

Integrating Adaptive Components: An Emerging Challenge in Performance-Adaptive Systems and a Server Farm Case-Study *

Jin Heo¹, Dan Henriksson¹, Xue Liu², Tarek Abdelzaher¹

¹ Department of Computer Science, University of Illinois at Urbana-Champaign

² School of Computer Science, McGill University

Abstract

The increased complexity of performance-sensitive software systems leads to increased use of automated adaptation policies in lieu of manual performance tuning. Composition of adaptive components into larger adaptive systems, however, presents challenges that arise from potential incompatibilities among the respective adaptation policies. Consequently, unstable or poorly-tuned feedback loops may result that cause performance deterioration. This paper (i) presents a mechanism, called adaptation graph analysis, for identifying potential incompatibilities between composed adaptation policies and (ii) illustrates a general design methodology for co-adaptation that resolves such incompatibilities. Our results are demonstrated by a case study on energy minimization in multi-tier Web server farms subject to soft real-time constraints. Two independently efficient energy saving policies (an On/Off policy that switches machines off when not needed and a dynamic voltage scaling policy) are shown to conflict leading to increased energy consumption when combined. Our adaptation graph analysis predicts the problem, and our co-adaptation design methodology finds a solution that improves performance. Experimental results from a 17-server farm running the industry standard TPC-W e-commerce benchmark show that co-adaptation renders a cut-down in energy consumption by more than 50%, when workload is not high, while maintaining latency within acceptable bounds. The paper serves as a proof of concept of the proposed conflict-identification and resolution methodology and an invitation to further investigate a science for composing adaptive systems.

1. Introduction

This paper addresses the problem of composition of adaptive components in large software systems. The topic

is motivated by the growing complexity of modern time-sensitive and performance-sensitive systems, which increasingly calls for adaptive capabilities to reduce the need for manual tuning. Automatic energy management policies [12, 10, 14, 21] and rate adaptation techniques (e.g., in 802.11 cards [11, 23]) are common examples of adaptive policies in current systems. Recent trends, exemplified by IBM's autonomic computing initiative [13], suggest increased need for self-managing, self-calibrating, self-healing, and self-tuning components in modern software. These components fundamentally rely on adaptation. A proliferation of adaptive policies is thus expected in future software. While automatic adaptation can significantly reduce the need for human intervention (and hence software ownership cost), it poses serious problems at system integration time. Namely, while each adaptive policy might work well in isolation, a system composed from multiple adaptive components may exhibit unexpected behavior that results from adverse interactions between different such policies at run-time.

To give an example of adverse interactions, consider an illustration from mobile ad hoc networks. In some real-time application, let shortest-delay routing be an adaptive policy that constantly discovers shorter-delay routes between sources and destinations. The MAC-layer rate adaptation policy, on the other hand, tunes the radio transmission rate to match channel quality (a lower rate is used on lower quality channels) [11]. Composing the two policies leads to an adverse interaction. Shortest delay routing may prefer longer hops (so there is fewer of them on the path). Longer hops tend to have lower quality, which causes the radio to lower its transmission rate. At the lower rate, even longer hops can be discovered, which further reduces channel quality leading to additional rate reductions. This adverse feedback cycle ultimately diminishes throughput. Composition problems in software systems are expected to get worse as these systems become more complex (i.e., made of more components) and feature more capabilities for adaptation.

This paper presents two contributions. First, we present a simple mechanism for identifying potentially adverse in-

*This work was funded in part by NSF grants CNS 05-53420, CNS 06-13665, CNS 06-15301, and CNS 07-20513. Also it was funded in part by the Wenner-Gren Foundations, and the Hans Werthen Foundation, Sweden.

teractions between adaptive policies at component composition time. Second, when adaptive policies are shown to interact, we describe a methodology for designing co-adaptation, i.e., joint adaptation where the adverse interaction is eliminated.

When adaptive policies optimize different contradictory objectives, the fix cannot be entirely automated. It is up to the designer to address the problem. For example, the designer might define a utility for each of the goals and then attempt to optimize total utility. Our framework helps the designer by identifying the adverse interaction loop which needs to be broken (e.g., by removing or altering one of the interacting components) or reconciled (e.g., by defining a shared currency such as utility to be jointly optimized). It also provides guidance for designing a single co-adaptive policy that optimally achieves the common objective (such as maximum utility).

A running case study illustrates our approach in the paper. We present an autonomous energy optimization framework for multi-tier delay-sensitive Web server farms. Energy saving is under the constraint that the server must meet a given end-to-end latency bound on served requests. Our choice of case study is motivated by the importance of energy saving in multi-tier Web server farms. The cost of energy is one of their dominant operating costs. In large server farms, it is reported that 23-50% of the revenue is spent on energy [10, 4]. This is because in order to handle peak load requirements, server farms are typically over-provisioned based on offline analysis.

In our energy optimization framework for Web server farms, two adaptive policies are used to save energy during off-peak load conditions. The first policy dynamically powers off underloaded machines and distributes their load across other machines in their tier. Conversely, when all active (*On*) machines exhibit high utilization, extra machines are powered on to accommodate traffic bursts. We call this scheme the *On/Off* policy. The second policy, which we refer to as the *DVS* policy, exploits dynamic voltage scaling (DVS) [10, 4, 9] on individual processors such that the speed and voltage of an active machine are reduced when the machine is underloaded and increased when it is overloaded. We show how unexpected adverse interactions may occur and make it non-trivial to integrate these adaptive policies into one coherent control framework. We present a mechanism, based on an abstraction we call *adaptation graphs*, for identifying potentially adverse interactions. This methodology uncovers the composition problem between the two policies. Since it is possible to find a single optimization objective that covers both policies, we show how to design co-adaptation of both *On/Off* and *DVS* “knobs” using our general design methodology. A multi-tier server farm testbed of 17 machines was implemented to illustrate the energy inefficiencies that arise from compos-

ing the two policies without regard to their interactions, as well as to evaluate the proposed scheme, showing significant performance improvement.

The rest of the paper is organized as follows. Section 2 describes the mechanism for identifying potentially adverse interactions between adaptation policies, and applies it to the case study. Section 3 presents a design methodology for co-adaptation of adaptive policies and an example of it, where we derive necessary conditions for optimality and a feedback correction algorithm in the combined energy minimization problem. Section 4 presents implementation aspects of the server farm testbed and Section 5 presents evaluation on our experimental testbed. Finally, conclusions and suggestions for future work are given in Section 6.

2. Composition of Adaptive Policies

In this section, we explain the problem of composition of multiple adaptive policies in large performance-sensitive systems and present adaptation graph analysis to facilitate detecting such problems.

A significant number of QoS adaptation and other performance adaptation policies have been reported in real-time computing literature over the last decade (e.g., [19, 17, 18, 2, 22, 1, 8]). In much of the current literature, these policies are designed and evaluated in isolation, showing efficacy in achieving the performance requirements of the system. However, unintended interactions between independently designed adaptive policies have not traditionally been addressed. As these policies gain popularity in deployed systems, a significant number of applications, middleware components, and operating system mechanisms will exhibit adaptive behavior. A performance-sensitive software system of the future will therefore likely include multiple adaptive components. For example, it might include an adaptive CPU allocation in the kernel-level scheduler, an adaptive admission controller at the application layer, and perhaps an adaptive energy management module in middleware. While these components will perform well in isolation, the interactions between them must be well understood to prevent unintended consequences.

