

# OptiTuner: An Automatic Distributed Performance Optimization Service and a Server Farm Application

Jin Heo\*, Praveen Jayachandran\*, Insik Shin†, Dong Wang\*, and Tarek Abdelzaher\*  
\* Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801  
† Department of Computer Science, KAIST, Korea

## ABSTRACT

The next generation of real-time performance-sensitive systems is expected to be more distributed and dynamic. They will have multiple “knobs” that affect performance and resource allocation. Settings of these knobs will need to be jointly and dynamically optimized in the face of a changing workload and resource supply. Centralized approaches, such as Q-RAM, can efficiently maximize global utility of systems of multiple resources and constraints. As such systems grow in scale, complexity, and distribution, distributed solutions may be preferred at run-time. This paper uses optimization decomposition, recently applied at length in network theory, to break complex system-wide global optimization problems into less coupled subproblems that can be optimally solved at run-time in a distributed fashion. We develop a software service, called OptiTuner, that monitors the current performance and the resource availability in a performance-sensitive system and executes the distributed global optimization algorithm derived from optimization decomposition to maximize utility subject to resource and performance constraints. We apply OptiTuner to perform distributed energy optimization in a real-time Web server farm. We show that our distributed solutions achieve roughly up to 14% lower energy consumption than previous literature in our server farm testbed comprising of 18 machines.

## Keywords

Optimization Decomposition, Energy Minimization, Server Farms

## 1. INTRODUCTION

Performance-sensitive systems are becoming more complex in terms of increased system size, the number of tunable parameters that affect performance and resource consumption (which we call *performance control knobs*), and subtle interactions among the performance management modules that control these parameters. The increasing complexity has made automatic performance management of such systems extremely challenging. In order to achieve good performance in a performance-sensitive system in the presence of a changing operating environment, the performance control knobs need to be jointly and dynamically optimized at runtime.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FeBID 2009, April 16, 2009, San Francisco, California, USA.

Centralized global optimization has been widely used in the real-time literature for the purpose of performance management and resource allocation [9, 13, 6]. While such centralized approaches can efficiently solve a wide range of optimization problems, scalability of such solutions can be a serious issue considering the growing complexity, scale, and distributed environment of performance-sensitive systems today. Distributed approaches, on the other hand, may lead to suboptimal solutions due to fragmented and ill-coordinated performance management modules and the lack of global knowledge resulting in poor aggregate behavior [6].

The need to reconcile the goal of global optimization with the challenge of modular and distributed design for automatic performance optimization motivates a sound methodology to design large performance-sensitive systems. In this paper, we show how the theoretical framework of *optimization decomposition* [3] can be applied to practical distributed performance-sensitive systems. To this end, we first develop a software service, called OptiTuner, that aides the implementation of the performance management algorithms derived from theory and the interactions between them on a performance-sensitive system. Next, we apply OptiTuner to an energy cost minimization problem in a Web server farm. We show how the global optimization problem is decomposed into smaller subproblems that can operate independently with limited information exchange and yet progress towards the globally optimal solution.

OptiTuner allows performance management algorithms to independently tune the performance control knobs, while preventing the system from violating any imposed performance or resource constraints. OptiTuner manages regulation policies that execute the performance management algorithms derived from a global optimization problem, connecting performance sensors and actuators instrumented in the target system. Constraint monitors in OptiTuner ensure that the regulation policies do not violate the imposed constraints and guide the solution towards the global optimal point. The resulting architecture allows simple and modular automatic performance optimization in a performance-sensitive system.

In our previous work [6], we presented a methodology to compose multiple performance management modules in a manner that reduces possible negative interactions and achieves good aggregate behavior. The approach involved centralized control of all the performance management modules. The achieved performance may be suboptimal, since it used feedback-based heuristics to search for the optimal solution starting from certain necessary conditions for optimality. In contrast, in this paper, performance management modules are derived from a global optimization formulation to execute independently in a distributed fashion.

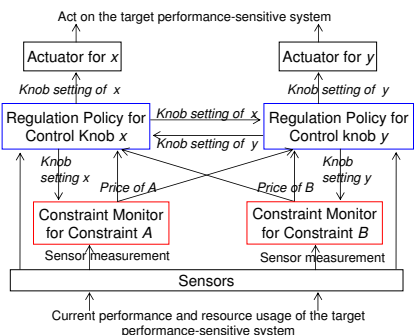
We evaluate our approach in the context of energy cost minimization in a Web server farm testbed. We install an Amazon-like

online book store in the testbed with 18 machines. The performance objective is to minimize the total energy cost of the server farm, subject to end-to-end delay constraints on client requests, and resource constraints on available computational capacity. The testbed is equipped with three different types of performance control knobs to save energy. We show how our approach decomposes the problem into smaller problems that are configured to run in OptiTuner. Finally, we show that the resulting energy cost minimization framework with OptiTuner can save more energy compared to other approaches including our previous work, demonstrating that OptiTuner indeed achieves good performance.

The rest of the paper is organized as follows. Section 2 provides an overview of our approach. Section 3 shows how optimization decomposition techniques can be applied to performance-sensitive systems. Section 4 presents the prototype implementation of OptiTuner. We present an energy cost minimization framework in a 3-tier Web server farm using OptiTuner in Section 5. We evaluate our approach in Section 6. We discuss related work in Section 7 and conclude with Section 8.

