To cite a prominent example, the US Congress recently mandated a study of the power efficiency of servers, including a feasibility study of an Energy Star standard for servers and data centers [16]. Growing interest in power management is also apparent in the formation of the Green Grid, a consortium of systems and other vendors dedicated to improving data center power efficiency [4]. Recent trade press articles also make it clear that computer purchasers and data center operators are eager to reduce power consumption and the heat densities being experienced with current systems.

In response to these concerns, researchers are tackling intelligent power control of processors, memory chips and whole systems, using technologies such as processor throttling, frequency and voltage manipulation, low-power DRAM states, feedback control using measured power values, and packing and virtualization to reduce the number of machines that need to be powered on to run a workload.

Power management mechanisms can affect other aspects of system behavior such as performance and availability. Moreover, power management mechanisms can affect performance in ways that are highly dependent on workload characteristics [7]. For example, for some workloads, performance suffers linearly in proportion to CPU frequency reductions, while other workloads are largely unaffected. For these reasons, power cannot be managed independently: uncoupled power and performance managers are practically guaranteed to work at cross purposes.

Many previous research efforts have attempted to address this problem through centralized approaches that manage power and performance jointly. Although perhaps feasible in prototype, centralized approaches do not recognize the reality of today's IT environments, which typically contain multiple management products specialized to different disciplines including performance and power as well as other characteristics such as availability [10]. In contrast, we take as given the specialization of systems management products into separate management disciplines, and seek to augment these existing products with new algorithms and protocols that allow them to work together effectively.

Our approach is based on the use of multi-criteria *utility functions*, and is summarized briefly as follows:

- Establish a joint utility function for power and performance.
- Convey the utility function or some aspect of it to the separate power and performance managers.
- Develop a utility-optimizing power management policy that maps current system state to a management

action such as setting a power cap on a particular system, or turning machines on and off.
- During runtime, execute the power management policy within the power manager.
- Feed relevant information (e.g. processor frequency) back to the performance manager.

The particular joint utility function $U_{pp}(\mathbf{RT}, \text{Pwr})$ of the vector of response times $\mathbf{RT}$ and power consumed Pwr that we use in this paper subtracts a linear power cost from a performance-based utility $U(\mathbf{RT})$:

$$U_{pp}(\mathbf{RT}, \text{Pwr}) = U(\mathbf{RT}) - \epsilon * \text{Pwr} \quad (1)$$

where ϵ is a tunable coefficient expressing the relative value of power and performance objectives. However, our approach also admits more general functional forms of U_{pp}. For example, one could consider a "performance value per watt" objective function $U_{pp} = U(\mathbf{RT})/\text{Pwr}$, or a simple performance-based utility $U_{pp} = U(\mathbf{RT})$ coupled with a constraint on total power.

In essence, our approach is agent-based: the two managers can be regarded as two interacting agents in a multi-agent system. While multi-agent environments frequently enable explicit *negotiation* between agents, the initial approach described in this paper is negotiation-free. The power manager receives state information from the performance manager. Then, using its power measurements, it applies a power policy attempting to maximize the joint utility U_{pp}, implicitly accounting for whatever power-unaware decisions are made by the performance manager. The performance manager is unmodified, with a single exception: it receives information about the dynamically varying CPU frequencies from the power manager. This is clearly a special case of the more general situation where negotiation is required; ultimately we wish to determine the amount and type of negotiation that is needed.

Although our ultimate goal is to optimize multiple aspects of data center behavior such as performance, power, and availability, the current study considers only how to manage performance and power for a small cluster of blade systems in a single chassis. A blade-based system shares power and cooling resources among a number of separate blades but offers some centralized control over both the individual blade systems and the shared resources. The sharing of power and cooling requires that the blades be managed as a group rather than as stand-alone systems.

After discussing related work in the following section, Section 3 of the paper presents an overview of our system architecture and implementation. Section 4 describes our experimental setup (4.1), presents two approaches to deriving power policies, one hand-designed using offline measurements, and one based on machine learning (4.2), and presents experimental results (4.3). Section 5 discusses conclusions and next steps in our ongoing research.

2. Related work

Many studies have proposed using processor performance states to meet an SLA performance requirement while reducing energy use. Wang et al. simulate a technique that uses optimal control theory to derive a schedule of processor frequency control [17]. It depends only on the global queue length being broadcast to each server and can therefore scale easily as the number of servers increases. Chen et al. simulate a cluster using both predictive queuing models and feedback controllers to derive frequency adjustments for web serving clusters over control intervals lasting several minutes [1]. Horvath et al. control end-to-end delay across a three-tier web service by adjusting the processing speed of the servers in each tier[5]. This technique uses a 200 ms control period and is more reactive to surges in workload and has been shown to save 30% energy use in their implementation on real servers.

In our work, both the SLA and peak power consumption are constraints. We cannot directly set the processor frequencies as in other studies because the frequencies are under the control of a server-level peak power manager, which is required for ensuring power reliability and safety at the enclosure and cluster level. The power manager in the firmware of each server adjusts processor speed several times a second, which would be impractical to do by an external controller in a large cluster. Instead, we set the power budgets that each server may use, which indirectly raises and lowers the performance level of the server. This methodology allows us to integrate our technique with power management products that will soon be available in the marketplace [2].

Previously, we have shown that a power manager embedded in the firmware of a server can precisely control the server's power consumption using feedback control [18]. This technique consists of a proportional controller coupled with a first-order sigma-delta modulator adjusting the clock throttling setting of the processor to adhere to a given power supply constraint. The work reported in this paper uses the same feedback controller and power measurement circuit, except that we have modified the firmware to improve the precision of the power measurement tenfold.