The question addressed in this section is how to discover unintended interactions between adaptive components in a large system (before it is deployed). To appreciate the answer suggested below, note that adaptive components are at their very essence feedback loops. These loops take measurements of the system (e.g., request rate, delay, utilization, or queue length) and respond by a corresponding action that adjusts some variable, such as the amount of resources allocated to a given process or the priority of a thread. Within each independently designed subsystem, the adaptation loop is typically well tuned. A problem occurs if an unintended loop or interaction emerges by virtue of composition of multiple subsystems.

2.1. Adaptation Graphs

To uncover unintended loops, we present the notion of adaptation graphs. Nodes in an adaptation graph represent the key variables in the system such as delay, throughput, utilization, length of different queues, settings of different policy knobs, etc. Arcs represent the direction of causality. For example, consider a Web server that serves requested pages over a network. When the utilization, U , of the outgoing link (connecting the server to the Internet) increases, the delay, D , of served requests increases as well (because they wait longer to be sent over the congested link). Hence, an arc exists from utilization to delay, $U \rightarrow D$, indicating that changes in the former affect the latter. The arcs in the adaptation graph are annotated by either a “+” or a “-” sign depending on whether the changes are in the same direction or not. In our above example, since an increase in utilization causes a *same-direction* change in delay (i.e., also an increase), the arc is annotated with a “+” sign: $U \rightarrow^+ D$. Moreover, some of the arcs represent fundamental natural phenomena (for example, an increase in delay is a natural consequence of an increase in utilization). Others, however, represent *programmed* behavior, or policies. For example, an admission controller of a performance-aware server may be programmed to decrease the fraction of admitted requests, R , to the server in response to an increase in delay, D . Hence, an arc exists in the adaptation graph from delay to admitted requests, $D \rightarrow^- R$. The arc is annotated with a “-” sign because an increase in delay results in a change in the opposite direction (i.e., a decrease) in admitted requests. This arc does not represent a natural phenomenon but rather the way the admission control policy is programmed. We call such arcs *policy arcs* and annotate them with the name of the module implementing the corresponding policy. Hence, we have $D \rightarrow_{AC}^- R$, where AC stands for the admission control module. Figure 1(a) depicts the adaptation graph of the server under consideration. The graph is composed of three arcs. The arc $D \rightarrow_{AC}^- R$ reflects that the admission controller reduces the number of admitted requests when delay increases and vice versa. The arc $R \rightarrow^+ U$ reflects the natural phenomenon that any changes in the number of admitted requests result in same-direction changes in outgoing link utilization. Finally, the arc $U \rightarrow^+ D$ expresses the fact that changes in link utilization cause changes in delay (in the same direction). The three arcs form a cycle (a feedback loop). An interesting property of the loop is that the product of the signs of the arcs is negative. This indicates a negative feedback loop, which is expected (since all stable feedback loops are negative).

As another example, consider a network power management middleware that measures network utilization, U , on the LAN of a load-balanced server cluster. If the network utilization is low, the cluster workload must be low. The

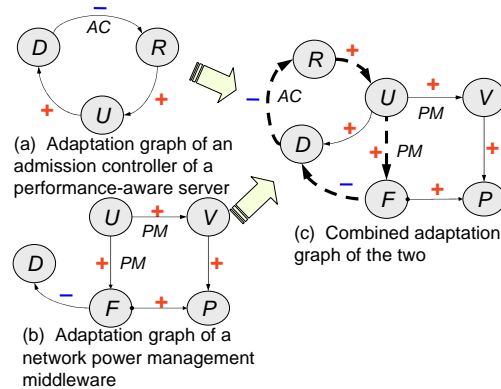


Figure 1: Examples of adaptation graphs and their combination.

middleware thus engages dynamic voltage scaling (DVS) on all machines in the cluster to lower processor voltage, V , and frequency, F , hence reducing power consumption, P , due to the off-peak load condition. This adaptation action can be expressed as $U \rightarrow_{PM}^+ V$ and $U \rightarrow_{PM}^+ F$, where PM stands for power management middleware (i.e., a decrease in link utilization causes the policy to decrease both voltage and frequency which explains the signs on the arcs). In turn, we have $V \rightarrow^+ P$ and $F \rightarrow^+ P$, which says how power consumption changes with voltage and frequency. Finally, we have $F \rightarrow^- D$, since lowering frequency (i.e., slowing down a processor) increases delay and vice versa. Figure 1(b) depicts the adaptation graph for the network power management middleware.

As might be inferred from above, each component or subsystem of a larger system has its own adaptation graph that describes what performance variables this component is affecting and what causality chains (or loops) exist within. The reader might notice that the adaptation graph is a simplified version of the block diagrams used in control theory to represent open-loop and closed-loop control systems [5]. We do not use full-fledged block diagrams (complete with transfer functions of components) because transfer functions require more accurate component modeling and present analysis challenges in the presence of nonlinearities, as would be common in computing systems.

When a system is composed, the adaptation graphs of individual components are coalesced. Fig. 1(c) shows the combined adaptation graph that results when a server described in Fig. 1(a) operates on a LAN cluster managed by the middleware described in Fig. 1(b). This creates a system-wide graph. To check for incompatibilities (adverse interactions), the graph is searched for loops using any common graph traversal algorithm. Loops that are entirely contained within the same subsystem (i.e., loops whose arcs are labeled by the same module name) are safe. Such loops are internal to single components and must have been well-tuned prior to composition. Loops that traverse component boundaries, however, may not have been created by de-

sign. These loops (i.e., loops where some arcs have different module names) may have emerged unintentionally due to composition of the corresponding modules. This insight leads to simple techniques (carried out at composition time) to discover potentially unintended interactions as described below.

2.2. Checks for Potential Incompatibility

We suggest two simple checks for potentially incompatible adaptation policies. These checks operate on the adaptation graph of the composed system. Slightly abusing terminology, we call any closed directed path a *cycle* in the adaptation graph. In other words, a cycle is a directed path which starts and ends with the same node and has distinct links (arcs). However, it may visit a node more than once. The checks for potential incompatibility are as follows:

- *Positive feedback:* A key requirement of adaptation graphs is that all cycles in the graph must be of a negative sign (i.e., the product of all arc signs in the cycle is negative). This requirement stems directly (and can be easily proved) from stability conditions in control theory [5]. A positive cycle indicates a potentially unsafe feedback loop. In other words, a stimulus reinforces itself causing more change in the same direction. Such a cycle may inadvertently develop when multiple adaptation policies are combined. All positive feedback loops in the adaptation graph are thus flagged as potentially unintended interactions.
- *Unstable negative feedback:* Another problematic condition that can be flagged when composing adaptation graphs is potentially unstable negative feedback loops. While adaptation loops that are governed by a single module will tend to be well-tuned and hence stable, emergent adaptation loops that arise from a combination of multiple modules must be explicitly analyzed for stability. All loops where some arcs have different module labels in the adaptation graph are thus flagged as potentially unintended interactions.