## 2. OVERVIEW OF OPTITUNER

In this paper, we use the term performance-sensitive system to refer to a system with a certain performance objective to achieve in the presence of various performance and resource constraints. Such a system may run a distributed application on a network of machines. For example, minimizing energy consumption can be a key performance objective in a Web server farm with application-level performance constraints such as bounded end-to-end response times for jobs. Different processor frequency/voltage levels and the number of available machines are possible performance control knobs that affect energy consumption. In order to achieve the performance objective of the system, the performance control knobs need to be continuously adjusted in the presence of dynamic fluctuations in the operating environment (e.g. workload changes). OptiTuner provides the necessary abstractions to realize this goal.



**Figure 1: Abstractions in OptiTuner with an example that has two control knobs  $x$  and  $y$  and two constraints  $A$  and  $B$ .**

Figure 1 shows the key aspects and abstractions employed by OptiTuner. The optimization problem is decomposed into smaller subproblems, each concerning a particular performance control knob. Each performance control knob is governed by update rules implemented by a *regulation policy*, which periodically updates the value of the performance control knob according to the values of other control knobs and the current availability of resources. A *constraint monitor* associates a *price* variable to each constraint imposed on the system, and the value of the variable is updated periodically using the rules derived from the solution to the optimization problem, based on the current values of the control knobs. For instance,

the price variable associated with a resource constraint reflects the availability of the resource - the price value increases as the availability of the resource drops. The price values are exchanged and serve to coordinate the local adaptation of individual performance control knobs so as to progress towards the global optimum solution. OptiTuner runs on all the machines involved in a server farm as a separate process to properly monitor and act on the server farm. Sensors are instrumented to collect performance measurements and resource usages that are fed to the derived regulation policies and constraint monitors. Further, actuators are instrumented to enforce the settings of the control knobs governed by the regulation policies. We formally describe how the optimization decomposition methodology can be applied to performance-sensitive systems in Section 3.

## 3. OPTIMIZATION DECOMPOSITION IN PERFORMANCE-SENSITIVE SYSTEMS

In OptiTuner, the performance objective of the target system is achieved by formulating and solving a constrained joint optimization problem. The purpose of the optimization is to find the joint settings of the performance control knobs that maximize the total utility (which represents the performance objective), subject to the various performance and resource constraints imposed.

The optimization decomposition techniques assume that the formulated constrained optimization problem is convex. When an optimization problem is non-convex in nature, approximate convex models can be used. While the approximation might compromise on optimality, they nevertheless guide the system towards improved global performance. We shall now describe the decomposition of the formulated optimization problem into subproblems (which are mapped into the corresponding regulation policies in OptiTuner), so that the associated control knobs can be independently tuned to maximize the global objective.

Let  $x_i$  denote the performance control knob setting of regulation policy  $i$ , where  $i = 1, \dots, n$  (there are  $n$  regulation policies, each governing a performance control knob). Let  $x = [x_1, \dots, x_n]^T$  denote a vector of the knob settings that need to be jointly optimized. A constrained utility maximization problem is formulated as follows:

$$\begin{aligned} \max_{x \geq 0} \quad & U(x) \\ \text{subject to} \quad & g_j(x) \leq 0, \quad j = 1, \dots, m, \end{aligned} \quad (1)$$

where  $U$  is the utility function to be maximized;  $g_j(\cdot)$ ,  $j = 1, \dots, m$ , are the resource and performance constraints. We assume that the knob settings are real numbers, the utility function  $U(\cdot)$  is concave and differentiable, and constraint functions  $g_j(\cdot)$  are convex and differentiable.

Introducing Lagrange multipliers  $p_1, \dots, p_m$  for the constraints, the Lagrangian of the problem is given as:

$$L(x, p) = U(x) - \sum_{j=1}^m p_j g_j(x), \quad (2)$$

where  $p$  denotes the vector of Lagrange multipliers,  $p = [p_1, \dots, p_m]^T$ . The Lagrange multiplier,  $p_j$ , is considered as a price for constraint  $j$ , indicating the availability of the resource corresponding to constraint  $j$ .

Note that the original problem shown in Equation (1) has the optimal point at  $x^*$ , if and only if there exists the optimal point  $(x^*, p^*)$  that maximizes the Lagrangian in Equation (2) with respect to  $x$  and minimizes the Lagrangian with respect to  $p$ . To reach the optimal point  $(x^*, p^*)$ , each regulation policy  $i$  periodically adjusts the associated control knob setting  $x_i$  to maximize

$L(x, p)$  with regard to  $x_i$  individually. In the mean time, each resource constraint monitor  $j$  periodically changes  $p_j$  to minimize  $L(x, p)$  with regard to  $p_j$ , achieving decomposition between the various regulation policies.

In each period  $t$ , each regulation policy  $i$  tries to adjust the performance control knob setting  $x_i$  to maximize  $L(x, p)$ , using the gradient method (a steepest descent algorithm) [1]:

$$x_i(t+1) = \left[ x_i(t) + \alpha_{x_i} \left( \frac{\partial U(x)}{\partial x_i} - \sum_{j=1}^m p_j \frac{\partial g_j(x)}{\partial x_i} \right) \right]^+ \quad (3)$$

where  $\alpha_{x_i}$  is a step size that determines the rate at which the error to the optimal point  $(x^*, p^*)$  is reduced<sup>1</sup>. We use a constant step size for the energy cost minimization framework in a Web server farm to better cope with workload changes, as explained later in Section 5. As long as the step size is small, the gradient method can be shown to converge to a small neighborhood of the optimal solution [1].

The price value  $p_j$  for constraint  $j$  is updated in a similar way. At invocation time  $t$ , each price value  $p_j$  is updated to minimize  $L(x, p)$  by using the steepest decent algorithm as follows ( $\alpha_{p_j}$  is a step size constant):

$$p_j(t+1) = [p_j(t) - \alpha_{p_j}(-g_j(x))]^+ \quad (4)$$

Both the step size constant and the update period affect the convergence rate and the stability of the system. Smaller step size values and larger period values would make the system more stable but converge slower upon workload changes.

