Chapter 7
Introduction to Control Theory And Its Application to Computing Systems

Tarek Abdelzaher, Yixin Diao, Joseph L. Hellerstein,
Chenyang Lu, and Xiaoyun Zhu

Abstract Feedback control is central to managing computing systems and data networks. Unfortunately, computing practitioners typically approach the design of feedback control in an ad hoc manner. Control theory provides a systematic approach to designing feedback loops that are stable in that they avoid wild oscillations, accurate in that they achieve objectives such as target response times for service level management, and settle quickly to their steady state values. This paper provides an introduction to control theory for computing practitioners with an emphasis on applications in the areas of database systems, real-time systems, virtualized servers, and power management.

7.1 Introduction

Feedback control is central to managing computing systems and networks. For example, feedback (or closed loop systems) is employed to achieve response time objectives by taking resource actions such as adjusting scheduling priorities, memory allocations, and network bandwidth allocations. Unfortunately, computing practitioners typically employ an ad hoc approach to the design of feedback control, often

Tarek Abdelzaher
Dept. of Comp. Sci., University of Illinois, Urbana-Champaign, IL, e-mail: zaher@cs.uiuc.edu

Yixin Diao
IBM T. J. Watson Research Center, Hawthorne, NY, e-mail: diao@us.ibm.com

Joseph L. Hellerstein
Developer Division, Microsoft Corp, Redmond, WA, e-mail: joehe@microsoft.com

Chenyang Lu
Dept. of Comp. Sci. and Eng., Washington University, St. Louis, MO, e-mail: lu@cse.wustl.edu

Xiaoyun Zhu
Hewlett Packard Laboratories, Hewlett Packard Corp., Palo Alto, CA, e-mail: xiaoyun.zhu@hp.com

with undesirable results such as large oscillations or slow adaptation to changes in workloads.

In other mechanical, electrical, aeronautical and other engineering disciplines, control theory is used to analyze and design feedback loops. Control theory provides a systematic approach to designing closed loop systems that are stable in that they avoid wild oscillations, are accurate in that they achieve the desired outputs (e.g., response time objectives), and settle quickly to steady state values (e.g., to adjust to workload dynamics). Recently, control theory has been used in the design of many aspects of computing. For example, in data networks control theory has been applied to flow control [18] and to the design of new versions of TCP/IP [17].

This paper provides an introduction to control theory for computer scientists with an emphasis on applications. Section 2 discusses key concepts and fundamental results in control theory. Section 3 describes how control theory has been applied to self-tuning memory management in IBM's DB2 Universal Data Base Management System. Section 4 addresses the use of model-predictive control in distributed real-time systems. Section 5 discusses automated workload management in virtualized data centers. Section 6 details the use of control theory for managing power and performance in data centers. Our conclusions and research challenges are presented in Section 7.

7.2 Control Theory Fundamentals

This section provides a brief overview of control theory for computer scientists with little background in the area. The focus is on key concepts and fundamental results.

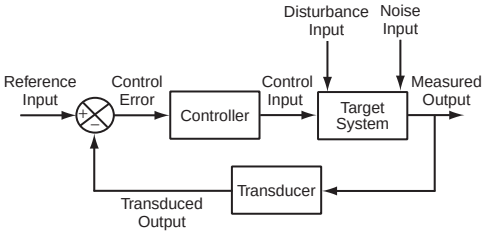


Fig. 7.1 *Block diagram of a feedback control system.*

Karl Astrom, one of the most prolific contributors to control theory, states that the "magic of feedback" is that it can create a system that performs well from components that perform poorly [2]. This is achieved by adding a new element, the controller, that dynamically adjusts the behavior of one or more other elements based on the measured outputs of the system. We use the term target system to refer to the elements that are manipulated by one or more controllers to achieve desired outputs.

The elements of a closed loop system are depicted in Figure 7.1. Below, we describe these elements and the information, or signals, that flow between elements. Throughout, time is discrete and is denoted by k. Signals are a functional of time.

- The reference input $r(k)$ is the desired value of the measured output (or transformations of them), such as CPU utilization. For example, $r(k)$ might be 66%. Sometimes, the reference input is referred to as the desired output or the set point.
- The control error $e(k)$ is the difference between the reference input and the measured output.
- The control input $u(k)$ is the setting of one or more parameters that manipulate the behavior of the target system(s) and can be adjusted dynamically.
- The controller determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on current and past values of control error.
- The disturbance input $d(k)$ is any change that affects the way in which the control input influences the measured output (e.g., running a virus scan or a backup).
- The measured output $y(k)$ is a measurable characteristic of the target system such as CPU utilization and response time.
- The noise input $n(k)$ changes the measured output produced by the target system. This is also called sensor noise or measurement noise.
- The transducer transforms the measured output so that it can be compared with the reference input (e.g., smoothing stochastics of the output).

In general, there may be multiple instances of any of the above elements. For example, in clustered systems, there may be multiple load balancers (controllers) that regulate the loads on multiple servers (target systems).

To illustrate the foregoing, consider a cluster of three Apache Web Servers. The Administrator may want these systems to run at no greater than 66% utilization so that if any one of them fails, the other two can absorb the load of the failed server. Here, the measured output is CPU utilization. The control input is the maximum number of connections that the server permits as specified by the `MaxClients` parameter. This parameter can be manipulated to adjust CPU utilization. Examples of disturbances are changes in arrival rates and shifts in the type of requests (e.g., from static to dynamic pages). Control theory provides design techniques for determining the values of parameters such as `MaxClients` so that the resulting system is stable and settles quickly in response to disturbances.

Controllers are designed for some intended purpose or control objective. The most common objectives are:

- **regulatory control**: Ensure that the measured output is equal to (or near) the reference input. For example, in a cluster of three web servers, the reference input might be that the utilization of a web server should be maintained at 66% to handle fail-over. If we add a fourth web server to the cluster, then we may want to change the reference input from 66% to 75%.
- **disturbance rejection**: Ensure that disturbances acting on the system do not significantly affect the measured output. For example, when a backup or virus

scan is run on a web server, the overall utilization of the system is maintained at 66%. This differs from regulator control in that we focus on changes to the disturbance input, not to the reference input.

- **optimization**: Obtain the "best" value of the measured output, such as optimizing the setting of `MaxClients` in the Apache HTTP Server so as to minimize response times. Here, there is no reference input.

There are several properties of feedback control systems that should be considered when comparing controllers for computing systems. Our choice of metrics is drawn from experience with commercial information technology systems. Other properties may be of interest in different settings. For example, [21] discusses properties of interest for control of real-time systems.

Below, we motivate and present the main ideas of the properties considered.

- A system is said to be *stable* if for any bounded input, the output is also bounded. Stability is typically the first property considered in designing control systems since unstable systems cannot be used for mission critical work.
- The control system is *accurate* if the measured output converges (or becomes sufficiently close) to the reference input in the case of regulatory control and disturbance rejection, or the measured output converges to the optimal value in the case of an optimization objective. Accurate systems are essential to ensuring that control objectives are met, such as differentiating between gold and silver classes of service and ensuring that throughput is maximized without exceeding response time constraints. Typically, we do not quantify accuracy. Rather, we measure inaccuracy. For a system in steady state, its inaccuracy, or **steady state error** is the steady state value of the control error $e(k)$.
- The system has *short settling times* if it converges quickly to its steady state value. Short settling times are particularly important for disturbance rejection in the presence of time-varying workloads so that convergence is obtained before the workload changes.
- The system should achieve its objectives in a manner that *does not overshoot*. The motivation here is that overshoot typically leads to undershoot and hence to increased variability in the measured output.