Applying the above checks to Fig. 1(c), we flag the cycle $U \xrightarrow{+}_{PM} F, F \xrightarrow{-} D, D \xrightarrow{-}_{AC} R, R \xrightarrow{+} U$ that crosses module boundaries, suggesting it may have not been created by design, and may result in unwanted interactions. The cycle is also positive indicating that unstable interactions may result (between the server admission controller and the network middleware). Interpreting this cycle, the interaction is explained as follows. Starting with the node labeled, U , when the network utilization decreases in the server cluster, the power management middleware will cause individual servers to slow down their processors. This, in turn, will increase the delay experienced by served requests causing the admission controller to accept fewer requests. The reduced accepted number of requests will further decrease the

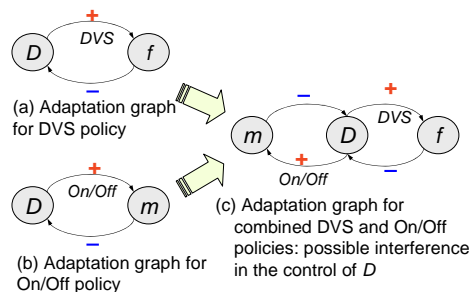


Figure 2: Adaptation graphs for the two policies in the multi-tier server farm and their combination.

load on the network, causing the power management middleware to slow down processors even more. This, in turn, may cause a more significant reduction in admitted requests and a further reduction in network load. This cycle could ultimately bring the server farm to a crawl, indeed an adverse consequence of unintended interaction.

2.3. Application to the Server Farm

Let us now analyze the undesirable interaction in our running case study, between a DVS policy (that controls frequency, f , of machines in a server farm given their delay D^1) and an independently designed machine On/Off policy (that increases the number of machines m in the server farm when the delay is increased and removes machines when the delay is decreased). This interaction is shown in Fig. 2. The DVS policy is described by the adaptation rule $D \xrightarrow{+}_{DVS} f$. Its effect is $f \xrightarrow{-} D$. The On/Off policy is described by the rule $D \xrightarrow{+}_{ON/OFF} m$. Its effect is $m \xrightarrow{-} D$. The adaptation graph for the combined system is shown in Fig 2 (c), in which we see that the loops representing the two policies join in the common D node resulting in a cycle spanning multiple modules. This indicates a possible incompatibility, which may lead to performance degradation or even unstable behavior when the two policies are combined.

Indeed, as flagged by the analysis, the two individually stable negative feedback loops actually exhibit an undesirable interaction that may increase total energy consumption instead of reducing it. The adverse interaction works as follows. When the system is underloaded, the DVS policy reduces the frequency of a processor, increasing system utilization. This will eventually increase the end-to-end delay of the system. Increased delay may cause the (DVS-oblivious) On/Off policy to consider the system to be overloaded, hence turning more machines on to cope with the problem. This may cause the DVS policy to slow down the machines further, leading the On/Off policy to turn more of them on. The energy expended on keeping a larger number of machines on may not necessarily be offset by DVS

¹Observe that changing frequency of a processor also changes the associated core voltage. Therefore, we interchangeably use "changing frequency (level)" and "changing DVS (level)" throughout this paper.

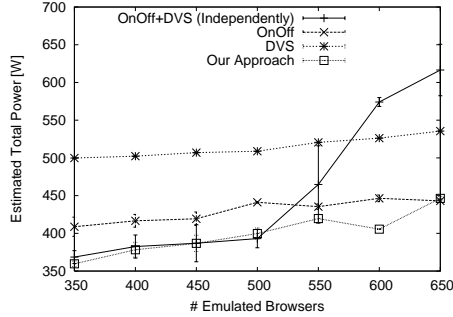


Figure 3: Comparison of estimated total system power consumption for different adaptive policies in the Web server case study.

savings. Hence, the resulting cycle may lead to poor energy performance, even despite the fact that both the DVS and On/Off policies have the same energy saving goal.

Fig. 3 shows experimental results from our three-tier Web server farm testbed. We only show the results when the system is not overloaded. The setup and power consumption estimation will be described in detail in Section 4. Four different energy saving configurations are compared: the On/Off policy, the DVS policy, the combination of On/Off + DVS (exhibiting adverse interaction) and finally our proposed optimal policy that we will discuss in the next section of this paper. It is clearly demonstrated that the combined On/Off + DVS policy spends much more energy than all other policies, when the number of emulate browser(EB)s is larger than 500. In contrast, our optimal policy shows best performance. When the workload is too small (e.g. # of EBs is less than 500), the unmodified combined policy doesn't show excessive energy consumption due to the adverse interaction, because the end-to-end delay remains below the threshold so that (DVS-oblivious) On/Off policy would not determine the system as overloaded. This experiment, performed on an actual server farm, indicates that the adverse interaction problem is not hypothetical but is something measurable in practice. Indeed, as the figure shows, combining two good energy saving policies may yield savings that are worse than with either policy in isolation! Next, we describe our methodology for reconciling adaptive policies with such adverse interactions.

3. Design Methodology for Integrating Adaptive Policies

Having identified the possible adverse interactions that may exist between a set of adaptive policies using adaptation graph analysis, a need for re-design becomes apparent. We now present our (re-)design methodology used to solve the incompatibility problem. One way to achieve co-adaptability when composing different adaptive policies is to break the adverse interactions by design. For example, a straightforward solution to the problem demonstrated in Figure 3 is to remove the DVS module, hence breaking

the undesirable interaction loop. This solution, however, may not be optimal. Below, we present a general solution methodology that co-designs the adaptive policies optimally and hence removes their adverse interactions. Observe the co-design does not necessarily undo modularity for the following two reasons.

First, most well-engineered systems feature separation of policy from mechanism. In our solution the mechanism implementations themselves will remain disjoint. Observe that most of the software complexity usually comes from the mechanism (and not policy) implementation. For example, implementing priority-based scheduling of processes in a kernel is more complex than setting process priorities. Hence, co-adaptation does not tangibly affect implementation complexity.

Second, our solution relies on implementing a co-adaptation module that simply returns jointly optimized knob settings. This module can be called independently by each of the original adaptation modules when it is time to compute that module's knob setting. The module will need only the setting for its own knob. Hence, much of the original policy code (such as code to measure relevant system performance, store state, or implement periodic invocation at a given frequency) remains the same, except that the act of computing an output from current state is replaced by a new function call. Below, we describe the design of the co-adaptation module.

3.1. General Solution Methodology

Our solution methodology to joint adaptation is based on a constrained optimization formulation, in which the value function represents the combined objective and the decision variables represent the application-level actuator "knobs" available. Formally defining the problem as an optimization problem helps in systematically finding the best composition strategy of the adaptive policies.