Femal et al. [3] allocate power budgets among a cluster of homogeneous servers using linear programming at each control interval. They use forecasts of each node's contribution to work done by the cluster during each control interval to set the budgets. Blade enclosure-level power optimization has been studied by Ranganathan et al. [11] Their enclosure controller takes a SLA performance requirement and power constraint as input and directly manipulates the processor frequencies to meet the constraints. Their results are based on simulation of power management algorithms which require modeling the dependence of server power consumption to the OS-reported processor utilization. Our

framework is different from these approaches in that we assume the nodes are heterogeneous, and may therefore display different power and performance characteristics. We employ a novel use of machine learning to identify these unique characteristics for each server, reducing manual parameter tuning for large clusters.

3 System Architecture and Implementation

Figure 1 provides a high-level overview of our experimental testbed. In brief, a Workload Generator produces a single workload with dynamically varying intensity, which is routed by a Workload Distributor to set of blades contained in a single chassis. The Workload Distributor's routing policy is set within WebSphere Extended Deployment (WXD)[6], a managed multi-node webserver environment comprising extensive data collection and performance management functionality, described in more detail below. WXD also manages control parameters on individual blades, such as the maximum workload concurrency. Power on each blade is managed dynamically based on current power and performance data, using a control policy set by our Power Manager module as detailed below. Our architecture also allows manager-to-manager interactions between the Power Manager and WXD, as discussed in section 3.3.

Our data collection approach integrates several different data sources to provide a single, consistent view. Several dozen elements of performance data, such as mean response time, queue length and number of CPU cycles per transaction, are collected by the WXD data server, a daemon running on WXD's deployment manager. We also run local daemons on each blade to provide CPU utilization per blade, and current CPU frequency, taking into account both the true frequency and any effects due to the current level of processor throttling. The CPU on each blade also contains firmware that collects current power measurements [18]; these are polled using IPMI commands sent from the BladeCenter management module. A data collector receives the several streams of data described above and provides a synchronized report to the policy evaluator at a configurable logging interval τ_l (typically set to 5 seconds). Data generated on much faster time scales than τ_l are time-averaged over the interval, otherwise the most recent values are reported.

3.1 Power Manager details

The Power Manager module uses a collection of TCL and C programs to compute the desired power management actions (i.e. per-blade power budgets, or "powercaps"), along with extensive standard Unix/ssh tooling to drive the OS and middleware components. The powercaps are relayed by IPMI commands to blade firmware containing a feedback controller to implement the desired cap. As described in [18], the controller combines a proportional controller with a first-order sigma-delta modulator adjusting the frequency setting of the processor to adhere to a given powercap. The controller is designed to work within the time constraint of the power supply overload condition, making adjustments every 64 msec. When the server uses less power than the powercap, it runs at full speed. When it seeks to use more power than the powercap, it runs at a reduced frequency to precisely meet the constraint. Our implementation increases the precision of the firmware power measurement by a factor of 10, from 1 watt to 0.1 watt.

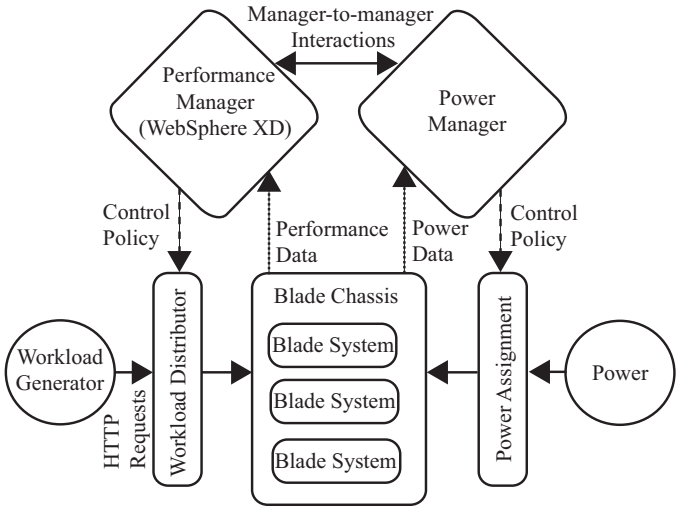

Figure 1: Overview of testbed environment.

The performance overhead introduced by data collection and power management has not been quantified, but it is expected to be negligible because in neither instance does the code execute on the managed blades.

3.2 WXD performance management

WebSphere Extended Deployment (WXD) is an advanced middleware application server environment. The features of WXD most relevant to our experiments are as follows. WXD provides a facility to set performance targets for each installed web application. It provides a load balancing reverse proxy (the "On Demand Router") that directs requests to individual servers such that the performance targets are met. WXD also has a resource controller that monitors response times and other performance metrics, and periodically recomputes the resource allocation parameter values (concurrency limits) used by the proxy. This is performed by first estimating $\alpha_{s,a}$, the average CPU resource (cycles/second) required per request by application a on server s, by linear regression on observations over the last 20 t time periods (= 5 minutes):

$$\rho_s^t \Omega_s^t = \sum_a \alpha_{s,a} N_{s,a}^t \lambda_{s,a}^t \qquad (2)$$

where, for each server s, ρ_s is the CPU utilization, Ω_s is its CPU capacity or frequency (in cycles/second), $N_{s,a}$ is the number of concurrent requests executed for application a, and $\lambda_{s,a}$ is the throughput per concurrent request. Given the $\alpha_{s,a}$ estimates, WXD then solves for new concurrency limits $N^*_{s,a}$, with the aim of achieving 90% utilization on each server, based on current empirical data at time t:

$$\rho^*_s \Omega^t_s = \sum_a \alpha_{s,a} N^*_{s,a} \lambda^t_{s,a}. \quad (3)$$

3.3 Autonomic Manager Interactions

We expect that integration of disparate autonomic managers will be an important research and practical challenge, since such managers may be made by multiple vendors, may manage to different criteria and may operate on vastly different time scales. For example, in our work we inadvertently discovered that WXD needed to be given current CPU frequencies when collaborating with our Power Manager module. This finding, while perhaps unsurprising in hindsight, points out what we expect will be a general need to identify a minimal data set to communicate between managers in order for them to work together effectively.