Much of our application of control theory is based on the properties of stability, accuracy, settling time, and overshoot. We refer to these as the **SASO** properties.

To elaborate on the SASO properties, we consider what constitutes a stable system. For computing systems, we want the output of feedback control to converge, although it may not be constant due to the stochastic nature of the system. To refine this further, computing systems have operating regions (i.e., combinations of workloads and configuration settings) in which they perform acceptably and other operating regions in which they do not. Thus, in general, we refer to the stability of a system within an operating region. Clearly, if a system is not stable, its utility is severely limited. In particular, the system's response times will be large and highly variable, a situation that can make the system unusable.

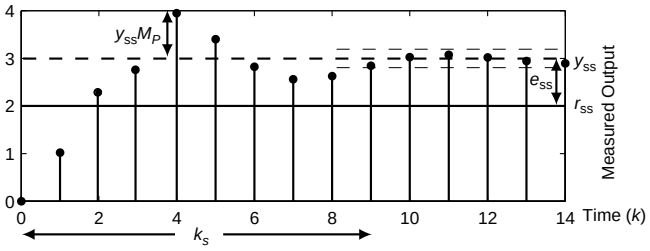


Fig. 7.2 *Response of a stable system to a step change in the reference input. At time* 0*, the reference input changes from* 0 *to* 2*. The system reaches steady state when its output always lies between the light weight dashed lines. Depicted are the steady state error (e_{ss}), settling time (k_s), and maximum overshoot (M_P).*

If the feedback system is stable, then it makes sense to consider the remaining SASO properties—accuracy, settling time, and overshoot. The vertical lines in Figure 7.2 plot the measured output of a stable feedback system. Initially, the (normalized) reference input is 0. At time 0, the reference input is changed to $r_{ss} = 2$. The system responds and its measured output eventually converges to $y_{ss} = 3$, as indicated by the heavy dashed line. The steady state error e_{ss} is -1, where $e_{ss} = r_{ss} - y_{ss}$. The settling time of the system k_s is the time from the change in input to when the measured output is sufficiently close to its new steady state value (as indicated by the light dashed lines). In the figure, $k_s = 9$. The maximum overshoot M_P is the (normalized) maximum amount by which the measured output exceeds its steady state value. In the figure, the maximum value of the output is 3.95 and so $(1 + M_P)y_{ss} = 3.95$, or $M_P = 32\%$.

The properties of feedback systems are used in two ways. The first is for analysis to assess the SASO properties of a system. The second is as design objectives. For the latter, we construct the feedback system to have acceptable values of steady state error, settling time, and maximum overshoot. More details on applying control theory to computing systems can be found in [16].

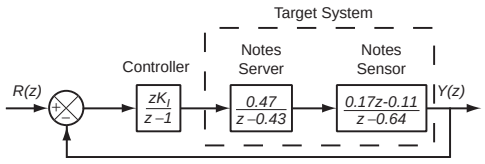


Fig. 7.3 *Block diagram of a feedback system to control RPCs in System for the IBM Lotus Notes Domino Server.*

We describe the essentials of control design using the IBM Lotus Domino Server in [26]. The feedback loop is depicted in Figure 7.3. It consists of the Controller, the Notes Server, and the Notes Sensor. The control objective is regulation, which is motivated by administrators who manage the reliability of Notes Servers by regulat-

ing the number of remote procedure calls (RPCs) in the server. This quantity, which we denote by *RIS*, roughly corresponds to the number of *active users* (those with requests outstanding at the server). Administrators choose a setting for *RIS* that balances the competing goals of maximizing throughput by having high concurrency levels with maximizing server reliability by reducing server loads.

RIS is measured by periodically reading a server log file, which we call the Notes Sensor. Regulation is accomplished by using the `MaxUsers` tuning parameter that controls the number of *connected users*. The correspondence between `MaxUsers` and *RIS* changes over time, which means that `MaxUsers` must be updated almost continuously to achieve the control objective. The controller automatically determines the value of `MaxUsers` based on the objective for *RIS* and the measured value of *RIS* obtained from the Notes Sensor.

Our starting point is to model how `MaxUsers` affects *RIS* as output by the Notes Server. We use $u(k)$ to denote the k-th of `MaxUsers`, and $y(k)$ to denote the k-th value of RIS. (Actually, $u(k)$ and $y(k)$ are offsets from a desired operating point.) We construct an empirical model that relates $y(k)$ to $u(k)$ by applying least squares regression to data obtained from off-line experiments. (Empirical models can also be constructed in real time using on-line data.) The resulting model is

$$y(k) = 0.43y(k-1) + 0.47u(k-1) \tag{7.1}$$

To better facilitate control analysis, Equation (7.1) is put into the form of a transfer function, which is a Z-transform representation of how `MaxUsers` affects *RIS*. Z-transforms provide a compact representation for time varying functions, where z represents a time shift operation. The transfer function of Equation (7.1) is

$$G(z) = \frac{0.47}{z - 0.43}$$

Note that the equation for $G(z)$ appears in the box in Figure 7.3 that corresponds to the Notes Server since $G(z)$ describes the essential control properties of the server. The poles of a transfer function are the values of z for which the denominator is 0. It turns out that the poles determine the stability of the system, and poles largely determine settling times as well. $G(z)$ has one pole, which is 0.43. The effect of this pole on settling time is clear if we solve the recurrence in Equation (7.1). The resulting expression for $y(k)$ has terms with $0.43^k, 0.43^{k-1}, \cdots$. Thus, if the absolute value of the pole is greater than one, the system is unstable. And the closer the pole is to 0, the shorter the settling time. A pole that is negative (or imaginary) indicates an oscillatory response.

The transfer function of a system provides another quantity of interest–steady state gain. Steady state gain quantifies how a change in the input affects the output, a critical consideration in assessing control accuracy. This can be calculated by evaluating $G(z)$ at $z = 1$. A steady state gain of 1 means that the output is equal to the input at steady state.

With this background, we outline how to do control design. We want to construct a controller for the system in Figure 7.3 that results in a closed loop system that

is stable, accurate, and has short settling times. First, observe that the closed loop system itself has a transfer function that relates the reference input to the measured output. We denote this transfer function by $F(z)$. Translating the design objectives into properties of $F(z)$, we want the poles of $F(z)$ to be close to 0 (which achieves both stability and short settling times), and we want $F(z)$'s steady state gain to be 1 (which ensures accuracy since the measured output will be equal to the reference input). These objectives are achieved by choosing the right Controller.

We proceed as follows. First, we construct a transfer function $S(z)$ for the Notes Sensor in the same way as was done with the Notes Server. Next, we choose a parameterized controller. We use an integral controller, which provides incremental adjustments in `MaxUsers`. Specifically, $u(k+1) = u(k) + K_I e(k)$, and its transfer function is $K(z) = \frac{zK_I}{z-1}$. With these two transfer functions, it is straight forward to construct $F(z)$ [16]. It turns out that an integral controller guarantees that $F(z)$ has a steady state gain of 1. Thus, the control design reduces to choosing K_I such that the poles of $F(z)$ are close to 0.