Our solution methodology is divided into four steps, inspired by recent work on network utility maximization and layering as optimization decomposition. A great recent survey is found in [6]. This work generally attempts to provide analytic foundations for decomposing network software into a layered (i.e., modular), composable protocol stack architecture. We adapt this general line of thought to modularizing software architecture and designing composable performance-sensitive systems.

1. *Casting the objective:* The first step is to find a common objective function to optimize for the underlying application among different adaptation policies. This step is straightforward to apply to co-design adaptive policies that optimize the same objective (e.g., the On/Off and DVS policies in the server farm). For policies with different objectives, a common value function should be obtained to relate the individual module objectives to a common higher-level

system objective (e.g., to optimize revenue or maximize total utility).

2. *Formulating the optimization problem:* The second step is to formally express the optimization problem given the arrived-upon common objective function. Optimization is performed with respect to the available adaptation knobs in the modules to be combined. The problem is subject to two types of constraints; namely, (i) resource constraints, and (ii) performance specifications. The goal of the adaptation policies is to optimize the objective function subject to these constraints. Suppose there are a total of n different adaptation policies. For each adaptation policy i , a corresponding set of adaptation knobs is denoted as x_i , where $i = 1, \dots, n$. For different policies, the set of adaptation knobs may overlap. With the common objective function obtained from Step 1, we can formulate a constrained optimization problem generally in the following form:

$$\begin{aligned} & \min_{x_1, \dots, x_n} && f(x_1, \dots, x_n) \\ & \text{subject to} && g_j(x_1, \dots, x_n) \leq 0, \quad j = 1, \dots, m, \end{aligned} \quad (1)$$

where f is the common objective function²; $g_j(\cdot)$, $j = 1, \dots, m$ are the resource and performance constraints related to the application. Introducing Lagrange multipliers ν_1, \dots, ν_m the Lagrangian of the problem is given as:

$$\begin{aligned} L(x_1, \dots, x_n, \nu_1, \dots, \nu_m) = & f(x_1, \dots, x_n) + \\ & \nu_1 g_1(x_1, \dots, x_n) + \\ & \dots + \\ & \nu_m g_m(x_1, \dots, x_n) \end{aligned} \quad (2)$$

3. *Derivation of necessary conditions:* Having defined the optimization problem, the third step in the general methodology is to derive conditions for optimality. Since it is hard to find perfect models for inherently stochastic computing systems, model inaccuracies (such as inaccuracies in estimating actual task computation times, their exact energy consumption, or end-to-end delay in practical systems) are likely to render the expressions for functions $f(\cdot)$ and $g_j(\cdot)$ above inaccurate. Hence, we would like to combine optimization with feedback control to compensate for such inaccuracies.

Our approach for mitigating modeling inaccuracies in the problem formulation is to derive only *approximate necessary conditions* for optimality, instead of exact necessary and sufficient conditions. A series of feedback loops is then used to (i) monitor deviations from the necessary conditions, and (ii) fix them in the direction that increases utility. This will considerably increase the chances to reach to

²In this case f represents a notion of cost to be minimized. Alternatively, it could represent a notion of utility to be maximized.

the optimal point by enlarging the searching space. In contrast, solving the exact and necessary condition might not allow to get to the optimal point due to the inevitable model inaccuracies. This approach is explained below.

The necessary conditions of optimality are derived by relaxing the original problem (i.e., where knob settings are discrete) into a continuous problem (where knob setting are real numbers and functions $g(\cdot)$ and $f(\cdot)$ are differentiable), then using the Karush-Kuhn-Tucker (KKT) optimality conditions [3], $\forall i : 1, \dots, n$:

$$\frac{\partial f(x_1, \dots, x_n)}{\partial x_i} + \sum_{j=1}^m \nu_j \frac{\partial g_j(x_1, \dots, x_n)}{\partial x_i} = 0 \quad (3)$$

Let us call the left-hand-side, Γ_{x_i} . Observe that, we have the necessary condition:

$$\Gamma_{x_1} = \dots = \Gamma_{x_n} \quad (4)$$

We then use a (hill climbing) measurement-based feedback control approach to empirically find the maximum utility point on the locus that satisfies Equation (4). Our feedback control approach is described next. We find it useful for the discussion below to also define the average $\Gamma_x = (\Gamma_{x_1} + \dots + \Gamma_{x_n})/n$.

4. *Feedback control:* The purpose of feedback control is to find knob values x_i such that the condition in Equation (4) is satisfied. Conceptually, when some values of Γ_{x_i} are not equal, two directions are possible for fixing the deviation in the condition. One is for modules with smaller values of Γ_{x_i} to change their knobs x_i to catch up with larger ones. The other is for those with larger values of Γ_{x_i} to change their knobs to catch up with smaller ones. Moreover, more than one knob may be adjusted together. In the spirit of hill climbing, we take the combination that maximizes increase in utility. Hence, we define the control error as $\Gamma_x - \Gamma_{x_i}$ and find the set of neighboring points (to the current x_i vector) that reduces the error and involves a maximum increase in utility. We call this technique *co-adaptation*. The algorithm will dynamically guide the system towards a better configuration.

We will next show how this general solution methodology can be applied to the multi-tier Web server farm case study.

3.2. Application to the Server Farm

1. *Casting the objective:* The first step of using the optimization methodology is to formulate a common optimization objective. In this case, the objective of both the On/Off and DVS components is to minimize energy consumption. Hence, energy is the common cost function.

2. *Formulating the optimization problem:* The decision variables in the optimization problem are the tuning knobs

for each individual adaptation policy, namely the frequency levels of each machine (for the DVS policy) and the number of active machines at each tier (for the On/Off policy). Since our methodology ultimately involves feedback control, it is robust to modeling errors. Hence, we use two approximations in the formulation of the optimization problem. First, we use a simple queuing-theoretic model for each server machine to predict delay. Second, the optimization problem is formulated as a non-negative real-valued relaxation of the true problem where the the number of machines at each tier is restricted to be non-negative integers.

We assume that the power consumption is equal for all machines at each tier. Furthermore, the machines are load balanced in steady-state, such that each machine at each tier has the same utilization, the same frequency, and the same traffic arrival rate.

We use the steady-state results of the M/M/1 queuing model [15] to model each machine in the server farm. In this model, the system utilization, U , is expressed as λ/μ , given the arrival rate λ of the traffic and the service rate μ of the server. Then, the utilization of a machine at tier i , U_i , is computed as

$$U_i = \frac{\lambda}{\mu} = \frac{\lambda_i/m_i}{f_i} = \frac{\lambda_i}{m_i f_i}, \quad (5)$$

where f_i is the per-machine frequency, λ_i is the total arrival rate to tier i , and m_i is the number of machines at tier i . The arrival rate, λ_i , is expressed in CPU cycles/second.

We approximate power consumption P_i by the following function of CPU frequency f_i for each machine at tier i :

$$P_i(f_i) = A_i \cdot f_i^p + B_i, \quad (6)$$

where A_i and B_i are positive constants. In realistic systems p varies between 2.5 and 3 [9]. A_i , B_i , and p can be obtained by curve fitting against empirical measurements when profiling the system off-line.