To illustrate the above point, Figure 2(a) shows behavior resulting when our power and performance managers operate without giving feedback to each other. The experiment (the setup for which is described in the next section) is extremely simple: we hold the workload intensity (number of clients) fixed, and we use a single blade with fixed powercap $p_\kappa = 100W$, so we'd expect WXD to have no trouble in finding a steady-state performance management policy. Yet we see numerous state variables such as CPU utilization and frequency showing persistent large amplitude oscillations.

Careful study revealed that, in the absence of feedback, WXD assumes that the server is running at its default CPU speed $\Omega = 3000$MHz whereas due to power management, the speed was generally less than this, and variable. This eventually causes WXD's α estimates to become too large, whereupon Equation 3 dictates that WXD should reduce the concurrency level N^*, effectively throttling the workload. The throttling eventually leads to reduced CPU utilization and power consumption, which causes the Power Manager to increase CPU speed to keep power close to the desired cap. Ultimately, this leads to reduced α values, leading WXD to increase N^*. The entire cycle repeats several times during the experiment.

The obvious solution is to provide the correct CPU frequency as measured by the power manager to WXD. This quells the self-induced oscillations, as seen in Figure 2(b). This minimal level of information feedback is sufficient to stabilize the performance and power metrics over a long period of time, signifying strong and enduring coordination between the two autonomic managers.

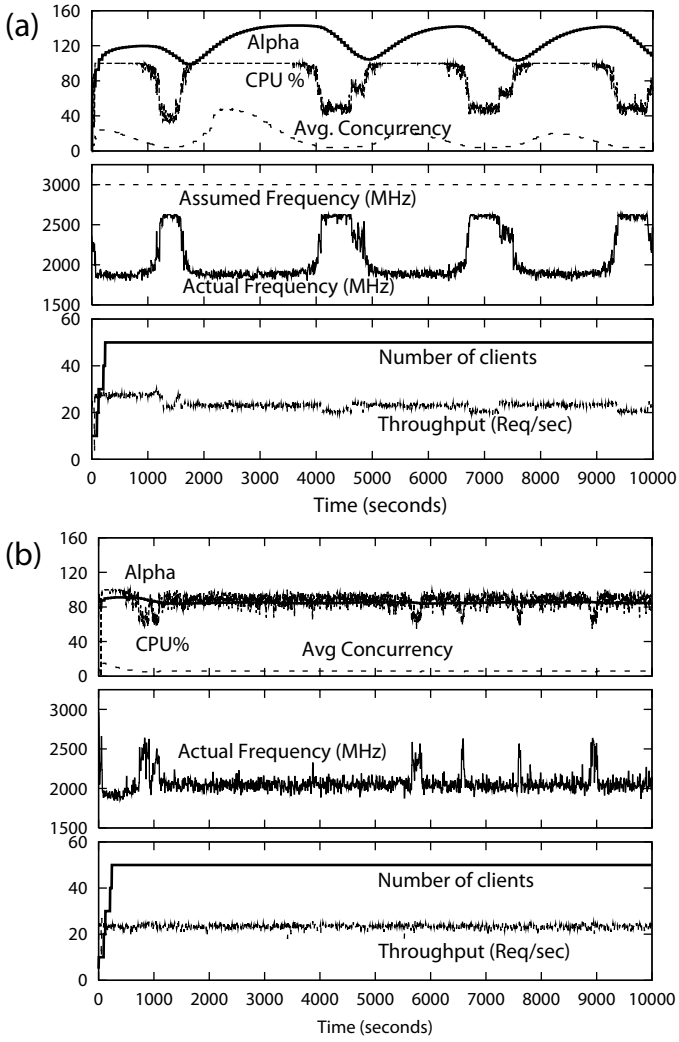

Figure 2: Effect of CPU frequency feedback on system behavior. (a) WXD receives no feedback (b) WXD receives feedback.

4 Experiments

We ran a series of experiments to establish whether we could effectively control the powercap dynamically to manage to a specified tradeoff between performance and power, as expressed in a utility function. After describing the experimental setup in section 4.1, we discuss in detail two methods for deriving dynamic powercap control policies in section 4.2, following this with a description of experimental results in section 4.3.

4.1 Experimental setup

Our experiment setup contains a rack of 7 somewhat heterogeneous IBM eServer xSeries HS20 blades running SUSE Linux Enterprise 9 Service Pack 3. A single installation of WXD is configured with the deployment manager residing on one blade and an ODR (On Demand Router)

residing on another; the remaining 5 blades are available as server nodes. While we have conducted experiments using multiple blades, here we focus on an in-depth analysis of experiments involving a single blade, Goldensbridge (GB). GB has an Intel Xeon 3.00 GHz processor with 1MB of level 2 cache and 2GB of DRAM. It is configured with hyperthreading enabled, so that two virtual processors are visible to the OS. Like the other blades, GB includes service processor firmware that supports peak power capping and power measurement, as described in section 3. For the test application, GB could serve up to 30-40 clients and still meet the 1000 millisecond average response time threshold. The power manager runs on a separate Windows-based system located outside the BladeCenter enclosure.

The workload is generated by the Wide-Spectrum Stress Tool (WSST) included in the IBM Web Services Toolkit. WSST contains a simple web-based application, which we configured to have a single service class. The workload intensity is controlled by varying the number of clients n_c sending requests to the ODR. For each client, we use a closed-loop workload generator [8] with a think time drawn from an exponential distribution with mean 125 msec. We varied n_c using a statistical model of web traffic derived from observations of a highly accessed Olympics web site [13].