Thus, the decomposition of the optimization problem enables regulation policies to update their performance control knobs asynchronously and independently of each other. The coordination of regulation policies toward the optimal point is done by exchanging the price values of the related constraints and the current settings of the dependent control knobs. This information exchange occurs locally among the dependent regulation policies and the related constraint monitors. The price values adjusted by constraint monitors serve to inform the regulation policies of how much further performance can be improved without breaking the constraints.

OptiTuner has some resemblance to market-based resource allocation approaches [15, 10] in that resource availability is expressed as prices and resource allocation is carried out in a distributed fashion based on the prices. However, participants in market-based approaches are usually assumed to be self-interested. In comparison, OptiTuner establishes a global optimization problem that leads to regulation policies that work cooperatively towards the globally optimal solution.

## 4. OPTITUNER IMPLEMENTATION

In this section, we briefly describe the prototype implementation of OptiTuner and explain the interfaces for implementing performance management algorithms derived from theory.

### 4.1 Implementation

OptiTuner provides *sensor*, *actuator*, *node*, *regulation policy*, and *constraint monitor* object classes to support the abstractions explained in Section 2 and Section 3. The regulation policy object class is used to implement the decomposed subproblems to update the associated performance control knobs, and the constraint monitor object class is used to update the price values for the constraints. The necessary sensor objects are instrumented in the application or the operating system running in the target performance-sensitive system to collect performance measurements and resource usage.

<sup>1</sup> $[Y]^+$  means that the resulting value should be greater than or equal to 0. Arithmetically it is the same as  $\max(0, Y)$

The collected data is passed on to regulation policies and constraint monitor objects. Similarly actuator objects are instrumented to enforce the performance control knob settings determined by regulation policy objects. OptiTuner connects to the instrumented sensors and actuators using a set of XML-RPC methods.

An OptiTuner process runs on each machine involved in the target performance-sensitive system. To simplify the communication between the objects running on different machines, a *node* object class is introduced to abstract the physical machines involved in the target system. The remote methods based on XML-RPC are defined on the node object class to access the objects residing on the corresponding machine. All objects including instrumented objects in the target system are registered with OptiTuner using a configuration file that is shared by all OptiTuner processes for efficient object management. The current prototype implementation of OptiTuner is written in Python.

## 4.2 Interfaces

OptiTuner provides a simple object-based interface to easily implement the derived performance management algorithms. It first defines several remote methods in the node object class for accessing the objects residing in remote nodes. With the object name and its location (the node name) that are specified in the configuration file, any objects registered in OptiTuner can be uniformly accessed. Given the name of a sensor, *node.getSensorVal(sensor\_name)* returns the value of the sensor. Actuators are invoked by calling *node.invoke(actuator\_name, value)*. These methods are mainly used by OptiTuner internally and are not directly used by programmers.

When implementing a regulation policy object, programmers specify in the configuration file, the sensor, actuator, constraint monitor, and possibly other dependent regulation policy objects that are necessary to update the control knob setting. OptiTuner automatically extracts the values of the specified objects at their invocation period, using *getSensorVal()* method of the corresponding node objects as explained above. It then stores the extracted values in an internal hash table of the regulation policy object. Programmers use this hash table to implement the performance management algorithms in *update()* method of the regulation policy object, which is invoked periodically by OptiTuner. Programmers are advised to use the newly adjusted control knob setting as the return value in *update()* method. When it returns, OptiTuner accesses the specified actuator to enforce the control knob setting by calling the *invoke()* method on the node object where the actuator is located. The interface of constraint monitors is similar, with the exception that the *update()* method returns price values instead. Further, actuators are not used, since constraint monitors don't need them when adjusting price values.

## 5. MINIMIZING ENERGY COST IN WEB SERVER FARMS WITH OPTITUNER

OptiTuner is designed primarily for server farms in a data center as a target application. In data centers, power consumption has become increasingly important, as energy bills constitute a major operational cost [12, 5].

The energy minimization problem considered in this paper is a slightly extended version of the one addressed in our previous work [6]. We opt to reuse that problem in order to compare the two solution approaches. The regulation policies in OptiTuner work independently and in a distributed fashion to guide the system towards the optimal energy consumption. In contrast, our previous approach offers a centralized solution. It uses heuristics to search for the optimal point, starting from a necessary (but not sufficient)

condition for optimality. We compare the performance of the two approaches in Section 6 together with other approaches.

In this section, we develop an energy cost minimization framework for 3-tier Web server farms. We assume that a server farm runs a typical 3-tier Web application and provides a database backup service for other organizations to fully utilize its extra computing power (e.g., CPU and storage). Client requests are to be served within pre-specified end-to-end delay constraints. Further, there are background activities performed at the database servers (3rd tier) for the database backup service. The rate of backup requests served can be dynamically adjusted using a rate controller. There is a utility associated with the rate at which backup requests are served. The total cost incurred by the system is composed of two parts: 1) the energy cost of server machines in the system and 2) less the utility achieved from database backup requests. The objective of the optimization problem is to minimize the total cost incurred by the system, subject to the delay constraints of client requests and a resource constraint on the number of available servers.