Using Equation (5) and by substitution into (6) we get

$$P_i(U_i, m_i) = A_i \cdot \left(\frac{\lambda_i}{U_i m_i} \right)^p + B_i = \frac{A_i \lambda_i^p}{U_i^p m_i^p} + B_i. \quad (7)$$

The total power consumption can be obtained by summing over all the N tiers as

$$P_{tot}(U_i, m_i) = \sum_{i=1}^N m_i \cdot P_i(U_i, m_i). \quad (8)$$

We want to minimize the total server power consumptions subject to two functional constraints. The first constraint is that the total end-to-end delay should be less than some end-to-end delay bound, L . In the M/M/1 queuing model, the total waiting time in the system, D , is expressed as $\frac{1}{\mu - \lambda}$, given the service rate μ and the arrival rate λ [15]. When each machine at tier i has service rate f_i and arrival rate λ_i/m_i , the average delay D_i^{CPU} is given as

$$D_i^{CPU} = \frac{1}{f_i - \frac{\lambda_i}{m_i}} = \frac{1/f_i}{1 - \frac{\lambda_i}{m_i f_i}} = \frac{m_i}{\lambda_i} \cdot \frac{\frac{\lambda_i}{m_i f_i}}{1 - \frac{\lambda_i}{m_i f_i}} = \frac{m_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \quad (9)$$

While the M/M/1-based delay model of Equation (9) does not model the delay relation exactly, the lack of precision of this simplified model will again be compensated by the presence of feedback control in our methodology. Using Equation (9), the delay constraint can be written as

$$\sum_{i=1}^N \frac{m_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \leq K, \quad (10)$$

where $K = L - \sum_{i=1}^N D_i^{non-CPU}$, where $D_i^{non-CPU}$ lumps non-CPU-related delays on stage i .

The second constraint is on the total number of machines M in the farm: $\sum_{i=1}^N m_i \leq M$.

For a 3-tier server farm ($N = 3$) and using $p = 3$, the constrained minimization problem can now be formulated as:

$$\begin{aligned} \min_{U_i \geq 0, m_i \geq 0} \quad & P_{tot}(U_i, m_i) = \sum_{i=1}^3 m_i \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) \\ \text{subject to} \quad & \sum_{i=1}^3 \frac{m_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \leq K, \\ & \sum_{i=1}^3 m_i \leq M \end{aligned} \quad (11)$$

3. Derivation of necessary conditions: To derive necessary conditions, let $x = [U_1 \ U_2 \ U_3 \ m_1 \ m_2 \ m_3]^T$ be the vector of decision variables. Introducing the Lagrange multipliers $\nu_1, \nu_2 \geq 0$, we can write the Lagrangian function as:

$$\begin{aligned} L(x, \nu_1, \nu_2) = & \sum_{i=1}^3 m_i \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) + \\ & + \nu_1 \cdot \left(\sum_{i=1}^3 \left(\frac{m_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \right) - K \right) + \nu_2 \cdot \left(\sum_{i=1}^3 (m_i) - M \right). \end{aligned} \quad (12)$$

The Karush-Kuhn-Tucker (KKT) conditions [3] associated with the optimization problem are:

$$\begin{aligned} \nabla_x L(x, \nu_1, \nu_2) &= 0, \\ \nu_1 \cdot \left(\sum_{i=1}^3 \left(\frac{m_i}{\lambda_i} \cdot \frac{U_i}{1 - U_i} \right) - K \right) &= 0, \\ \nu_2 \cdot \left(\sum_{i=1}^3 (m_i) - M \right) &= 0. \end{aligned} \quad (13)$$

The last two equations are the complementary slackness conditions, which state that for all inactive constraints the corresponding Lagrange multiplier is zero. For our problem, the KKT conditions give:

$$\begin{aligned} \frac{\partial L}{\partial U_i} &= -\frac{nA_i\lambda_i^3}{m_i^2U_i^4} + \frac{\nu_1 m_i}{\lambda_i(1-U_i)^2} = 0 \quad \forall i, \\ \frac{\partial L}{\partial m_i} &= -\frac{2A_i\lambda_i^3}{m_i^3U_i^3} + B_i + \frac{\nu_1}{\lambda_i} \cdot \frac{U_i}{1-U_i} + \nu_2 = 0 \quad \forall i, \\ \nu_1 \cdot \left(\sum_{i=1}^3 \left(\frac{m_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} \right) - K \right) &= 0, \\ \nu_2 \cdot \left(\sum_{i=1}^3 (m_i) - M \right) &= 0. \end{aligned} \quad (14)$$

Solving for ν_1 and ν_2 then substituting in the first two sets of equations above, we get after some rearranging:

$$\frac{\lambda_1(1-U_1)^2}{U_1(3U_1-1)} = \frac{\lambda_2(1-U_2)^2}{U_2(3U_2-1)} = \frac{\lambda_3(1-U_3)^2}{U_3(3U_3-1)}, \quad (15)$$

$$\frac{\lambda_1^3(3U_1-1)}{m_1^3U_1^3} = \frac{\lambda_2^3(3U_2-1)}{m_2^3U_2^3} = \frac{\lambda_3^3(3U_3-1)}{m_3^3U_3^3}. \quad (16)$$

The two conditions can be combined to yield a single relation

$$\frac{\lambda_1^4(1-U_1)^2}{m_1^3U_1^4} = \frac{\lambda_2^4(1-U_2)^2}{m_2^3U_2^4} = \frac{\lambda_3^4(1-U_3)^2}{m_3^3U_3^4}. \quad (17)$$

To simplify the notations, we will use $\Gamma(m_i, U_i)$ to denote $\frac{\lambda_i^4(1-U_i)^2}{m_i^3U_i^4}$ in the following discussions. Then the necessary condition for optimality is expressed as

$$\Gamma(m_1, U_1) = \Gamma(m_2, U_2) = \Gamma(m_3, U_3). \quad (18)$$

4. Feedback control: It can be easily seen from the necessary condition that, assuming stable changes in λ_i and m_i , the value of $\Gamma(m_i, U_i)$ will increase as U_i decreases. On the other hand, $\Gamma(m_i, U_i)$ will decrease if U_i increases. From this, we can deduce that a smaller value for $\Gamma(m_i, U_i)$ indicates that tier i is *overloaded* and, similarly, a larger value for $\Gamma(m_i, U_i)$ indicates that tier i is *underloaded*. Based on this observation, we will design a feedback loop corresponding to the fourth step in our general solution methodology. For example, to maximize utility when the system is overloaded (i.e., experiences large delay), the utilization and the number of machines are adjusted in a direction to enforce Equation (18) while incurring the least energy consumption.

More specifically, at each sampling interval k , every machine measures its average CPU utilization and frequency. These values are broadcast in the farm. The per-tier average utilization U_i and frequency f_i at tier i can thus be locally computed on each machine. The average end-to-end

delay, D , is computed by the first-tier machines and broadcast. The number of machines m_i in each tier i is estimated by counting the number of participating machines. λ_i is estimated from the relation $\lambda_i = U_i f_i m_i$ by applying an exponentially weighted moving average (EWMA) filter (λ_i is estimated since it is not feasible to measure the request rate in CPU cycles/second as in the optimization formulation) as $\lambda(k) = \alpha\lambda(k-1) + (1-\alpha)U_i f_i m_i$. Given this measurement, the algorithm at each machine computes the function $\Gamma(m_i, U_i)$ for all tiers. The system is determined to be overloaded or underloaded based on the average delay D and the reference average delay R . If D is within a region around the set-point, $[R - W_d, R + W_d]$ where $0 < W_d < R$, the algorithm takes no action.