WXD permits an administrator to establish a simple performance utility function expressing the value of attaining a given average response time. One way in which the utility function can be elicited is via a simple template: the administrator inputs a response time target RT_0 and selects one of seven importance levels ranging from very low to very high. These two parameters are mapped to a simple utility function $U(\mathrm{RT})$. In our experiments, we employed the type of power-performance utility function given by Eq. 1, in which the performance part of the utility function is 1.0 when the response time is less than the response time threshold and drops linearly with response time when it exceeds the threshold. Formally, the combined power-performance utility function can be expressed as:

$$U_{pp}(\mathrm{RT}, \mathrm{Pwr}) = U(\mathrm{RT}) - \epsilon * \mathrm{Pwr} = \\ 1.0 - (r-1)\Theta(r-1) - \epsilon * \mathrm{Pwr} \quad (4)$$

where Θ represents the step function and $r = \mathrm{RT}/\mathrm{RT}_0$ denotes the ratio of the current mean response time to the response time threshold. (Note that RT is now a scalar rather than a vector because there is a single service class.)

In all of our experiments, the response target was RT_0 = 1000 msec. Adjusting ϵ allows one to explore a range of power-performance tradeoffs. For low values of ϵ ($\epsilon \leq 0.01$), the tradeoff is strongly biased in favor of performance, and power savings are only considered if the performance is less than 1000 msec. If the administrator seeks greater power conservation and is willing to tolerate response times somewhat higher than the nominal response time target of 1000 msec, the value of ϵ can be increased to 0.05 or greater.

4.2 Deriving Powercap Policies

In order to understand how much power could be saved by dynamically manipulating the powercap, we used three different types of powercap policy:

- **Unmanaged**. Power management is turned off, and therefore the processor always runs at the highest clock frequency.
- **Handcrafted**. The powercap policy is a function of the number of clients n_c, $p_\kappa(n_c)$. It is established by conducting exhaustive experiments to map the dependency of power consumption and response time upon the powercap p_κ and n_c.
- **Machine-learned**. The powercap policy is a function of measured state variables that could include n_c, and is derived via reinforcement-learning methods.

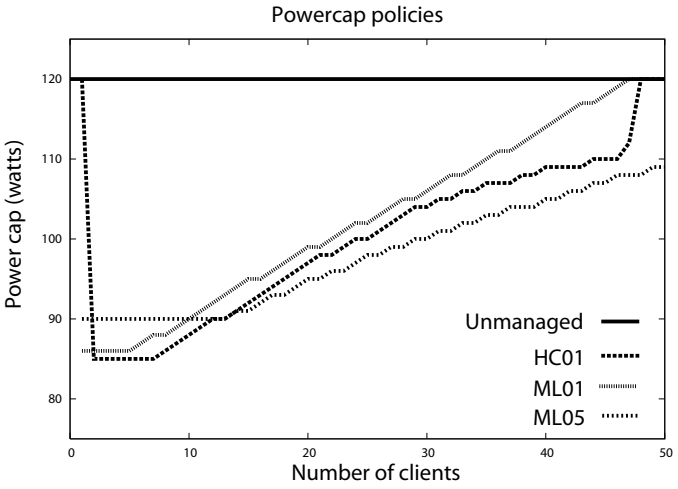

Figure 3: Powercap policies: Unmanaged, Hand-crafted, and two machine-learning derived policies for different values of the power cost parameter (0.01 and 0.05).

Instances of these types of powercap policies are illustrated in Figure 3. For the handcrafted policy, the power-performance tradeoff parameter ϵ was set to 0.01. Two different policies derived via machine learning are displayed: one for $\epsilon = 0.01$ and the other for $\epsilon = 0.05$. The policy for the larger value of ϵ uniformly sets a lower powercap for a given value of n_c because the emphasis is being shifted from high performance towards power conservation.

Now we describe the derivation of the handcrafted and machine-learned policies in greater detail.

4.2.1 Handcrafted policy from offline measurements

Suppose that we have an arbitrary utility function $U_{pp}(\mathbf{RT}, \mathrm{Pwr})$, where $\mathbf{RT}$ represents a vector of response

times (one for each service class) and Pwr represents the total power allocated to the set of blades used by the application. Then, given models of how response time and consumed power depend upon the workload intensity (expressed in terms of the number of clients n_c) and the powercap p_κ, we follow [14], substituting them into our utility function $U_{pp}(\mathbf{RT}, \text{Pwr})$ to obtain an equivalent utility function U' that is purely a function of p_κ and n_c:

$$U'(p_\kappa, n_c) = U_{pp}(\mathbf{RT}(p_\kappa, n_c), \text{Pwr}(p_\kappa, n_c)) \quad (5)$$

Then, from $U'(p_\kappa, n_c)$ we compute the optimal powercap for each possible value of the workload intensity n_c to generate a powercap policy $p^*_\kappa(n_c)$:

$$p^*_\kappa(n_c) = \arg\max_{p_\kappa} U'(p_\kappa, n_c) \quad (6)$$

Since we use a single service class and a single blade, the vector $\mathbf{RT}$ is reduced to a scalar RT and Pwr represents the power consumed by that blade. To obtain the scalar models $\text{RT}(p_\kappa, n_c)$ and $\text{Pwr}(p_\kappa, n_c)$, we measured power consumption on GB at extremely low ($n_c = 1$) and high ($n_c = 50$) loads, finding that in all cases the power consumption ranged between 75 and 120 watts. Given this range, we established a grid of sample points, with p_κ running from 75 watts to 120 watts in increments of 5 watts, and the number of clients running from 0 to 50 in increments of 5 [1]. For each of the 10 possible settings of p_κ, we held n_c fixed at 50 for 45 minutes to permit WXD to adapt to the workload, and then decremented n_c by 5 every 5 minutes. Then, for each of the resulting $10 \times 11 = 110$ samples, we recorded the average response time and power consumption. The results are summarized in Figure 4.

Figure 4a shows that the response time is a strongly nonlinear function of the powercap and the number of clients. For the smallest values of n_c, the response time is very low and well below the response time target regardless of the power cap p_κ. For moderate to large values of n_c, the response time can exceed 10000 msec when p_κ is 75 watts. Raising p_κ to 80 watts has no discernible effect. On closer examination, the reason is clear: at $p_\kappa = 80$ watts the processor stays in its lowest frequency state (375 MHz) the vast majority of the time, so setting the powercap even lower cannot yield significant additional savings. However, as p_κ is increased above 80 watts, the response time decreases rapidly until p_κ has increased to 110 watts, beyond which there is no further reduction because at this point the chip is already in its highest frequency state.