The theory discussed so far addresses linear, time-invariant, deterministic (LTI) systems with a single input (e.g., `MaxUsers`) and a single output (e.g., RIS). There are many extensions to LTI theory. Adaptive control (e.g., [4]) provides a way to automatically adapt the controller in response to changes in the target system and/or workloads. Stochastic control (e.g., [3]) is a framework for going beyond deterministic systems. State space and hybrid systems (e.g., [24]) provide a way to address multiple inputs and multiple outputs as well as complex phase changes. Non-linear control provides a way to address complex relationships between inputs and outputs [29].

7.3 Application to Self-Tuning Memory Management of A Database System

This section describes a feedback control approach that achieves the optimization objective. We study such an approach in the context of memory management in IBM's DB2 Universal Database Management System. The feedback controller manages memory allocation in real time to respond to workload variation and minimize system response time.

Figure 7.4 shows the architecture and system operations of a database server that works with multiple memory pools. The database clients interact with the database server through the database agents which are computing elements that coordinate access to the data stored in the database. Since disk accesses are much slower relative to main memory accesses, database systems use memory pools to cache disk pages so as to reduce the number and time of disk input/output operations needed. The in-memory data are organized in several pools, which are dedicated for different purposes and can be of different types and characteristics (e.g., buffer pools, package cache, sort memory, lock list).

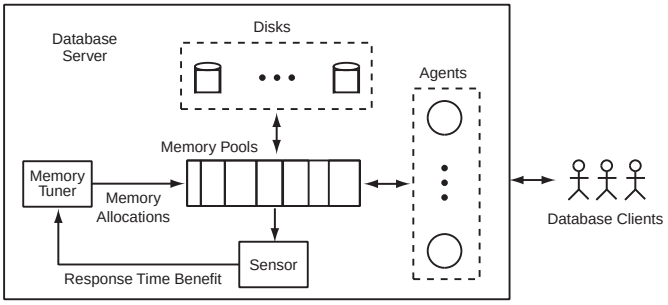


Fig. 7.4 *Architecture of database memory management.*

The management of these pools, especially in terms of determining their optimal sizes, is a key factor in tuning and determining database system performance. However, several challenges are associated with self-tuning memory management.

- Interconnection: In the database memory management problem, the total size of all memory pools is fixed. Increasing the size of one pool necessarily means decreasing the size of another. Although memory pool size increase can drastically reduce its response time to access disk data (since there is a higher probability that a copy of the data is cached in memory), its impact to other memory pools need to be considered as well.
- Heterogeneity: Buffer pools that store data pages or index pages exhibit different data access patterns. Furthermore, besides saving the I/O time, a larger size of memory pool can also lower the CPU time. For example, a larger sort memory increases the width of a merge tournament tree and reduces the number of merge passes so that the time spent in performing tournament merge can be reduced. These dissimilar usage characteristics make memory pool trade offs difficult.
- Adaptation and robustness: Each customer has its unique database configuration, a self-tuning memory controller is required to work out of the box without on-site adjustment. The controller is also required to automatically adjust itself in real time in response to database workload variation. On the other hand, robustness is of great concern to database administrators. Furthermore, for a database server, oscillations in the size of buffer pools is highly undesirable because it reduces throughput as a result of increased I/O rates to write dirty pages and to read new data.
- Cost of control: Care must be taken not to change memory pools too frequently since excessive adjustments introduce substantial resizing overheads that can decrease throughput and increase response time.

We start controller design from identifying the three key signals in a feedback control system: control input, measured output, and reference input (as depicted in Figure 7.1). The control input, $u_i(k), i = 1,2,\ldots,N$, consists of the sizes of all N memory pools subject to self-tuning.

Although we could use system throughput or response time as measured output, they are not proper choices, because they can be affected not only by memory allocation but by many other factors (e.g., indexes and query plans) and their corresponding controllers (e.g., index advisor, query optimizer). Since the direct effect of having memory pools is to reduce the disk access time (and CPU computation time), we only focus on the saved response time in this sense. Specifically, we define measured output, $y_i(k)$, as the response time reduction caused by memory size increase. We refer to this as the *response time benefit* (or simply *benefit*), which is also known as the marginal gain. We measure benefit in units of seconds per page.

The response time benefit is measured dynamically by a special sensor. This sensor uses a "ghost buffer" that estimates the reduction in disk I/Os for a buffer pool if the the size of that buffer pool had been larger. The response time benefit is calculated as the saved disk access time divided by the size of the ghost buffer.

For the database memory problem, the control objective is optimization. Specifically, this is a constrained optimization problem where the objective is to maximize the total saved response time subject to the constraint of the total available memory [9] [11].

We introduce some notation. The scalar performance function is:

$$J = f(u_1, u_2, \ldots, u_N) \tag{7.2}$$

The scalar equality constraint on total memory is:

$$g(u_1, u_2, \ldots, u_N) = \sum_{i=1}^{N} u_i - U = 0 \tag{7.3}$$

Further, there may be N scalar inequality constraints imposed on the memory pools:

$$h_i(u_i) = u_i - \underline{u}_i \geq 0 \tag{7.4}$$

where $\underline{u}_i$ is the minimum size for memory pool i.

Note that for each memory pool, saved response time is increasing in memory size, and saved response time becomes saturated when the pool memory is large enough to hold the entire data block (so that there is no further I/O involved and no additional time can be saved). We assume the relationship between the pool size u_i and saved response time x_i is approximated by $x_i = a_i(1 - e^{-b_i u_i})$. We further assume that the interactions between memory pools are negligible so that the objective function is separable and convex. This gives $f = \sum_{i=1}^{N} x_i = \sum_{i=1}^{N} a_i(1 - e^{-b_i u_i})$ and its partial derivative (i.e., measured output) is $y_i = \frac{\partial f}{\partial u_i} = \frac{dx_i}{du_i} = a_i b_i e^{-b_i u_i}$.

According to the first order Karush-Kuhn-Tucker (KKT) necessary conditions, we define the Lagrange function as $L = f(u_1, u_2, \ldots, u_N) + \lambda g(u_1, u_2, \ldots, u_N) + \sum_{i=1}^{N} \mu_i^{\top} h_i(u_i)$, which adjoins the original performance function and the constraints using the Lagrange multipliers λ and μ_i. The KKT necessary conditions for a solution $u = [u_1, u_2, \ldots, u_N]$ to be locally optimal are that the constraints are satisfied, i.e., $g(u) = 0$ and $h(u) = [h_1(u_1), h_2(u_2), \ldots, h_N(u_N)] \geq 0$, and there exist Lagrange multipliers λ and μ_i such that the gradient of the Lagrangian vanishes. That is,

$\frac{\partial L}{\partial u_i} = \frac{\partial f}{\partial u_i} + \lambda \frac{\partial g}{\partial u_i} + \sum_{j=1}^{N} \mu_j \frac{\partial h_j}{\partial u_i} = y_i + \lambda + \mu_i = 0$. Furthermore, μ_i satisfies the complementarity condition of $\mu_i h_i = 0$ with $\mu_i \geq 0$. This implies that when the memory allocation is optimal and pool sizes are not at the boundaries, the measured outputs of memory pool are equal ($y_i = -\lambda$, and $\mu_i = 0$ since $h_i > 0$). In the case that the memory allocation is optimal when some pool sizes are at the boundaries, the measured output from these memory pool may be smaller ($y_i = -\lambda - \mu_i$, and $\mu_i \geq 0$ since $h_i = 0$). Since f is a convex function, the optimal solution is unique in that the local optimum is also the global optimum.