To implement greedy hill climbing, when the system is *overloaded* ($D > R + W_d$), the most overloaded tier is determined first and more computing power is provided to it. The tier s with $\min_i \Gamma(m_i, U_i)$ value is selected, (since the most overloaded tier should have the lowest $\Gamma(m_i, U_i)$). This value must be increased to the average of the other tiers, denoted by Γ_{target} . We choose all the (m_s, f_s) pairs that contain $\Gamma(m_s, U_s)$ within $[\Gamma_{target} - W, \Gamma_{target} + W]$ from the realizable discrete values of m_s and f_s . When calculating $\Gamma(m_s, U_s)$ with a (m_s, f_s) pair, the relation between utilization and frequency, $U_s = \frac{\lambda_s/m_s}{f_s}$ is used. These pairs correspond to possible solutions to the energy minimization problem. We then choose the one with the lowest total energy by evaluating the power objective function of Equation (8). This procedure realizes the hill climbing algorithm. This computation occurs in parallel on all machines. The list of machines in each tier is ordered. The first m_s machines in that list are awakened (if they are not on already), while the others remain asleep. All machines are set to operate at frequency f_s . If the system is *underloaded* ($D < R - W_d$), the tier with the maximum $\Gamma(m_i, U_i)$ value (the most underloaded tier) is selected and adjusted approximately to the average of the other two tiers in the same way as described above.

4. Implementation of Energy Optimization Framework for Web Server Farms

In this Section, we describe the implementation of our autonomous energy optimization framework for multi-tier Web servers as an example of how the proposed co-adaption methodology can be applied to real large-scale software systems.

4.1. Testbed Setup

We first give an overview of our test-bed setup. Our prototype three-tier server farm is composed of a total of 17 machines: 2 fast machines with Intel Pentium IV 3GHZ CPU and 2GB of RAM; 15 slow machines with Intel Celeron 2.53GHZ CPU and 512MB of RAM. All ma-

chines are equipped with Redhat Fedora Core 4 Linux. The 15 slow machines are used for the first-, second-, and third-tiers with 5 machines per tier. The first-tier machines run Apache 2.0 with *mod_jk* support, the second-tier machines run Apache Tomcat 5.1 and the third-tier machines run the MySQL 5.0 database server. One fast machine is used as load balancer for the first tier with Apache 2.3 *mod_proxy_balancer*. One more fast machine is used to balance the database workload. We use Sequoia [7], a JDBC-based database clustering solution, to replicate database servers. In addition to the 17 machines, two more machines with Intel Pentium IV 3GHZ CPU and 2GB of RAM are used to measure power consumption.

We have implemented a three-tier server farm system based on the TPC-W specification [20] on this platform. TPC-W is an industry standard e-Commerce benchmark tool for evaluating the performance of e-commerce Web sites.

4.2. Distributed Feedback Control Algorithms

We have implemented three distributed feedback algorithms in our multi-tier Web server farm test-bed. The Feedback DVS algorithm only exploits the DVS policy, whereas the Feedback On/Off algorithm uses the On/Off policy. The Feedback On/Off & DVS algorithm is our optimal approach based on the proposed co-adaptation design methodology reconciling the incompatibility of the DVS policy and the On/Off policy. These feedback algorithms use delay measurements to dynamically adjust the utilization and the number of machines to move the system state to the optimal state as directed by the derived necessary conditions. All of these feedback algorithms make control decisions locally based on a globally obtained system snapshot, which includes CPU utilization, request rate and end-to-end delay, as described in Section 3.

When implementing the Feedback On/Off & DVS algorithm, we still maintain the two adaptation modules separately and introduce a new module used by both On/Off and DVS modules to implement our co-adaptation policy. The co-adaptation module computes the optimal combination of machine assignment and frequency, an (m, f) pair, given the current system state. When one of the adaptive policies needs to adjust its value as it observes excessive delay violations or the farm experiences low utilization, it executes the co-adaptation module. For example, when the On/Off module finds that it needs to add new machines due to high utilization of the system, it executes the co-adaptation module, which will produce the optimal (m, f) , but then only uses m to adjust the assignment of machines. Similarly, when the DVS policy needs to adjust its value, it runs the co-adaptation module to get only f (and hence the associated voltage). In this way, each individual adaptive policy

runs in a separate module without violating modularity but the adaptation is done by the common co-adaptation module, which gives the optimal value according to the current system state.

4.3. Power Measurement

We estimate the power consumption of the server farm in two ways. The first method we use is to measure power consumption with external power meters. We use AC power meters to measure power consumption in the server farm. Power meters are connected through a serial line to a server machine, periodically recording AC power every second.

We also estimate power consumption based on frequency measurements of the processors. This allows us to evaluate our algorithms on system settings where the effect of DVS on power savings are different. Specifically, the current frequency is measured every sampling interval and we then estimate the achieved power consumption based on the measured frequency.

We first average the power consumption when the maximum frequency level is applied against different offered workload and use it as the base power consumption. Let us call this value $\overline{P_{max}}$. We use the relationship between power and frequency from Equation (6) and set the A_i - and B_i -values such that $A_1 = A_2 = A_3 = A$ and $B_1 = B_2 = B_3 = B$, since all the server machines are assumed identical across all tiers. B represents the fixed energy consumption regardless of the current frequency setting and A is a coefficient for calculating the effect of frequency changes (hence the core voltage of the processor). We can then set B as a scaling of the maximum power as $B = \delta \overline{P_{max}}$ where $0 < \delta < 1$. In this way, δ characterizes how much DVS affects the overall power consumption. A larger δ means that the effect of DVS is less and a smaller δ means that energy consumption is more dominated by DVS. Then we may calculate A as $A = \frac{\overline{P_{max}} - B}{f_{max}^p}$. Finally, the estimated power \widehat{P}_f with frequency f is

$$\widehat{P}_f = Af^p + B = \overline{P_{max}} \left[(1 - \delta) \left(\frac{f}{f_{max}} \right)^p + \delta \right]. \quad (19)$$

With given $\overline{P_{max}}$ and the measurement of the current frequency, f , we can calculate the estimated power consumption from this equation. We will use this equation in the following evaluation.

5. Evaluation

In this Section, we demonstrate that co-adaptation design improves system performance considerably by evaluating the energy minimization schemes in server farms. To this end, we evaluate five different energy saving approaches: a baseline, the Linux On-demand governor, and our three control algorithms (the Feedback DVS, the Feedback On/Off, and the Feedback On/Off & DVS). For the

baseline, we set the CPU frequency to the maximum on all machines. The Linux On-demand [16] is a dynamic in-kernel CPU frequency governor that changes CPU frequency levels based on CPU utilization. When the CPU utilization is over 80%, it steps up to the next higher frequency level and if it is below 20%, it reduces the frequency level. Note that we don't compare with other existing energy saving mechanisms because our goal is to show that the system performance can be considerably improved by resolving possible conflicts between multiple adaptive policies rather than to claim our energy saving approach is the best.