We use three different types of performance control knobs to adjust the cost of the system. First, we adjust the number of assigned machines for the server farm by dynamically turning on and off machines. Second, dynamic voltage scaling on an individual processor is used to change the speed and voltage of the processors in an active machine. Finally, we adjust the rate of the incoming database backup requests for the database tier using admission control. The three performance control knobs, that we will call On/Off, DVS, and BackupAC respectively, are periodically adjusted by their corresponding regulation policies: On/Off policy, DVS policy, and BackupAC policy.

In Section 5.1, we formulate and solve the optimization problem for a 3-tier Web server farm to derive regulation policies to update the performance control knobs. In Section 5.2, we present the implementation details of the energy cost minimization framework with OptiTuner.

## 5.1 Optimization Formulation and Decomposition into Subproblems

We assume that the load is evenly distributed across the machines at each tier in steady state. That is, all  $m_i$  machines belonging to a tier  $i$  operate at the same frequency  $f_i$ . Let  $P_{tier}(i)$  be the power consumption of tier  $i$  in the system, and is a function of the DVS frequency level and number of machines operating in that tier.  $\lambda_{backup}$  is the rate at which database backup requests are admitted (in cycles/sec). The end-to-end delay bound of client requests is denoted as  $K$ . The total number of machines available is denoted by  $M$ , and each machine is either turned off or is operating at one of the 3 tiers. It is desired to minimize:

$$\min \sum_{i=1}^3 P_{tier}(i) - h \cdot \log(\lambda_{backup}) \quad (5)$$

$$\text{subject to} \quad \sum_{i=1}^3 Delay(i) \leq K, \quad \sum_{i=1}^3 m_i \leq M. \quad (6)$$

Power consumption at tier  $i$ ,  $P_{tier}(i)$ , is estimated based on the current frequency,  $f_i$ , and the number of machines operating at tier  $i$ :

$$P_{tier}(i) = m_i \cdot P_{machine} = m_i \cdot (A_i \cdot f_i^n + B_i), \quad (7)$$

where  $B_i$  is the power consumption that is independent of the current frequency and  $A_i$  is a positive constant that indicates the effect of the frequency on power.  $A_i$ ,  $B_i$ , and  $n$  can be measured using offline experiments. In general, power changes linearly with frequency and quadratically with voltage [4]. Further, a change in

voltage involves a proportional change in frequency. Hence, we will use the value 3 for  $n$  throughout this paper.

Since  $f_i$  is discrete in a real system and is not differentiable, we cannot directly use it in the optimization formulation. To solve this problem, the steady-state result of an M/M/1 queuing model [8],  $utilization = \lambda/\mu$ , is used where  $\mu$  is the service rate and  $\lambda$  is the arrival rate. We use a queuing model, because it has been widely used for estimating various performance metrics for computing systems such as delay and resource utilization [7]. It yields:

$$U_i = \frac{\lambda_i/m_i}{f_i} = \frac{\lambda_i}{f_i m_i}, \quad (8)$$

where  $\lambda_i$  denotes the rate of arrival of requests to tier  $i$ , and  $U_i$  denotes the utilization of a machine at tier  $i$ . Using the relationship shown in Equation (8), we rewrite Equation (7):

$$P_{tier}(i) = m_i \cdot P_{machine} = m_i \cdot \left( \frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right). \quad (9)$$

Also,  $\lambda_{backup}$  is estimated using the relationship between  $U_{backup}$ ,  $m_3$ , and  $f_3$  from Equation (8):

$$\lambda_{backup} = U_{backup} \cdot m_3 f_3 = (U_3 - U_{user}) m_3 f_3, \quad (10)$$

where  $U_{backup}$  is the utilization of backup requests alone at the third tier, and  $U_{user}$  denotes the utilization of user requests.

Assuming the load is equally distributed over the machines at each tier, the delay experienced at each tier  $i$ ,  $Delay(i)$ , is estimated using the steady state result of M/M/1 queuing:

$$Delay(i) = \frac{C_i}{f_i - \frac{\lambda_i}{m_i}} = \frac{m_i C_i}{\lambda_i} \cdot \frac{\frac{\lambda_i}{m_i f_i}}{1 - \frac{\lambda_i}{m_i f_i}} = \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i}, \quad (11)$$

where  $C_i$  is a constant in cycles per request to normalize the delay, since  $f_i$  is measured in cycles and  $\lambda_i$  is estimated in cycles/sec rather than in requests/sec.

Finally, the resulting cost minimization problem is formulated with the control knobs  $x = [U_1 \ U_2 \ U_3 \ U_{backup} \ m_1 \ m_2 \ m_3]^T$  and the two constraints as follows:

$$\begin{aligned} \min \quad & \sum_{i=1}^3 m_i \left( \frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) - h \cdot \log(U_{backup} \cdot m_3 f_3) \\ \text{subject to} \quad & \sum_{i=1}^3 \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \leq K, \\ & \sum_{i=1}^3 m_i \leq M \end{aligned} \quad (12)$$

The Lagrangian of the problem (1) is formulated as:

$$\begin{aligned} L(x, p_1, p_2) = & \sum_{i=1}^3 m_i \left( \frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) - h \cdot \log(U_{backup} \cdot m_3 f_3) \\ & + p_1 \cdot \left( \sum_{i=1}^3 \left( \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \right) - K \right) + p_2 \cdot \left( \sum_{i=1}^3 (m_i) - M \right), \end{aligned} \quad (13)$$

where  $p_1, p_2 \geq 0$  denote the Lagrange multipliers for the two constraints.

By differentiating the Lagrangian with respect to each of the optimization variables, we can decompose the problem into subproblems that adapt their knob setting individually. Seven different regulation policies are created to iteratively adjust the 7 knob settings,  $x = [U_1 \ U_2 \ U_3 \ U_{backup} \ m_1 \ m_2 \ m_3]^T$ . A constraint monitor is created to adapt the two price values,  $p_1$  and  $p_2$ , for the two constraints.