We design a multiple-input multiple-output (MIMO) feedback controller to equalize the measured output. Such an approach allows us to exploit well established techniques for handling dynamics and disturbances (from changes in workloads) and to incorporate the cost of control (throughput reductions due to load imbalance and resource resizing) into the design. The feedback control system is defined as follows (where matrices are denoted by boldface uppercase letters and vectors by boldface lowercase):

$$\mathbf{y}(k+1) = \mathbf{A}\mathbf{y}(k) + \mathbf{B}\left(\mathbf{u}(k) + \mathbf{d}^{\mathbf{I}}(k)\right) \tag{7.5}$$

$$\mathbf{e}(k) = \left(\frac{1}{N}\mathbf{1}_{N,N} - \mathbf{I}\right)\left(\mathbf{y}(k) + \mathbf{d}^{\mathbf{O}}(k)\right) \tag{7.6}$$

$$\mathbf{e}_{\mathbf{I}}(k+1) = \mathbf{e}_{\mathbf{I}}(k) + \mathbf{e}(k) \tag{7.7}$$

$$\mathbf{u}(k) = \mathbf{K}_{\mathbf{P}}\mathbf{e}(k) + \mathbf{K}_{\mathbf{I}}\mathbf{e}_{\mathbf{I}}(k) \tag{7.8}$$

The first equation represents a state space model [14], which is a local linear approximation of the concave memory-benefit relationship. Although most computing systems are inherently non linear, from the local point of view, a linear approximation can be effective and rational, especially when considering the existence of system noise and the ability of on line model adaptation. The $N \times 1$ vector $\mathbf{y}(k)$ denotes the measured output (i.e., response time benefit), the $N \times 1$ vector $\mathbf{u}(k)$ represents the control input (i.e., memory pool size), and the $N \times 1$ vector $\mathbf{d}^{\mathbf{I}}(k)$ indicates possible disturbances applied on the control inputs (e.g., adjustments made to enforce the equality and inequality resource constraints). The $N \times N$ matrices $\mathbf{A}$ and $\mathbf{B}$ contain state space model parameters that can be obtained from measured data and system identification [20].

Equation (7.6) specifies the $N \times 1$ control error vector $\mathbf{e}(k)$, where $\mathbf{I} = \begin{bmatrix} 1 \cdots 0 \\ \vdots \quad \vdots \\ 0 \cdots 1 \end{bmatrix}$ and $\mathbf{1}_{N,N} = \begin{bmatrix} 1 \cdots 1 \\ \vdots \quad \vdots \\ 1 \cdots 1 \end{bmatrix}$ are $N \times N$ matrices. The $N \times 1$ vector $\mathbf{d}^{\mathbf{O}}(k)$ indicates possible disturbances applied on the measured outputs (e.g., measurement noises that are not characterized by the deterministic model). Implied from this equation is that we define the average measured output $\bar{y}(k) = \frac{1}{N}\sum_{i=1}^{N} y_i(k)$ as the control reference

for all measured outputs, and the i-th control error $e_i(k) = \overline{y}(k) - y_i(k)$. Note that in contrast to having a static value or external signal as the reference input, we specify the reference as a linear transformation of the measured outputs. The control objective is to make $e_i(k) = 0$, that is, equalizing the measured outputs (i.e., $y_i(k) = y_j(k)$ for any i and j) so as to maximize the total saved response time.

The dynamic state feedback control law is defined in Equation (7.8), and the integral control error $\mathbf{e_I}(k)$ is the $N \times 1$ vector representing the sum of the control errors as defined in Equation (7.7). The $N \times N$ matrices $\mathbf{K_P}$ and $\mathbf{K_I}$ are controller parameters to be chosen (through controller design) in order to stabilize the closed loop system and achieve the SASO performance criteria regarding convergence and settling time.

We design the controller and choose the control parameters in a way that considers the cost of control–both the cost of transient memory imbalances and the cost of changes in memory allocations [10]. Reducing memory imbalance generally indicates an aggressive control strategy with short settling time of moving the memory from imbalance to balance. However, too aggressive control can also lead to overreacting to random fluctuations and thus incurs additional cost of allocation changes.

We handle this trade-off by exploiting optimal linear quadratic regulator (LQR) control [15]. LQR chooses control parameters that minimize the quadratic cost function

$$J = \sum_{k=1}^{\infty} \left[\mathbf{e}^\top(k) \ \mathbf{e_I}^\top(k) \right] \mathbf{Q} \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} + \mathbf{u}(k)^\top \mathbf{R}\mathbf{u}(k) \tag{7.9}$$

over an infinite time horizon as well as satisfy the dynamics defined in Equation (7.5)-(7.8). The cost function includes the control error $\mathbf{e}(k)$ and $\mathbf{e_I}(k)$, and the control input $\mathbf{u}(k)$. The former is related to the cost of transient resource imbalances, and the latter the cost of changing resource allocations. The matrices $\mathbf{Q}$ and $\mathbf{R}$ determine the trade-off. Intuitively, if $\mathbf{Q}$ is large compared to $\mathbf{R}$, the controller will make big changes in resource allocations and hence can react quickly to disturbances. On the other hand, if $\mathbf{R}$ is large compared to $\mathbf{Q}$, the controller is much more conservative since there is a high cost for changing resource allocations.

With $\mathbf{Q}$ and $\mathbf{R}$ defined, the control parameters $\mathbf{K_P}$ and $\mathbf{K_I}$ can be computed in the usual way by solving the Riccati equation [4]. Hence, the controller design problem is to select the proper weighting matrices $\mathbf{Q}$ and $\mathbf{R}$ which quantify the cost of control. We achieve this by developing a cost model, regarding to the performance impact of control, and constructing $\mathbf{Q}$ and $\mathbf{R}$ in a systematic way [10].

Although the cost model and LQR framework provides a systematic way to study the cost of control, it is more appropriate to be used off-line for analyzing the target system and designing the controller prior to operation. Further simplification is needed to facilitate real time adaptation when the workload is unknown in advance and can change overtime. This also helps to manage a large set of memory pools where the number of pools is varying.

This simplification is achieved using a distributed control architecture and adaptive pole placement techniques. The model is built and the controller is designed locally for each individual memory pool; the only connection between different

pools is the control reference signal–the average measured output. Specifically, a single-input single-output (SISO) model

$$y_i(k+1) = b_i(k)u_i(k) \tag{7.10}$$

is built on line for the i-th memory pool. This is equivalent to having $\mathbf{A} = \mathbf{0}$ and $\mathbf{B} = diag([b_1, \ldots, b_N])$ in Equation (7.5), while the disturbance term $\mathbf{d^I}(k)$ is enlarged to include the modeling uncertainty. Having a set of SISO models simplifies the model structure and parameter, so that on line modeling techniques such as recursive least squares can be effectively applied with less computational complexity [20].