For each test run, 2500 seconds of TPC-W workload are applied, with a 300-second ramp-up period, a 2000-second measurement interval, and finally a 200-second ramp-down period. The TPC-W benchmark generates requests by starting a number of emulated browsers (EB). To evaluate the energy consumption of the system under different workloads, we applied different numbers of EBs: 0, 100, 200, 300, 400, 500, and 600. The user think time was set to 1.0 seconds. We used 1.0 seconds as the (soft real-time) latency constraint with a maximum allowed miss ratio of 0.1. We used 450 ms as the delay set-point for all experiments. The delay set-point is computed such that if the average delay is kept around or below it, the miss ratio of the latency constraint is maintained at or below 0.1, assuming that the end-to-end delay follows an exponential distribution. This is the same distribution assumed by the M/M/1 delay model in Equation (9).

5.1. Comparison of Total Power

To compare power consumption on different system settings, we estimated power consumption based on the measurement of frequency changes using Equation (19). We have plotted the estimated total power of the various control algorithms in Fig. 4. The figure shows power consumptions of the algorithms on three different setups: $\delta = 0.3, 0.5,$ and 0.8 .

- A larger δ corresponds to a larger B in Equation (19), which means that the effect of DVS is low and power consumption is relatively constant. With larger δ , one could save more power by turning more machines off rather than decreasing DVS levels of the CPUs.
- A smaller δ corresponds to a smaller B in Equation (19). In this case, total power is dominated by the term $A \cdot f^p$ rather than B . Decreasing DVS levels can save more energy compared to when B is large.

By estimating power consumption with different B s, we can see how well the proposed algorithms perform in systems with different DVS capabilities. The power consumption of the baseline and Feedback On/Off is measured with AC power meters, since they do not use DVS.

As conjectured, the system deployed with the baseline algorithm consumes the most power for all δ . Fig. 4(a) shows that for $\delta = 0.8$, where DVS is less effective, the Feedback On/Off & DVS algorithm gives the best performance among all. It can save about 60% of power when workload is low and 30% for the highest workload in the experiment. The next best algorithm is the Feedback On/Off. It saves 43% when workload is low and 27% for the highest workload. The Feedback DVS slightly outperforms the Linux On-demand algorithm although they both save as much as 20% in total power depending on workload. The reason for the better performance for the Feedback DVS algorithm is because it shares the global view of the system so that it can save more power up to the point that the deadline is not violated, whereas the Linux On-demand governor has limited such knowledge.

The Feedback On/Off & DVS algorithm also shows the best performance in Fig. 4(b), where $\delta = 0.5$. Since δ is decreased, DVS plays a larger role compared to when $\delta = 0.8$. The Feedback DVS, the Feedback On/Off and the Linux On-demand algorithms show similar performance in this case. It is interesting to see that the Feedback On/Off actually performs worse than the Feedback DVS and Linux On-demand.

As seen in Fig. 4(c), the Feedback On/Off & DVS algorithm still outperforms all other algorithms when $\delta = 0.3$. It saves as much as 84% in total power when workload is lowest and 55% when workload is highest. However, the Feedback On/Off and Linux On-demand save comparable power to the Feedback On/Off & DVS. They save more than 60%.

5.2. Comparison of Other Performance Metrics

We next present results from other performance metrics of the five algorithms. All metrics are measured from our server farm testbed. Note that, except power consumption, the performance results shown here are not affected by power saving characteristics of the system (e.g. δ values) and thus we show only one result for each performance measurement. Fig. 5(a) depicts the average delay of the five control algorithms. As described above, the reference delay is set to 450 ms and the latency bound is 1 sec. As the workload increases, it is observed that the average delay of all algorithms increases. However, except for the Linux On-demand, all other algorithms keep the delay below the set-point. This proves that the proposed distributed algorithms work well by keeping the average delay under the set-point while providing considerable power savings compared to the baseline. We also plotted the deadline miss ratio in Fig. 5(b). For all algorithms, the miss ratio is less than 7%, which is less than the maximum tolerable miss ratio of 0.1 (10%).

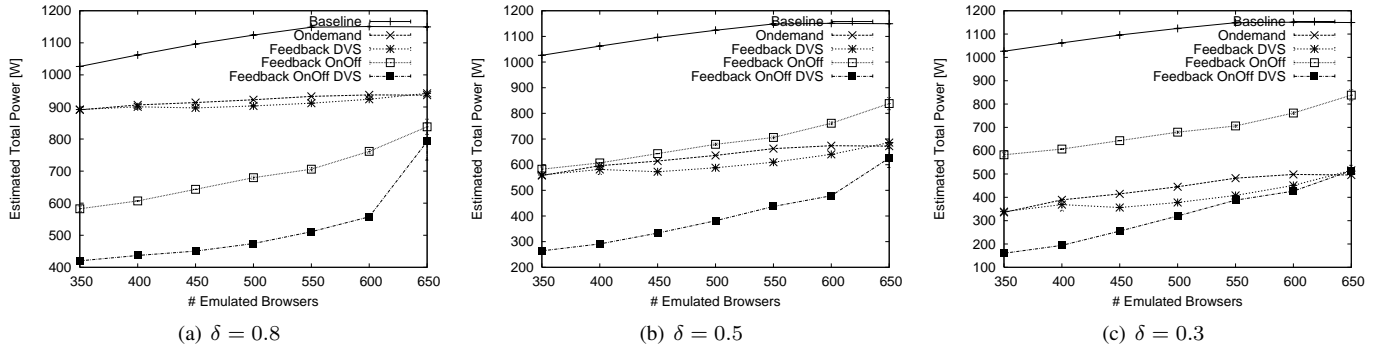


Figure 4: Estimated Total System Power Consumption.