By differentiating the Lagrangian with respect to  $U_1$  and  $U_2$ , the update equations for the DVS policies at tiers 1 and 2 are obtained as:

$$U_i(t+1) = \left[ U_i(t) - \alpha_{U_i} \left( -\frac{3A_i \lambda_i^3}{m_i^2 U_i(t)^4} + \frac{p_1 m_i C_i}{\lambda_i (1 - U_i(t))^2} \right) \right]^+ \quad (14)$$

Once  $U_i$  is calculated, the corresponding frequency,  $f_i$ , is determined using Equation (8). Since the frequency value is discrete in a real system, we choose the closest frequency setting to the calculated value,  $f_i$ . The DVS Policy at tier 3 updates its knob setting  $U_3$  in a similar way:

$$U_3(t+1) = \left[ U_3(t) - \alpha_{U_3} \cdot \left( -\frac{3A_3 \lambda_3^3}{m_3^2 U_3(t)^4} + \frac{p_1 m_3 C_3}{\lambda_3 (1 - U_3(t))^2} - \frac{h}{U_3(t) - U_{user}} \right) \right]^+ \quad (15)$$

The DVS policy at tier  $i$  determines its control knob setting,  $U_i$  (hence  $f_i$ ) every second, since the frequency can be changed very quickly. For all tiers, we use 0.05 for the step size constants,  $\alpha_{U_i}$ .

Since the control knob settings  $U_{backup}$  and  $U_3$  are related to each other, we calculate  $U_{backup}$  based on  $U_3$ :

$$U_{backup}(t+1) = \frac{\lambda_{backup}}{\lambda_3} U_3(t+1). \quad (16)$$

With the calculated  $U_{backup}$ , the BackupAC policy adjusts the portion of the incoming database backup requests accordingly. It uses a control theoretic approach to keep the measured utilization for the backup requests around the set point,  $U_{backup}$ , by properly adjusting the admission rate for the incoming backup requests.

The On/Off policies at tiers 1 and 2 update their control knob settings  $m_1$  and  $m_2$  at each invocation period as follows:

$$m_i(t+1) = \left[ m_i(t) - \alpha_{m_i} \cdot \left( -\frac{2A_i \lambda_i^3}{m_i(t)^3 U_i^3} + B_i + \frac{p_1 C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} + p_2 \right) \right]^+ \quad (17)$$

The On/Off policy at tier 3 updates its control knob setting  $m_3$  as follows:

$$m_3(t+1) = \left[ m_3(t) - \alpha_{m_3} \cdot \left( -\frac{2A_3 \lambda_3^3}{m_3(t)^3 U_3^3} + B_3 + \frac{p_1 C_3}{\lambda_3} \cdot \frac{U_3}{1 - U_3} + p_2 - \frac{h}{m_3} \right) \right]^+ \quad (18)$$

The update period for the On/Off policies for all tiers is set to 60 seconds to prevent the machines from being turned on and off too frequently.

The constraint monitor periodically adjusts  $p_1$  and  $p_2$  respectively using the following update rules:

$$p_1(t+1) = \left[ p_1(t) + \alpha_{p_1} \left( \sum_{i=1}^3 \left( \frac{m_i C_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \right) - K \right) \right]^+ \\ p_2(t+1) = \left[ p_2(t) + \alpha_{p_2} \left( \sum_{i=1}^3 (m_i) - M \right) \right]^+ \quad (19)$$

Both  $p_1$  and  $p_2$  are updated every second.  $\alpha_{p_1}$  is set to 0.05 and  $\alpha_{p_2}$  is set to 0.5. Note that the direction of changes in the update rules are opposite to the ones explained in Section 3, since the objective is to minimize the total cost rather than to maximize utility.

## 5.2 Implementing Derived Regulation Policies with OptiTuner

The decomposition of the optimization problem results in regulation policies to adjust the seven different performance control knobs and a constraint monitor to adjust the two price values for

the constraints. Figure 2 describes the resulting architecture at tier 1 with OptiTuner. Tier 2 has a similar architecture, while tier 3 has one more policy for database backup requests that is not shown in the figure.

Three *DVSPolicy* objects were implemented to perform the adaptation rules, described in Equation (14) for the first and the second tier and Equation (15) for the third tier. Similarly, three *On/OffPolicy* objects were implemented to perform the adaptation rules specified in Equation (17) for the first and the second tier and Equation (18) for the third tier. The *BackupACPolicy* object executes the update rules in Equation (16). As all policies work on tiers instead of individual machines, they are placed in one of the machines selected as the leader at each tier. The leader is statically defined and specified in the configuration file. The *ConstraintMonitor* object implements the update rules for the two constraints described in Equation (19) and runs on the leader machine of each tier. The constraint monitor needs some information (e.g. the average utilization and the number of machines currently used at each tier) from all three tiers to adjust the price values. While this entails that there is exchange of information between all the tiers, the regulation policies at different tiers are still implemented in a distributed fashion and their control knobs are adjusted asynchronously and independently of each other.