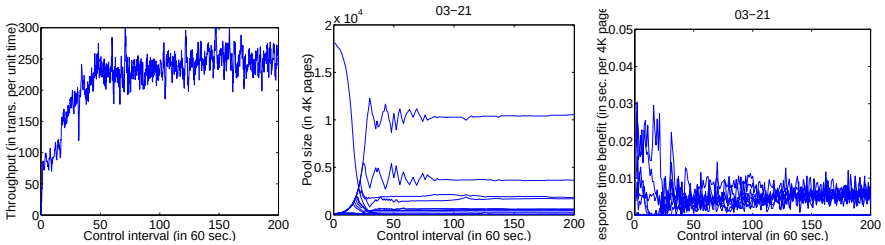
The controller is also built individually

$$u_i(k+1) = u_i(k) - \frac{1-p}{b_i(k)}\left(y_i(k) - \frac{1}{N}\sum_{j=1}^{N} y_j(k)\right) \tag{7.11}$$

The controller takes the format of integral control, a simplification from Equation (7.8) by setting $\mathbf{K_P} = \mathbf{0}$ and $\mathbf{K_I} = diag([\frac{1-p}{b_1(k)}, \ldots, \frac{1-p}{b_N(k)}])$. The control parameter $\frac{1-p}{b_i(k)}$ is designed through adaptive pole placement so that it will be adapted when different model parameter $b_i(k)$ is estimated on line.

With reasonable simplifications, a distributed architecture makes the controller agile to workload and resource variations, and increase its robustness regarding to measurement uncertainties and maybe uneven control intervals. For example, although in general for a database server the system dynamics may not be negligible (i.e., an increase of buffer pool size may not immediately result in response time benefit decrease, as time is needed to fill up the added buffer space) and the cross memory pool impact does exist (i.e., an increase of sort memory will not only bring down the benefit for sort memory but also that for the buffer pool that stores temporary sort spill pages), our experimental results confirm the control performance of this distributed controller.

Figure 7.5 evaluates the performance of the feedback controller under an on line transaction processing (OLTP) workload. The OLTP workload consists of a large number of concurrent requests, each of which has very modest resource demands; we use 20 buffer pools to contain data and index for the database tables and 50 database clients to generate the load. Figure 7.5(a) shows the throughput (measured in transactions per unit time) that indicates the performance impact of buffer pool re-sizings. Figure 7.5(b) and (c) display the memory allocations and response time benefits for the controlled buffer pools (as indicated by the 20 solid lines in the plot). Initially, the database memory is not properly allocated: most of the memory has been allocated to one buffer pool, while the other buffer pools are set at the minimum size. The controller adjusts the size of buffer pools so as to equalize the response time benefits of the pools. We see that even for a large number of memory pools the controller converges in approximately 80 intervals. Further, our studies in [10] show that the controller's actions increases throughput by a factor of three.



(a) OLTP throughput *(b) Memory pool sizes* *(c) Response time benefit*

Fig. 7.5 *Control performance under an OLTP workload.*

7.4 Application to CPU Utilization Control in Distributed Real-Time Embedded Systems

Distributed real-time embedded (DRE) systems must control the CPU utilization of multiple processors to prevent overload and meet deadlines in face of fluctuating workload. We present the *End-to-end Utilization CONtrol (EUCON)* algorithm that controls the CPU utilization of all processors in a DRE system by dynamically adjusting the invocation rates of periodic tasks. A DRE system is comprised of m end-to-end periodic tasks $\{T_i | 1 \leq i \leq m\}$ executing on n processors $\{P_i | 1 \leq i \leq n\}$. Task T_i is composed of a chain of subtasks $\{T_{ij} | 1 \leq j \leq n_i\}$ running on multiple processors. The execution of a subtask T_{ij} is triggered by the completion of its predecessor $T_{i,j-1}$. Hence all the subtasks of a task are invoked at a same rate. For example, on a Real-Time CORBA middleware a task may be implemented as a sequence of remote operation requests to distributed objects, where each remote operation request corresponds to a subtask. Each subtask T_{ij} has an *estimated* execution time c_{ij} known at deployment time. However, the *actual* execution time of a subtask may differ from c_{ij} and vary at run time. The rate of T_i can be dynamically adjusted within a range $[R_{min,i}, R_{max,i}]$. A task running at a higher rate contributes higher utility at the cost of higher CPU utilization. For example, both video streaming and digital control applications usually deliver better performance when running at higher rates.

As shown in Figure 7.6, EUCON is composed of a centralized *controller*, and a *utilization monitor* and a *rate modulator* on each processor. A separate TCP connection connects the controller with the pair of utilization monitor and rate modulator on each processor. The user inputs to the controller include the utilization set points, $\mathbf{B} = [B_1 \ldots B_n]^T$, which specify the desired CPU utilization of each processor, and the rate constraints of each task. The *measured output* is the CPU utilization of all processors, $\mathbf{u}(k) = [u_1(k) \ldots u_n(k)]^T$. The *control input* is the change to task rates $\triangle\mathbf{r}(k) = [\triangle r_1(k) \ldots \triangle r_m(k)]^T$, where $\triangle r_i(k) = r_i(k) - r_i(k-1)$ $(1 \leq i \leq m)$. The goal of EUCON is to regulate the CPU utilizations of all processors so that they remain close to their respective set points by adjusting the task rates, despite variations in task execution times at run time.

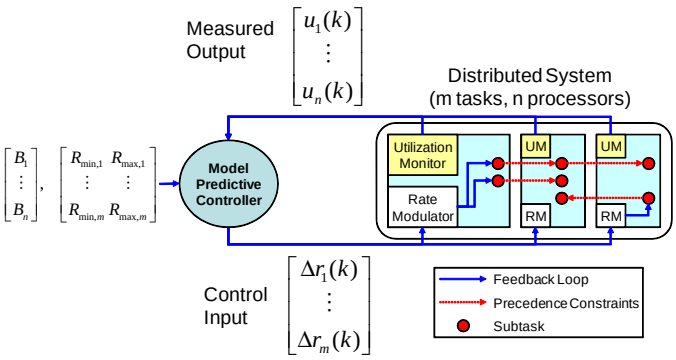


Fig. 7.6 *The feedback control loop of EUCON.*

DRE systems pose several challenges to utilization control. First, the utilization control problem is *multi-input-multi-output (MIMO)* in that the system needs to regulate the CPU utilization of multiple processors by adjusting the rates of multiple tasks. More importantly, the CPU utilization of different processors is *coupled* to each other due to the correlation among subtasks belonging to a same task, *i.e.*, changing the rate of a task will affect the utilization of all the processors hosting its subtasks because they must execute at the same rates. Therefore the CPU utilization of different processors cannot be controlled independently from each other. Finally, the control is subject to *actuator constraints* as the rate of a task must remain with an application-specific range.

To deal with inter-processor coupling and rate constraints, EUCON adopts *Model Predictive Control (MPC)* [23], an advanced control technique used extensively in industrial process control. Its major advantage is that it can deal with coupled MIMO control problems with constraints on the actuators. The basic idea of MPC is to optimize an appropriate cost function defined over a time interval in the future. The controller employs a model of the system which is used to predict the behavior over P sampling periods called the *prediction horizon*. The control objective is to select an *input trajectory* to minimize the cost subject to the actuator constraints. An input trajectory includes the control inputs in the following M sampling periods, $\triangle\mathbf{r}(k), \triangle\mathbf{r}(k+1|k), \ldots \triangle\mathbf{r}(k+M-1|k)$, where M is called the *control horizon*. The notation $\triangle\mathbf{r}(k+1|k)$ means that $\triangle\mathbf{r}(k+1)$ depends on the conditions at time k. Once the input trajectory is computed, only the first element $(\triangle\mathbf{r}(k))$ is applied as the control input to the system. In the next step, the prediction horizon slides one sampling period and the input trajectory is computed again based on the measured output $(\mathbf{u}(k))$.