Figure 4b shows that the consumed power is also a nonlinear function of the powercap and the number of clients, although there is linear dependence on p_κ in a broad middle range of p_κ and n_c. For any given value of n_c, the power consumed levels out below roughly $p_\kappa = 85$ watts and above $p_\kappa = 110$ watts. For any given value of p_κ, the consumed power rises rapidly as n_c is increased from zero to some threshold n_c, beyond which the power consumption does not increase further; the threshold value of n_c increases nonlinearly with p_κ.

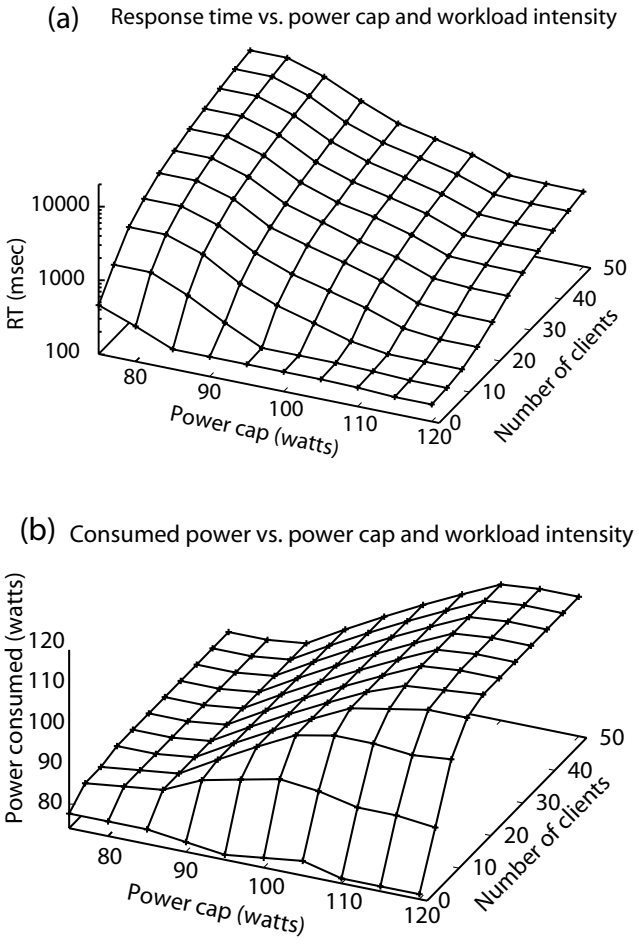

Figure 4: a) Experimentally-determined models of a) $\text{RT}(p_\kappa, n_c)$ (note log scale) and b) $\text{Pwr}(p_\kappa, n_c)$ for GB.

The curve labeled HC01 in Figure 3 is a powercap policy computed by inserting the models of Figure 4 into Eq. 5 and then using Eq. 6. The utility function is as described in Eq 4 with $\epsilon = 0.01$. One subtlety in performing the $\arg\max$ operation of Eq 6 is that we need to interpolate values of the response time and power consumption models between sampled grid points of Figure 4. Following Numerical Recipes [9], we simply identified the four nearest neighbors of each point and did a simple linear weighting of their values based on proximity.

Except for when $n_c \leq 3$, $p^*_\kappa(n_c)$ for HC01 is a monotonically increasing function. For extremely low numbers

[1] More precisely, the first measurement was taken for a single client, as a measurement for zero clients is meaningless.

of clients, the powercap has very little effect on the power consumption or the response time, so small model inaccuracies can strongly influence the computation of the optimal powercap. For very small n_c, it matters very little whether the powercap is set to 120 watts (as recommended by HC01) or 85 watts (as one might expect given the monotonicity everywhere else).

4.2.2 Machine learning derived powercap policy

Our machine learning approach leverages our recent work applying Hybrid Reinforcement Learning to autonomic resource allocation [15]. This entails devising an initial control policy, running the initial policy in the live system and logging a set of (state, action, reward) tuples, and then using standard Reinforcement Learning (RL) to train in batch mode a nonlinear function approximator $V(s,a)$ estimating cumulative expected reward of taking action a in state s. (Since U_{pp} is our reward function, $V(s,a)$ estimates the expected sum of all discounted future values of U_{pp} starting from state s and action a.) The learned value function V then implies a policy of selecting the action in state s with highest expected value, i.e., $a^* = \arg\max_a V(s,a)$.

In implementing an initial policy to be used with Hybrid RL, one would generally want to exploit the best available human-designed policy, combined with sufficient randomized exploration needed by RL, in order to achieve the best possible learned policy. However, in view of the difficulty expected in designing such initial policies, it would be advantageous to learn effective policies starting from simplistic initial policies, as was demonstrated in [15]. We have therefore trained our RL policies using an extremely simple performance-biased random walk policy for setting the powercap, which operates as follows: At every decision point, p_κ either is increased by 1 watt with probability p_+, or decreased by 1 watt with probability $p_- = (1 - p_+)$. The upward bias p_+ depends on the ratio $r = \mathrm{RT}/\mathrm{RT}_0$ of current mean response time to response time threshold according to: $p_+ = r/(1+r)$. Note that this rule implies an unbiased random walk when $r = 1$ and that $p_+ \to 1$ for $r \gg 1$, while $p_+ \to 0$ when $r \ll 1$. This simple rule seems to strike a good balance between keeping the performance near the desired threshold, while providing plenty of exploration needed by RL, as can been seen in Figure 5.

Having collected training data during the execution of an initial policy, the next step of Hybrid RL is to design an (input, output) representation and functional form of the value function approximator. For simplicity we use the basic input representation studied in [15], in which the state s is represented using a single metric of workload intensity (number of clients n_c), and the action a is a single scalar variable—the powercap p_κ. This scheme robustly produces good learned policies, with little sensitivity to exact learning algorithm parameter settings.