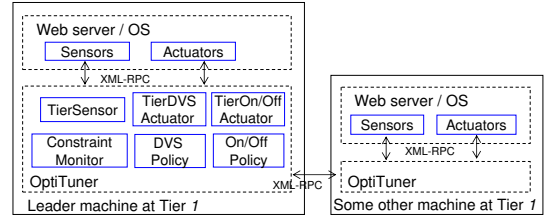


Figure 2: Implementation of tier 1 with OptiTuner

We also implemented tier-wise sensor objects called *TierSensor* that collect performance measurements and resource usage information such as the utilization, request rate, and response time for the corresponding tier that are required to execute the regulation policy and constraint monitor objects. They aggregate performance measurements collected from the sensors instrumented in the individual machines at the corresponding tier. Similarly, we implemented two actuators, *TierDVSActuator* and *TierOn/OffActuator*, that act on the DVS and On/Off actuators instrumented in the individual machines.

## 6. EVALUATION

In this section, we evaluate OptiTuner on the energy cost minimization framework in our 3-tier Web server farm testbed of 18 machines. We first describe the setup of the testbed and then presents results from experiments without and with database backup requests, respectively. We do experiments without database backup requests for an appropriate comparison with the approach used in our previous work [6] that didn't consider database backup requests.

### 6.1 Testbed Setup

We constructed a testbed for a three-tier Web server farm with a total of 18 machines. In the setup, we used Apache 2.0 Web server for the first tier, Tomcat for the second tier, and MySQL 5.0 for the third tier. We used Apache 2.3 with `mod_proxy_balancer` support to distribute the incoming load for the first tier machines. Database servers in the third tier are load-balanced using a JDBC-based database clustering solution. Each tier was given 5 machines

with Intel Celeron 2.53GHZ CPU and 512MB RAM. We used the p4-clockmod driver to change the processor frequency. Load balancers for the first and third tiers run on machines with Intel Pentium 4 3GHZ CPU and 2GB RAM. Another machine with an Intel Pentium 4 3GHZ CPU and 2GB RAM was used to generate backup requests for the database tier. MySQL Proxy is installed in the same machine to implement an admission control mechanism for the backup requests. All machines were equipped with Redhat Fedora Core 4 Linux and run OptiTuner as described in Section 5.2. We installed a three-tier Web application on our testbed based on TPC-W [14] a transactional web benchmark.

In the following experimental results, 1500 seconds of the TPC-W shopping mix workload was applied for each test run. The user think-time between consecutive requests from one EB is set to 1.0 sec. The performance objective is to minimize energy cost subject to the end-to-end delay constraint of 0.5 sec (500 msec) and the resource constraint of the given 15 machines.

In the current testbed, the On/OffPolicy in the third tier is inactive, since the database clustering solution doesn't fully support consistent data migration between replicas. This setup, however, is reasonable considering the database tier is the most overloaded tier.

Power consumption is estimated based on the frequency samples of the processors measured every second, using Equation (7). It allows us to evaluate various energy saving approaches on system settings with different DVS capabilities. In Equation (7),  $B$  represents the fixed energy consumption regardless of the current frequency and  $A$  is a coefficient for calculating the effect of frequency changes. We evaluate our approach with other approaches assuming three different system settings: when  $B$  is 70%, 50%, and 30% of the power consumption at the maximum frequency. The power consumption at the maximum frequency is measured with a AC power meter. More detailed explanation about the power estimation is presented in our previous paper [6].

## 6.2 Experiments Without Backup Requests

We evaluate four different energy saving approaches in the absence of backup requests. In this set of experiments, we only consider two different types of performance control knobs, DVS and On/Off knobs.

The *Ondemand governor* approach uses only DVS knob. It uses the Linux Ondemand governor [11], which is a dynamic in-kernel CPU frequency governor that changes CPU frequency levels for energy saving based on CPU utilization. The *independent* approach uses two independent performance management modules, On/Off policy and DVS policy for the two control knobs, On/Off and DVS knobs. They determine their knob settings in isolation only on the basis of the current load. Therefore, the two knobs are not jointly optimized. The *centralized-heuristic* approach, described in our previous work, adapts both performance control knobs in a centralized way. It derives a set of necessary (but not sufficient) conditions for optimality, then uses feedback control to achieve the desired end-to-end delay while satisfying the necessary conditions. The approach presented in this paper decomposes the global optimization problem, naturally deriving separate On/Off and DVS policies for the two control knobs. Both our approach and the centralized-heuristic approach try to jointly optimize the two performance control knobs.

Figure 3 shows the power consumption of the various approaches for different values of the parameter  $B$  representing systems with different DVS capabilities. OptiTuner achieves the least power consumption in all the three different system settings, saving up to 14% of energy compared to the next best approach at high loads. The centralized heuristic approach comes next (it may not be optimal

as it uses a heuristic feedback approach to search for the optimal point). The Linux Ondemand governor approach has higher energy consumption as it uses only one performance control knob. The independent approach adjusts the two performance control knobs independently, incurring higher energy consumption as the two knobs are not jointly optimized.

The Linux Ondemand approach achieves the highest throughput and lowest delay as shown in Figure 4(a) and 4(b), respectively, but this improved performance is at the expense of increased power consumption. Our approach shows comparable performance in terms of throughput and delay when the workload is not high. At higher workload, our approach shows slightly lower throughput and higher delay in exchange for the better power savings.

## 6.3 Experiments with Backup Requests

In this set of experiments, we consider database backup requests that results in an additional performance control knob, BackupAC knob, that determines the admission rate for the backup requests, in addition to the two performance control knobs, DVS and On/Off knobs. We evaluate five different energy saving approaches.