Before designing the controller for EUCON, we derive a dynamic model that characterizes the relationship between the control input $\triangle\mathbf{r}(k)$ and the measured output $\mathbf{u}(k)$. First, we model the utilization $u_i(k)$ of one processor P_i. Let $\triangle r_j(k)$ denote the change to the task rate, $\triangle r_j(k) = r_j(k) - r_j(k-1)$. We define the *estimated change to utilization*, $\triangle b_i(k)$, as:

$$\triangle b_i(k) = \sum_{T_{jl} \in S_i} c_{jl} \triangle r_j(k) \tag{7.12}$$

where S_i represents the set of subtasks located at processor P_i. Note $\triangle b_i(k)$ is based on the *estimated* execution time. Since the *actual* execution times may differ from their estimation, we model the utilization $u_i(k)$ as:

$$u_i(k) = u_i(k-1) + g_i \triangle b_i(k-1) \tag{7.13}$$

where the *utilization gain* g_i represents the ratio between the change to the *actual* utilization and the estimated change $\triangle b_i(k-1)$. For example, $g_i = 2$ means that the actual change to utilization is twice of the estimated change. Note that the value of g_i is *unknown a priori* due to the uncertainty of subtasks' execution times. A system with m processors is described by the following MIMO model:

$$\mathbf{u}(k) = \mathbf{u}(k-1) + \mathbf{G} \triangle \mathbf{b}(k-1) \tag{7.14}$$

where $\triangle \mathbf{b}(k-1)$ is a vector including the estimated change to the utilization of each processor, and $\mathbf{G}$ is a diagonal matrix where $g_{ii} = g_i (1 \leq i \leq n)$ and $g_{ij} = 0 (i \neq j)$. The relationship between the changes to the utilizations and the changes to the task rates is characterized as follows:

$$\triangle \mathbf{b}(k) = \mathbf{F} \triangle \mathbf{r}(k) \tag{7.15}$$

where the *subtask allocation matrix*, $\mathbf{F}$, is an $n \times m$-order matrix. $f_{ij} = c_{jl}$ if subtask T_{jl} (the l^{th} subtask of task T_j) is allocated to processor i, and $f_{ij} = 0$ if no subtask of task T_j is allocated to processor i. Note that $\mathbf{F}$ captures the inter-processor coupling caused by end-to-end tasks. Equations (7.14-7.15) give the dynamic model of a distributed system with m tasks and n processors.

Based on the system model, we now design the controller. In the end of every sampling period, the controller computes the control input $\triangle \mathbf{r}(k)$ that minimizes the following cost function under the rate constraints:

$$\mathbf{V}(k) = \sum_{i=1}^{P} \| \mathbf{u}(k+i|k) - \mathbf{ref}(k+i|k) \|^2 + \sum_{i=0}^{M-1} \| \triangle \mathbf{r}(k+i|k) - \triangle \mathbf{r}(k+i-1|k) \|^2 \tag{7.16}$$

where P is the *prediction horizon*, and M is the *control horizon*. The first term in the cost function represents the *tracking error*, *i.e.*, the difference between the utilization vector $\mathbf{u}(k+i|k)$ and a *reference trajectory* $\mathbf{ref}(k+i|k)$. The reference trajectory defines an ideal trajectory along which the utilization vector $\mathbf{u}(k+i|k)$ should change from the current utilization $\mathbf{u}(k)$ to the utilization set points $\mathbf{B}$. Our controller is designed to track the following exponential reference trajectory so that the closed-loop system behaves like a linear system:

$$\mathbf{ref}(k+i|k) = \mathbf{B} - e^{-\frac{T_S}{T_{ref}} i} (\mathbf{B} - \mathbf{u}(k)) \tag{7.17}$$

where T_{ref} is the time constant that specifies the speed of system response. A smaller T_{ref} causes the system to converge faster to the set point. By minimizing the tracking error, the closed loop system will converge to the utilization set point if the system is stable. The second term in the cost function (7.16) represents the *control penalty*, which causes the controller to reduce the changes to the control input.

The controller minimizes the cost function (7.16) under the rate constraints based on an approximate system model. This constrained optimization problem can be transformed to a standard constrained *least-squares* problem. The controller can then use a standard *least-squares* solver to solve this problem on-line [22].

Note that the system model described in (7.14) and (7.15) cannot be used directly by the controller because the system gains **G** are unknown. The controller assumes $\mathbf{G} = \mathbf{I}$ in (7.14), *i.e.*, the actual utilization is the same as the estimation. Although this approximate model may behave differently from the real system, as proven in [22], the closed loop system can maintain stability and track the utilization set points as long as the actual **G** remains within a certain range. Furthermore, this range can be established using stability analysis of the closed-loop system.

EUCON has been implemented in FC-ORB [31], a distributed middleware for DRE systems. We now summarize the representative experimental results presented in [31]. All tasks run on a Linux cluster composed of four Pentium-IV machines. The EUCON controller is located on another Pentium-IV machine. The workload comprises 12 tasks with a total of 25 subtasks. In the first experiment shown in Figure 7.7(a), the average execution times of all subtasks change simultaneously. The execution times of all subtasks increase by 50% at 600 seconds, EUCON responds to the overload by reducing task rates, which causes the utilization of every processor to converge to its set point within 100 seconds (25 sampling periods). At 1000 seconds, the utilization of every processor drops sharply due to 56% decrease in the execution times of all subtasks. EUCON increases task rates until the utilizations reconverge to their set points. In the second experiment shown in Figure 7.7(b), only the average execution times of the subtasks on one of the processors experience the same variations as in the first run, while all the other subtasks maintain the same average execution times. As shown in Figure 7.7(b) the utilization of every processor converges to its set point after the variation of execution times at 600 seconds and 1000 seconds, respectively. These results demonstrate that EUCON can effectively control the utilization of multiple processors under varying execution times, while handling inter-processor coupling and rate constraints.

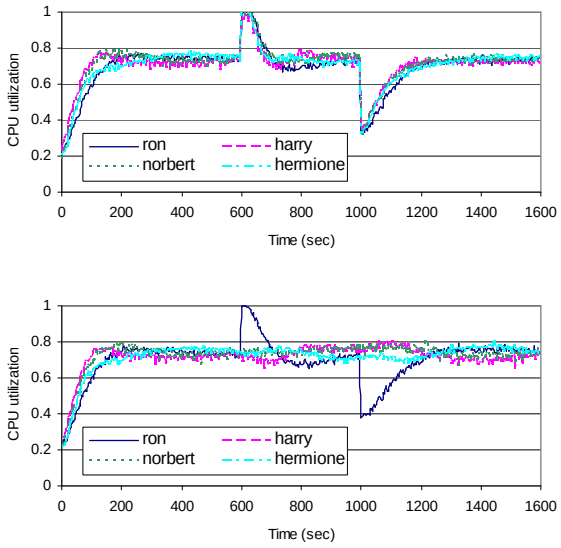


Fig. 7.7 The CPU utilization of all the processors in a Linux cluster when subtask execution times change on all four processors (top figure) and only one processor (bottom figure)

7.5 Application to Automated Workload Management in Virtualized Data Centers

7.5.1 Introduction

Data centers today play a major role in providing on-demand computing to enterprise applications supporting key business processes including supply chain, e-commerce, payroll, customer relationship management, etc. These applications typically employ a multi-tier architecture where distinct components of a single application, e.g., the web tier, the application tier, and the database tier, spread across multiple servers. In recent years, there has been wide adoption of server virtualization in data centers due to its potential to reduce both infrastructure and operational costs. Figure 7.8 shows an example scenario where multiple multi-tier applications share a common pool of physical servers. Each physical server contains multiple virtual containers, and each virtual container hosts a specific component of a multi-tier application. Here a "virtual container" can be a hypervisor-based virtual machine (e.g., VMware, Xen), an operating system level container (e.g., OpenVZ, Linux VServer), or a workload group (e.g., HP Global Workload Manager, IBM Enterprise Workload Manager). Although the grouping of application tiers can be arbitrary in general, we specifically consider the case where the same tiers from different ap-

plications are hosted on the same physical server. This is a common scenario for shared hosting environments for potential savings in software licensing costs.