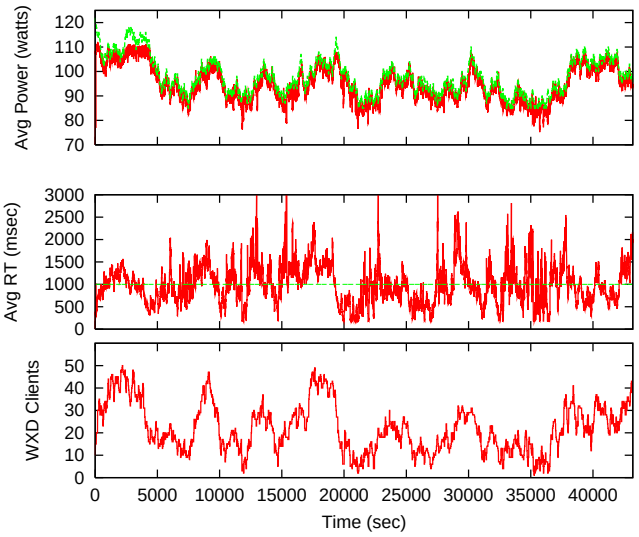

Figure 5: Random-walk powercap policy.

To represent V, we propose an innovation taking advantage of the fact that total utility (reward) U_{pp} in our system, expressed in equation 1, is a linear combination of performance utility U and power cost $-\epsilon * Pwr$. Since information regarding the reward components is generally available, and since these should have completely different functional forms relying on completely different state variables, we propose training two separate function approximators estimating V_{perf} and V_{pwr} respectively. In such a "decompositional reward" approach, the sum of the learned value function components provably converges to the correct total value function [12].

Our approach to learning V_{perf} makes use of a standard neural network (multilayer perceptron) architecture – we use a single hidden layer with 12 sigmoidal hidden units – combined with an innovative type of neuronal output unit. This is motivated by the shape of U, which is a piecewise linear function of response time, with constant value for low response times and linearly decreasing for large response times. This functional form is is not naturally approximated by either a linear or a sigmoidal transfer function. However, we can devise an appropriate transfer function by noting that the derivative of U is a step function (changing from 0 to -1 at the threshold), and that sigmoids give a good approximation to step functions. This suggests using an output transfer function that behaves as the integral of a sigmoid function: specifically, our output transfer function has the form $Y(x) = 1 - \chi(x)$ where $\chi(x) = \int \sigma(x)dx + C$, where $\sigma(x) = 1/(1+\exp(-x))$ is the standard sigmoid function, and the integration constant C is chosen so that $\chi \to 0$ as $x \to -\infty$. We find that this type of output unit is easily trained by standard back-propagation and provides

quite a good approximation to the true expected rewards.

We have also trained separate neural networks to estimate V_{pwr} using a similar hidden layer architecture and a standard linear output unit. However, we found only a slight improvement in Bellman error over a simple estimator of predicted power $\cong p_\kappa$ (although this is not always a good estimate, as seen earlier). Hence for simplicity we used $V_{pwr} = -\epsilon * p_\kappa$ in computing the overall learned policy maximizing $V = V_{perf} + V_{pwr}$. Experiments described below used two different ϵ values, 0.01 and 0.05, which nicely illustrate a range of power-performance tradeoffs. With $\epsilon = 0.01$, the learned policy generally keeps the performance quite close to the threshold value, with limited opportunities for power savings, while at $\epsilon = 0.05$ the learned policy tolerates ∼20-30% degradations in performance in order to economize more aggressively on power consumption. Both policies are illustrated in Fig. 3.

Again following [15], we use the standard Sarsa RL algorithm, setting the discount parameter $\gamma = 0.5$. We usually obtain good convergence to Bellman error minima in ∼5-10K training epochs, requiring only a few CPU minutes on a 3GHz workstation.

4.3 Dynamic power control

We ran all four of the powercap policies illustrated in Fig. 3 on the same time-varying workload for approximately 10 hours, measuring approximately 50 state variables aggregated by the data collector every 5 seconds from the four different data sources described in section 3. A small subset of the measurements taken from the first 20000 seconds (nearly 6 hours) of each run are displayed in Figures 6-7.

First, we ran the *Unmanaged* powercap policy, which was achieved by setting the powercap to 120 watts. During the course of the experiment, we confirmed that the frequency setting never dropped below the nominal frequency setting for the processor, which was 3000 MHz. The workload intensity n_c, the response time, and the consumed power are displayed in Fig. 6. Note that, during periods of low workload intensity, the performance is considerably better than necessary. This suggests that the powercap (and thus the frequency) could be reduced during those periods, potentially reducing power consumption without violating the response-time target of 1000 msec.

Next, to see whether we could realize the power savings implied by Fig. 6, we ran the dynamic HC01 powercap policy, with the result shown in Fig. 7(a). Overall, HC01 yields a response time that adheres much more closely to the 1000 msec target by lowering the powercap, and hence the consumed power, during periods of low workload intensity, and raising it when the workload intensity increases again. For example, at $t = 6000$ sec, the unmanaged policy consumes approximately 109 watts to generate a response

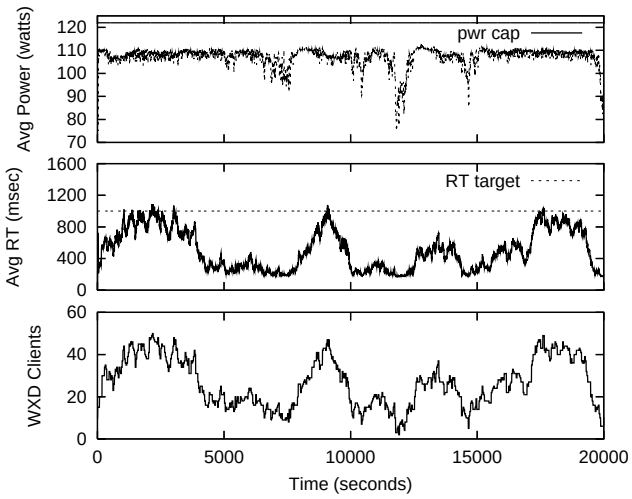

Figure 6: Experimental results for powercap policy "Unmanaged".

time of about 350 msec, while HC01 consumes approximately 97 watts to generate a response time of about 900 msec. Since a response time of 350 msec is not regarded by the utility function as being any more valuable than a response time of 900 msec, HC01 is strictly superior at this point in time because it is saving 12 watts of power. While HC01 exceeds the response-time target more often than occurs without power management, the excursions are generally tolerable, and more than made up for by the power savings.