The *Ondemand governor* approach uses two independent performance management modules: the Linux Ondemand governor (for DVS knob) and BackupAC policy (for BackupAC knob). The BackupAC policy controls the incoming backup request rate based on the end-to-end delay constraint regardless of the decision of the Ondemand governor. The *independent* approach uses three independent performance management modules: On/Off policy, DVS policy and BackupAC policy. All three policies independently determine their knob settings based on the current load. The *centralized-heuristic-independent* approach combines the centralized-heuristic approach that jointly adapts DVS and On/Off knobs, with a BackupAC policy for database backup requests that runs independently of the other two control knobs. Similarly, the *OptiTuner-independent* approach uses an independent BackupAC policy for the database backup requests and the other two knobs are adjusted by the corresponding policies derived from optimization decomposition. Finally, our approach has the DVS, On/Off, and BackupAC policies for the three control knobs co-adapted as per the optimization decomposition methodology described.

For different system settings obtained by varying the parameter  $B$ , OptiTuner incurs the least power consumption at high workloads as shown in Figure 5. When the workload is not high, the power consumption of OptiTuner is comparable to that of the independent, the central-heuristic-independent, and the OptiTuner-independent approaches. However, OptiTuner accepts more backup requests as shown in Figure 6(c). This can be attributed to the fact that our scheme minimizes energy cost which is a function of energy consumption and the number of served backup requests. The overall cost incurred by OptiTuner is lesser than that of other approaches as it serves more backup requests while incurring nearly the same power consumption. While the Linux Ondemand approach performs poorly at low workloads (has more machines turned on than required), it performs better than the independent, the central-heuristic-independent, and OptiTuner-independent approach at high workloads. This is because, as the workload increases, the settings of all three control knobs are not jointly adapted in these approaches. Hence, they spend more energy than the Linux Ondemand approach that uses only two control knobs.

This ill-coordinated behavior of the performance management modules at high workloads in the independent, the central-heuristic-independent, and the OptiTuner-independent approach, becomes more obvious from Figure 6(c). At high loads, the number of backup requests served by these approaches actually increases with

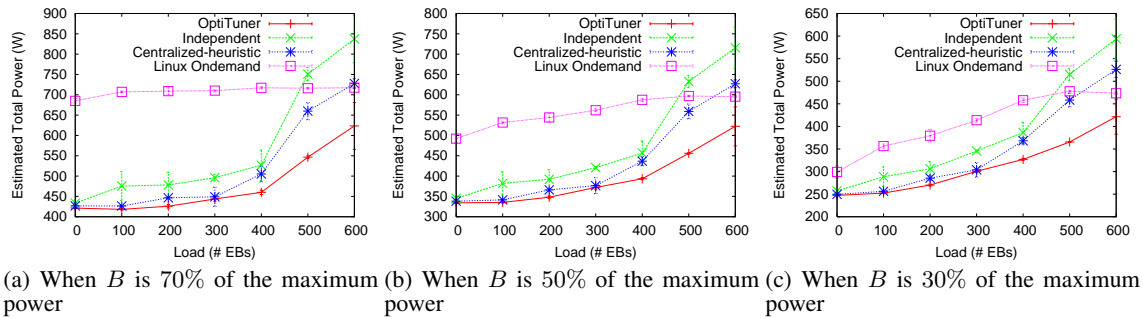


Figure 3: Performance (power consumption) comparison between various approaches without backup requests

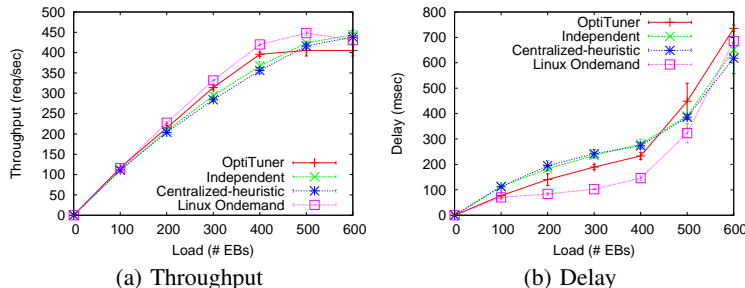


Figure 4: Other metric comparison between various approaches without backup requests

system load. This is an instance of undesirable interaction between the performance management algorithms when they execute independently. For instance, as the load increases, the DVS policy increases the CPU frequency, and the On/Off policy turns on more machines. The BackupAC policy, acting independently, notices more available capacity in the system (due to the increased frequency level and number of machines) and admits even more backup requests. In turn, the DVS policy further increases the CPU frequency and the On/Off policy turns on more machines, resulting in a vicious unstable cycle. This shows the need for jointly optimizing the various performance control knobs, and the approach presented in this paper achieves this objective, while still providing a modular design and implementation with OptiTuner.

For the same experiment, the throughput, end-to-end delay, and the number of backup requests served are shown in Figure 6(a), 6(b), and 6(c), respectively. Our approach shows comparable performance with the other schemes with respect to all these metrics, showing that our approach does not compromise on throughput or delay in order to achieve the reduced energy consumption. By admitting more backup requests when the user load is low, and by always operating close to the optimal solution, our approach minimizes energy cost while still ensuring that the delay constraints of user requests are met.

## 7. RELATED WORK

Recent breakthroughs in networking literature have led to the development of a mathematical theory for optimally layering network protocols using *optimization decomposition* techniques [3, 2] to maximize the global utility of the involved network components. Optimization decomposition as applied to network layering has been summarized in [3]. In this work, we show how optimization decomposition techniques can be applied to performance and resource management problems in performance-sensitive systems.

Utility maximization has been widely adopted in the real-time literature [9, 13, 6]. Q-RAM [9] provides a centralized optimization solution to assign available system resources to applications along multiple QoS dimensions in a way that maximizes the total utility. It supports discrete QoS operating points and non-convex utility functions that are not supported by OptiTuner. In contrast, OptiTuner provides a distributed algorithm that can be executed with information available locally, yet progressing towards the globally optimal solution, while Q-RAM only provides a centralized solution requiring global information. Although OptiTuner focuses on a smaller set of problems assuming continuous variables and convexity of the problem, we demonstrated that OptiTuner works well in a practical soft-real time system.