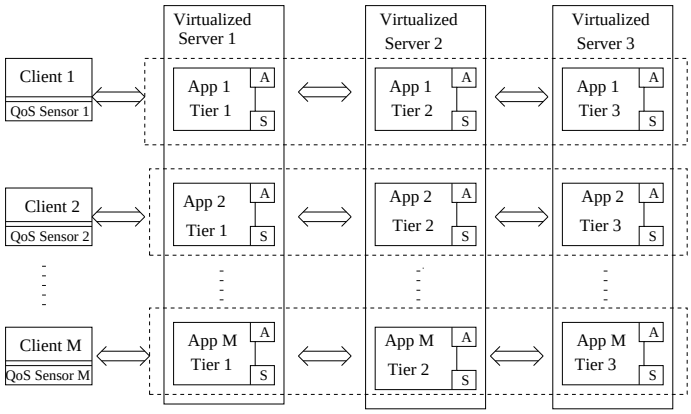


Fig. 7.8 A virtualized server pool hosting multiple multi-tier applications

When multiple enterprise applications share a common infrastructure, meeting application-level QoS goals becomes a challenge for data center operators due to the time-varying nature of typical enterprise workloads, and the complex interactions among individual tiers of the hosted applications. Existing workload management tools for UNIX systems or mainframes typically allow individual virtual containers to be dynamically sized in order to maintain a specified level of resource utilization. However, these tools can neither manage other types of containers such as virtual machines, nor provide direct guarantees for application-level QoS. In the past few years, there has been work in applying control theory to the design of automated workload management solutions that fill these gaps [34, 32, 19]. In [34], QoS-driven workload management was presented using a nested feedback controller, where the inner loop regulates the CPU utilization of a virtual container and the outer loop maintains the application-level response time at its target. In [32], a predictive controller was developed to allocate CPU resource to a virtual container proactively by exploiting repeatable patterns in an application's resource demands. This controller has been tested for managing Xen virtual machines, and a variation of it has been integrated into the latest release of the HP Global Workload Manager [8].

The work in [19] deals with the scenario where some virtualized servers are *overloaded*. This means, the aggregate demand from all the application components sharing a server exceeds its total capacity. In this case, the performance of all the applications may suffer. This is undesirable because failing to meet the QoS goals may have different consequences for different applications, depending on their respective service level agreements (SLAs). Therefore, it is desirable for a workload management solution to also provide service differentiation among co-hosted applications in order to maximize the overall business value generated by these applications.

7.5.2 Problem statement

Consider the system in Figure 7.8, where N ($N = 3$) virtualized servers are used to host M 3-tier applications. When one or more of the virtualized servers become overloaded, the workload management tool needs to dynamically allocate the shared server resources to individual tiers of the M applications in a coordinated fashion such that a specified level of QoS differentiation can be maintained. Next, we describe how this problem can be cast into a feedback control problem. For simplicity, we assume that only a single resource on a server (e.g., CPU) may become a bottleneck. The approach described here can be generalized to handle multiple resource bottlenecks.

Each virtual container has an actuator (box "A" in Figure 7.8) associated with it, which can allocate a certain percentage of the shared server resource to the application component running in the container. This is referred to as "resource entitlement." At the beginning of each control interval k, the *control input* $\mathbf{u}(k)$ is fed into the actuators, where $u_{i,j}(k)$ denotes the resource entitlement for tier j of application i during interval k. Since $\sum_{i=1}^{M} u_{i,j} = 1$, $1 \leq j \leq N$, there are a total of $(M-1) \times N$ such independent variables. Hence, $\mathbf{u}(k)$ is an $(M-1) \times N$-dimensional vector.

Each application has a QoS sensor (see Figure 7.8) that measures some end-to-end performance (e.g., mean response time, throughput) at the end of each control interval. Let $q_i(k)$ denote the QoS measurement for application i during interval $k-1$. We then define the *measured output*, $\mathbf{y}(k)$, to be the normalized QoS ratios for individual applications, where $y_i(k) = \frac{q_i(k)}{\sum_{m=1}^{M} q_m(k)}$. Since $\sum_{i=1}^{M} y_i(k) = 1$, only $M-1$ of such $y_i(k)$'s are independent. As a result, the system output $\mathbf{y}(k)$ is an $(M-1)$-dimensional vector.

The goal of the feedback controller is to automatically determine the appropriate value for each $u_{i,j}(k)$, such that each $y_i(k)$ can track its *reference input*, $r_i(k)$, the desired QoS ratio for application i when the system is overloaded.

7.5.3 Adaptive optimal controller design

We now describe the adaptive optimal controller we presented in [19] for the service differentiation problem. A block diagram of the closed-loop control system is shown in Figure 7.9. The controller consists of two key modules: a *model estimator* that learns and periodically updates a linear model between the resource entitlements for individual application tiers and the measured QoS ratios, and an *optimal controller* that computes the optimal resource entitlements based on estimated model parameters and a quadratic cost function.

We use the following linear, auto-regressive MIMO model to represent the input-output relationship in the controlled system:

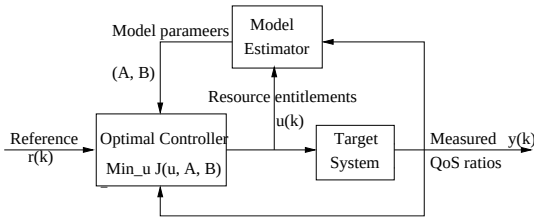


Fig. 7.9 A self-tuning optimal resource control system

$$\mathbf{y}(k+1) = \sum_{l=1}^{n} \mathbf{A}_l \mathbf{y}(k+1-l) + \sum_{m=0}^{n-1} \mathbf{B}_m \mathbf{u}(k-m). \tag{7.18}$$

Note that $\mathbf{A}_l \in \Re^{O\times O}$ and $\mathbf{B}_m \in \Re^{O\times V}$, where $V = (M-1)\times N$ is the input dimension, and $O = M-1$ is the output dimension. The use of a MIMO model allows us to capture complex interactions and dependencies among resource entitlements for different application tiers, which cannot be captured by individual SISO models. The order of the model, n, captures the amount of memory in the system. Its value can be estimated in offline system identification experiments [20]. Typically, a low-order model is sufficient for computing systems [16]. Since the linear model is a local approximation of the real system dynamics that is typically nonlinear, we estimate and adapt the values of the coefficient matrices, $\mathbf{A}_l$ and $\mathbf{B}_m$, online using the recursive least squares (RLS) estimator [4], whenever a new measurement of $\mathbf{y}(k)$ becomes available.