Finally, we ran both of the powercap policies derived by machine learning, with results provided by Figures 7b ($\epsilon = 0.01$) and 7c ($\epsilon = 0.05$). The resultant powercap polices are referred to as ML01 and ML05, respectively. As can be seen in Fig. 3, the powercap policies for HC01 and ML01 are fairly similar, with ML01 being slightly less aggressive about power conservation. Thus it is not surprising that ML01 yields a response-time curve that is slightly lower than that of of HC01, and a power-consumption curve that is slightly higher. In contrast, ML05 yields a noticeably different behavior, in which the powercap and consumed power are lower, and the response time target is exceeded much more frequently. For example, at $t = 6000$, the response time is about 1000 msec while the consumed power is only 92 watts, resulting in another 5 watts savings at this particular moment in time.

Fig. 8 provides an alternative view of how the various powercap policies affect power consumption and response time by plotting these quantities as a function of the number of clients. Each data point represents approximately 100-200 samples of moments during the experimental run when n_c took on a particular value.

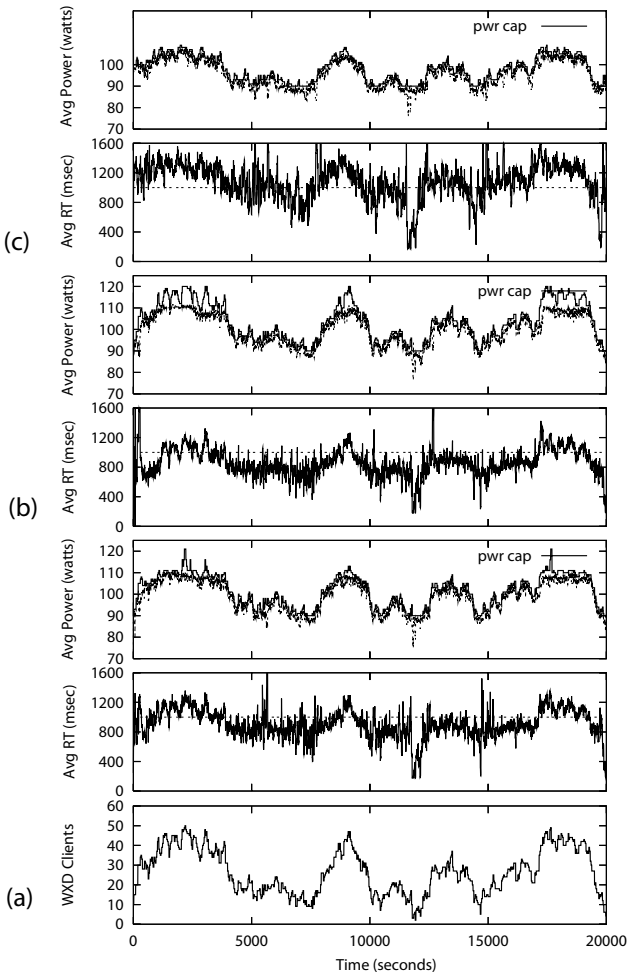

Figure 7: Experimental results for powercap policy (a) *HC01*, machine learning (b) $\epsilon = 0.01$, and (c) $\epsilon = 0.05$.

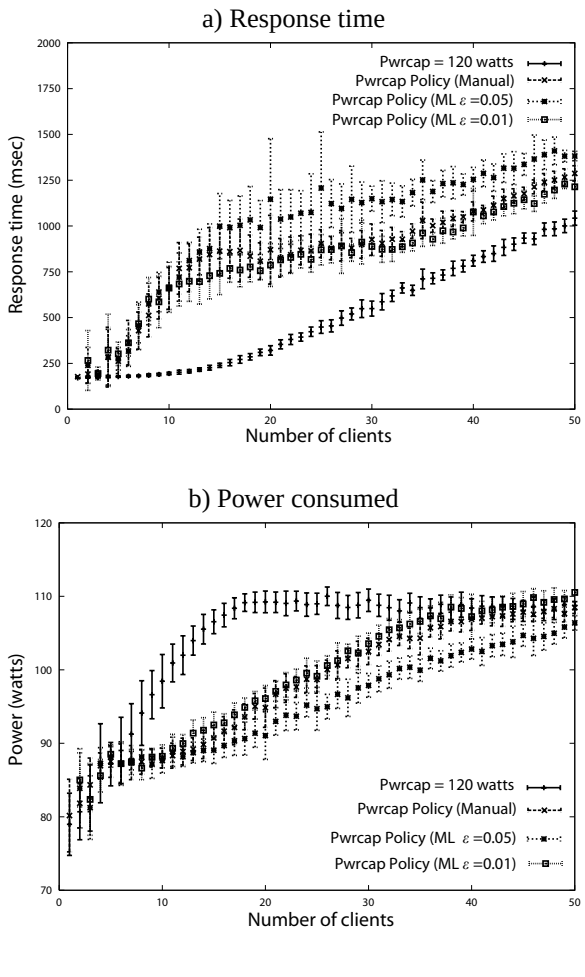

Figure 8: a) Response time and b) consumed power as a function of number of clients for several different powercap policies: FullPower, Policy1000, PolicyML01, PolicyML05.