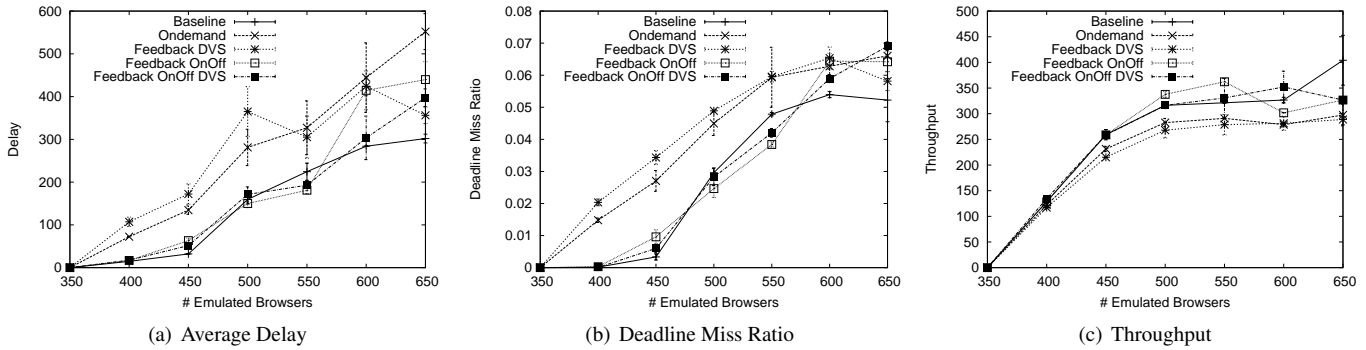


Figure 5: Other Metrics: Average Delay, Deadline Miss Ratio, and Throughput

It should be noted that two of our algorithms (the Feedback On/Off, and the Feedback On/Off & DVS) experience smaller deadline miss ratios and average delay (except at the highest workload) than the Feedback DVS algorithm and Linux On-demand algorithms. One possible reason is that by turning off machines, the overhead for distributing load to the next tier and synchronizing session information becomes smaller, whereas the Feedback DVS algorithm and Linux On-demand algorithms use all the available machines to serve requests (hence with more overhead). For example, we use Tomcat5’s clustering scheme to synchronize session information across tier 2. Therefore, the number of messages exchanged between tier 2 servers increases as the workload increases. Hence, if there is a smaller number of machines participating in the server farm (tier 2), the overhead reduces considerably. Similarly, the Apache *mod.jk* module, which distributes requests to tier 2 servers, keeps track of how many requests are served by each Tomcat server to properly apply the weighted load balancing algorithm.

The average number of machines for tier 1 and tier 2 are plotted in Fig. 6(a) and 6(b) to confirm the above reasoning. Only a small number of machines were assigned to tier 1 and tier 2 when the workload is low. This also explains why sometimes the Feedback On/Off & DVS and Feedback On/Off algorithms perform better than the baseline when workload is not the highest, since the baseline always keeps

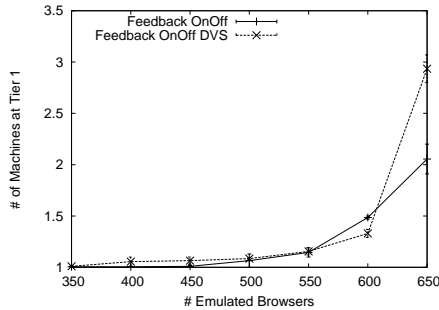
all machines alive.

Observe that the Feedback On/Off & DVS algorithm generally assigns more machines than the Feedback On/Off algorithm because the Feedback On/Off & DVS algorithm tries to find an optimal combination of machine assignments and DVS levels, while the Feedback On/Off only changes the number of machines with the highest DVS level.

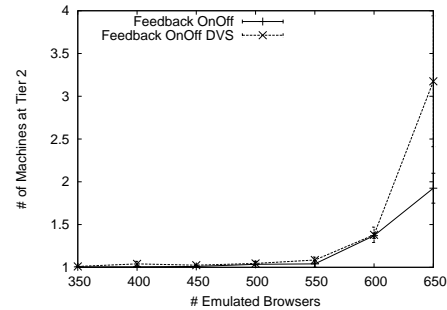
Fig. 5(c) shows the throughput for the five algorithms. As we can see, none of our three control algorithms experience severe throughput penalty by saving power. When the number of EBs is 300 and 400, the Feedback On/Off & DVS and the Feedback On/Off algorithms show higher throughput than the baseline. This comes from the same reason as the case for miss ratio and average delay.

6. Conclusions and Future Work

This paper has presented methods for composition of adaptive components in complex software systems. A mechanism, called *adaptation graph analysis*, was introduced to identify potential incompatibilities between multiple adaptation policies and a general design methodology was presented to resolve the conflicts between interacting adaptive components. The usefulness of the proposed approach was demonstrated by an application to energy minimization in multi-tier server farms. Using adaptation graph analysis, we first identified incompatibilities between the *On/Off* and *dynamic voltage scaling* policies



(a) Average Number of Machines at Tier 1



(b) Average Number of Machines at Tier 2

Figure 6: Average Number of Machines at Tier 1 and 2

used in the server farm. A constrained energy minimization formulation was then used to solve the co-adaptation problem to reduce the adverse effects between the two adaptive mechanisms. Experimental results demonstrate that the energy minimization framework based on the proposed design methodology for co-adaptation can save considerable amounts of energy compared to the baseline and Linux On-demand policies, while maintaining the latency within acceptable bounds. Future work involves automating the design methodology with AADL-based tools and using it in other application areas.

References

- [1] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. Qos negotiation in real-time systems and its application to automated flight control. *IEEE Trans. Comput.*, 49(11):1170–1183, 2000.
- [2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.
- [3] D. P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1995.
- [4] R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–74, 2004.
- [5] C.-T. Chen. *Linear System Theory and Design*. Oxford University Press, 1998.
- [6] M. Chiang, S. Low, A. Calderbank, and J. Doyle. Layering as optimization decomposition: A mathematical theory of network architectures. *Proceedings of the IEEE*, 95(1):255–312, 2007.
- [7] Continuent.org. The Sequoia Project.
- [8] B. Dasarathy, S. Gadgil, R. Vaidyanathan, K. Parmeswaran, B. Coan, M. Conarty, and V. Bhanot. Network qos assurance in a multi-layer adaptive resource management scheme for mission-critical applications using the corba middleware framework. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 246–255, Washington, DC, USA, 2005. IEEE Computer Society.
- [9] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *PACS*, pages 179–196, 2002.
- [10] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
- [11] G. Holland, N. Vaidya, and P. Bahl. A rate-adaptive mac protocol for multi-hop wireless networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 236–251, New York, NY, USA, 2001. ACM Press.
- [12] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56(4):444–458, 2007.
- [13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [14] B. Khargharia, S. Hariri, and M. S. Yousif. Autonomic power and performance management for computing systems. In *IEEE International Conference Autonomic Computing*. IEEE Computer Society, 2006.
- [15] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.
- [16] V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proc. of the Linux Symposium*, volume 2, 2006.
- [17] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM Press.
- [18] L. Sha, X. Liu, Y. Lu, and T. Abdelzaher. Queueing model based network server performance control. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 81, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware qos management in web servers. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 63, Washington, DC, USA, 2003. IEEE Computer Society.
- [20] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce).
- [21] M. Wang, N. Kandasamy, A. Guez, and M. Kam. Adaptive performance control of computing systems via distributed cooperative control: Application to power management in computing clusters. In *IEEE International Conference Autonomic Computing*. IEEE Computer Society, 2006.
- [22] Y. Wei, S. H. Son, J. A. Stankovic, and K. D. Kang. Qos management in replicated real time databases. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 86, Washington, DC, USA, 2003. IEEE Computer Society.
- [23] S. H. Y. Wong, S. Lu, H. Yang, and V. Bharghavan. Robust rate adaptation for 802.11 wireless networks. In *MobiCom '06: Proceedings of the 12th annual international conference on Mobile computing and networking*, pages 146–157, New York, NY, USA, 2006. ACM Press.