In our previous work [6], the design methodology for composing regulation policies required a centralized module that invokes a heuristic search to progress towards the optimal solution. On the contrary, OptiTuner derives a set of smaller independent functions coordinated in a distributed fashion, while achieving global optimization. We also provide a software service that exports abstractions to aid the implementation of the derived distributed solution.

With an expanding number of server machines in large data centers, a large portion of maintenance cost is due to energy bills [12, 5]. Hence, significant research efforts have been expended to save power of computing systems. Dynamic voltage scaling (DVS) has been one of the most popular power saving mechanisms [12, 5]. In this work, we consider the problem of applying multiple power saving mechanisms (such as DVS, switching off machines, and admission control), and jointly optimizing them for performance subject to resource constraints.

## 8. CONCLUSION

In this paper, we presented an automatic performance optimization service in distributed performance-sensitive systems, called

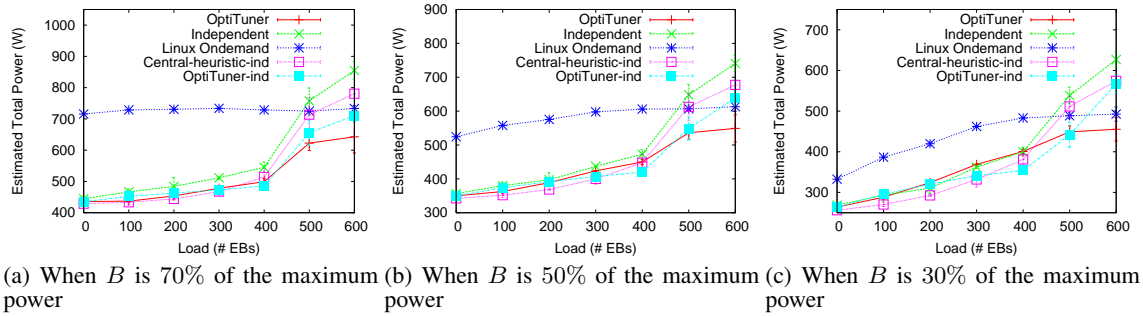


Figure 5: Performance (power consumption) comparison between various approaches with backup requests

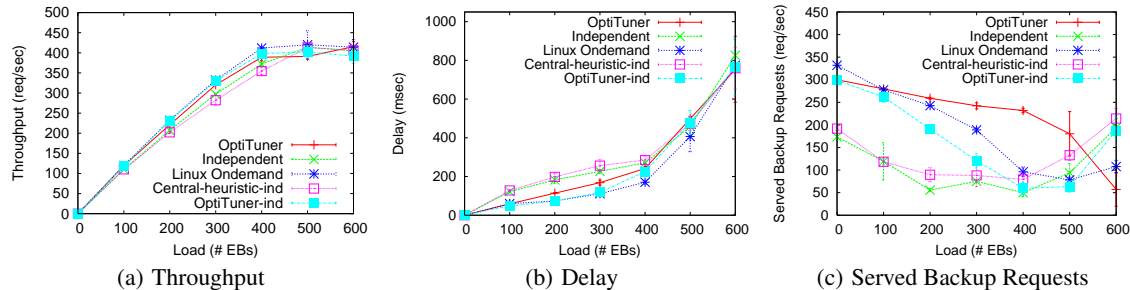


Figure 6: Other metric comparison between various approaches with backup requests

OptiTuner. We applied it to a practical performance-sensitive system to demonstrate the efficacy. OptiTuner uses optimization decomposition to break down the global optimization problem into smaller subproblems, which can independently execute with minimal information exchange, and yet collectively optimize system performance. OptiTuner provides proper abstractions to help the implementation of the performance management algorithms derived from theory. It aims to achieve the performance objective by appropriately adjusting the performance control knobs of the target system as determined by the performance management algorithms. The presented approach was applied to an energy cost minimization framework in a Web server farm. Results from the evaluation in a server farm testbed with 18 machines showed that OptiTuner was able to reduce total energy consumption by intelligently composing multiple regulation policies.

## 9. REFERENCES

- [1] D. P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1995.
- [2] L. Chen, S. H. Low, and J. C. Doyle. Joint tcp congestion control and medium access control. In *IEEE Infocom*, volume 3, pages 2212–2222, 2005.
- [3] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007.
- [4] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *PACS*, pages 179–196, 2002.
- [5] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
- [6] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *IEEE RTSS*, pages 227–238, 2007.
- [7] M. G. Kienzie and K. C. Sevcik. Survey of analytic queueing network models of computer systems. *SIGMETRICS Perform. Eval. Rev.*, 8(3):113–129, 1979.
- [8] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.
- [9] C. Lee, J. Lehoczy, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. In *IEEE RTSS*, pages 315–326, 1999.
- [10] Özgür Erçetin and L. Tassiulas. Market-based resource allocation for content delivery in the internet. *IEEE Transactions on Computers*, 52(12):1573–1585, 2003.
- [11] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proc. of the Linux Symposium*, volume 2, 2006.
- [12] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *ACM ASPLOS XIII*, pages 48–59, 2008.
- [13] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, volume 0, pages 55–60, 2005.
- [14] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce).
- [15] W. Wang and B. Li. Market-based self-optimization for autonomic service overlay networks. *IEEE JSAC*, 23:2320–2332, 2005.