Interestingly, the power savings occurs within a broad range of moderate workload intensities, approximately $10 < n_c < 30$. A little reflection based on experimental results presented earlier in this paper reveals why. When n_c is very low, the consumed power is small even when the powercap is large, so very little savings are possible. When n_c is high, the powercap policies all push the powercap to the highest possible value in order to bring the performance within acceptable limits, so there is little opportunity to save power. When $\epsilon = 0.01$, a maximal savings of approximately 16 watts, or about 15%, occurs when $n_c \approx 18$ for both the handcrafted and machine-learned policies, whereas the maximal savings is about 18 watts, or about 17%, for $\epsilon = 0.05$ at $n_c \approx 18$. Since the power never drops much below 80 watts in any case, it is not possible to save more than 30 watts under any circumstances, so this represents about half of the available power savings. Averaged over the full statistical distribution of n_c, the unmanaged system consumes 104.9 watts on average, compared with 95.3 watts for HC01, or a 9.2% reduction. For ML01, the average power consumption is 96.1 watts, or a reduction of 8.4%, whereas for ML05 the average power consumption is 92.7 watts, or a reduction of 11.6%.

5 Conclusions

This paper presents an initial attempt to develop a coherent policy for managing power and performance using two separate managers. The problem is non-trivial since controlling power can affect performance, and meeting performance goals may conflict with maintaining power limits or reducing the total amount of power required. The scheme described here uses utility functions to express the relationship between the power and performance goals. Once a utility function is defined, it is still a challenge to devise a policy to maximize it. Initial work with manually developed policies yields reasonable results, but it also rep-

resents a very large effort, one that is probably too difficult even for a two-blade environment, much less an entire data center. However, by applying machine learning to the data collected from the power and performance instrumentation of the system, it becomes possible to develop policies to fit the utility function and the configuration without a time-consuming manual effort. It took roughly a week of experiment time to generate the data in Figure 4, whereas the machine learning just required 12 hours of data and much less manual labor, and the amount of data required for training could almost certainly be reduced further. The hand-crafted and machine-learned policies are very comparable in form and in their engendered behavior, saving about the same amount of power. A policy generated using a different power-performance tradeoff in the utility function can save even more power, especially when the load is relatively low.

However, this paper is only an initial step. We are beginning to extend this work in several dimensions, including scaling to more blades and more types of autonomic managers (such as availability), as well as diversifying the applications. For example, while all of the experiments reported here use a single blade, we have begun to apply the same scheme with minor modifications to two-blade configurations, and we believe that our general approach should scale comfortably to a fully-populated BladeCenter with 14 blades. For multiple blade scenarios, we have implemented a second power management control action: turning blades on and off, offering an even greater potential for power savings. There is no inherent barrier to applying machine learning for this case as well.

Other potential extensions include studying whether it would be advantageous to effect some power management through the performance manager's control actions, rather than solely through the power manager, and exploring whether various types of negotiation provide some benefit over the present scheme, in which each manager reacts to the other's actions.

As power management implementations improve, the opportunities for large power savings with limited performance impact will emerge. But to take advantage of them, data centers need scalable, realistic ways to make power, performance and other types of autonomic managers work together to meet the goals and tradeoffs specified by the administrator.

Acknowledgments

We heartily thank Mike Frissora, who conjured the equipment for our experiments, and Ian Whalley and Wolfgang Segmuller for their expert assistance with the intricacies of Websphere Extended Deployment.

References

[1] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *Proc. of the Intl. Conf. on Measurements and Modeling of Computer Systems*, 2005.

[2] V. Durani. IBM BladeCenter Systems Up to 30 Percent More Energy Efficient Than Comparable HP Blades. IBM Press Release, Nov. 16, 2006.

[3] M. Femal and V. Freeh. Boosting data center performance through non-uniform power allocation. In *Second Intl. Conf. on Autonomic Computing*, 2005.

[4] Green Grid Consortium. Green grid. http://www.thegreengrid.org, 2006.

[5] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multi-tier web servers with end-to-end delay control. In *IEEE Transactions on Computers*. IEEE, 2007. To appear.

[6] IBM. WebSphere Extended Deployment. http://www-306.ibm.com/software/webservers/appserv/extend/, 2007.

[7] R. Kotla, A. Devgan, S. Ghiasi, T. W. Keller, and F. L. Rawson. Characterizing the impact of different memory intensity levels. In *Proc. of the 7th Annual IEEE Intl. Workshop on Workload Characterization*, 2004.

[8] D. Menasce and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998.

[9] W. H. Press et al. *Numerical Recipes: The Art of Scientific Computing*. Cambridge Univ. Press, Cambridge (UK) and New York, 2nd edition, 1992.

[10] O. F. Rana and J. O. Kephart. Building effective multivendor autonomic computing systems. *IEEE Distributed Systems Online*, 7(9), 2006.

[11] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. In *Proc. of the 33rd Annual Intl. Symp. on Computer Architecture*, 2006.

[12] S. Russell and A. L. Zimdars. Q-decomposition for reinforcement learning agents. In *Proc. of ICML-03*, pages 656–663, 2003.

[13] M. S. Squillante, D. D. Yao, and L. Zhang. Internet traffic: Periodicity, tail behavior and performance implications. In *System Performance Evaluation: Methodologies and Applications*, 1999.

[14] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Second Intl. Conf. on Autonomic Computing*, 2005.

[15] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proc. of ICAC-06*, pages 65–73, 2006.

[16] United States Environmental Protection Agency. Letter to Enterprise Server Manufacturers and Other Stakeholders. http://www.energystar.gov, 2006.

[17] M. Wang, N. Kandasamy, A. Guez, and M. Kam. Adaptive Performance Control of Computing Systems via Distributed Cooperative Control: Application to Power Management in Computer Clusters. In *Proc. of the Third Intl. Conf. on Autonomic Computing*, 2006.

[18] X. Wang, C. Lefurgy, and M. Ware. Managing peak system-level power with feedback control. Research Report RC23835, IBM, December 2005. Revision submitted to ICAC 2007.