We use optimal control that minimizes the following quadratic cost function:

$$J = \|\mathbf{W}(\mathbf{y}(k+1) - \mathbf{r}(k+1))\|^2 + \|\mathbf{Q}(\mathbf{u}(k) - \mathbf{u}(k-1))\|^2. \tag{7.19}$$

The controller aims to steer the system into a state of optimum reference tracking, while penalizing large changes in the control variables. $\mathbf{W} \in \Re^{O\times O}$ and $\mathbf{Q} \in \Re^{V\times V}$ are weighting matrices on the tracking errors and the changes in the control actions, respectively. They are commonly chosen as diagonal matrices. Their relative magnitude provides a trade off between the *responsiveness* and the *stability* of the control system.

The optimal control law, $\mathbf{u}^*(k)$, can be derived by first explicitly expressing the dependency of the cost function J on $\mathbf{u}(k)$, and then solving the equation $\frac{\partial J}{\partial \mathbf{u}(k)} = 0$. As a result, we get

$$\mathbf{u}^*(k) = ((\mathbf{W}\hat{\mathbf{B}}_0)^T \mathbf{W}\hat{\mathbf{B}}_0 + \mathbf{Q}^T\mathbf{Q})^{-1}[(\mathbf{W}\hat{\mathbf{B}}_0)^T \mathbf{W}(\mathbf{r}(k+1) - \hat{\mathbf{X}}\tilde{\phi}(k)) + \mathbf{Q}^T\mathbf{Q}\mathbf{u}(k-1)],$$

where

$$\tilde{\phi}(k) = [0\ \mathbf{u}^T(k-1)\ \ldots\ \mathbf{u}^T(k-n+1)\ \mathbf{y}^T(k) \cdots\ \cdots\ \mathbf{y}^T(k-n+1)]^T\ ,$$
$$\hat{\mathbf{X}} = [\hat{\mathbf{B}}_0,\ \ldots,\ \hat{\mathbf{B}}_{n-1},\ \hat{\mathbf{A}}_1,\ \ldots,\ \hat{\mathbf{A}}_n].$$

Note that $\hat{\mathbf{X}}$ and $\hat{\mathbf{B}}_0$ are online estimates of the model parameters.

7.5.4 Experimental evaluation

Our controller design has been validated on a two-node testbed hosting two instances of the RUBiS application [1], an online auction benchmark. We use a two-tier implementation consisting of an Apache web server and a MySQL database (DB) server. Each application tier is hosted in a Xen virtual machine. The "web node" is used to host two web tiers, and the "DB node" is used to host two DB tiers. For this application, CPU is the only potential resource bottleneck. We use the credit-based CPU scheduler in the hypervisor of Xen 3.0.3 unstable branch [7] as the actuator in our control loop. It implements proportional fair sharing of the CPU capacity among multiple virtual machines.

We choose a control interval of 20 seconds, which offers a good balance between responsiveness of the controller and predictability of the measurements. For each RUBiS application i, we use mean response time per interval ($RT_i(k)$) as the QoS metric, and the normalized RT ratio, $y(k) = RT_1(k)/(RT_1(k) + RT_2(k))$, as the measured output. The reference input, $r(k)$, indicates the desired level of QoS differentiation between the two applications. Note that both $y(k)$ and $r(k)$ are scalars in this example.

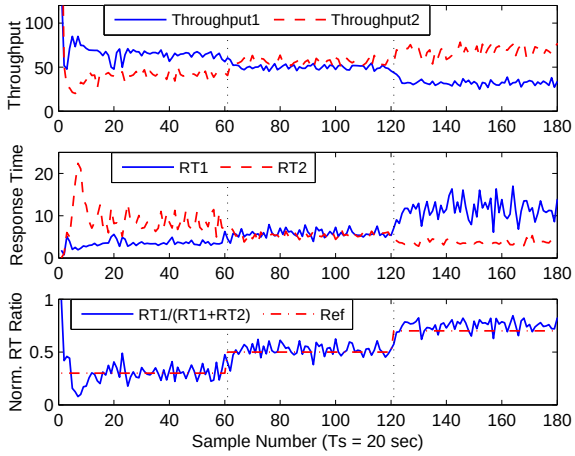
In the first experiment, we varied the reference input, $r(k)$, from 0.3 to 0.5 then to 0.7. Each reference value was used for a period of 60 control intervals.

Figure 7.10(a) shows the measured per-interval throughput in requests per second (top) and the mean response time in seconds (middle) for the two applications, as well as the normalized RT ratio $y(k)$ against the reference input $r(k)$ (bottom) over a period of 180 control intervals (one hour). The vertical dashed lines indicate the two step changes in the reference input. As we can see, the measured output was able to track the changes in the reference input fairly closely. The performance of both applications also behaved as we expected. For example, a $r(k)$ value of 0.3 gave preferential treatment to application 1, where application 1 achieved higher throughput and lower average response time than application 2 did. When $r(k)$ was set at 0.5, both applications achieved comparable performance. Finally, as $r(k)$ was increased to 0.7, application 2 was able to achieve a higher level of performance than application 1 did, which was consistent with our expectation.

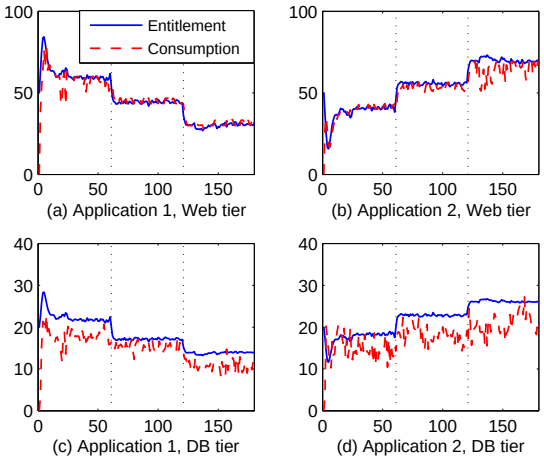
Figure 7.10(b) shows the corresponding CPU entitlements and resulting CPU consumptions of individual application tiers. As we can see, as $r(k)$ went from 0.3 to 0.5 to 0.7, our controller allocated less and less CPU capacity to both tiers in application 1, and more CPU capacity to application 2.

In the second experiment, we fixed the target RT ratio at $r(k) = 0.7$, and varied the intensity of the workload for application 1 from 300 to 500 concurrent users. This effectively created varying resource demands in both tiers of application 1. Experimental results showed that, the controller was able to allocate the CPU capacity on both nodes accordingly, and always maintained the normalized RT ratio near the reference value, in spite of the change in the workload.

In this section, we described how control theory can be applied to the design of automated workload management solutions for a virtualized data center. In particular, as one or more virtualized servers become overloaded, our controller can



(a) QoS metrics for the two applications



(b) CPU entitlement (solid) and consumption (dashed) for individual application tiers

Fig. 7.10 Experimental results with changes in reference input

dynamically allocate shared server resources to individual application tiers in order to maintain a desired level of service differentiation among co-hosted applications. The self-tuning optimal controller we presented has been validated on a lab testbed, and has demonstrated good closed-loop properties in face of workload variations or changes in the reference input.

7.6 Application to Power and Performance in Data Centers

The following case study is motivated by the importance of energy saving in multi-tier Web server farms. In large server farms, it is reported that 23-50% of the revenue is spent on energy [13, 6]. In order to handle peak load requirements, server farms are typically over-provisioned based on offline analysis. A considerable amounts of energy can be saved by reducing resource consumption during non-peak conditions. Significant research efforts have been expended on applying dynamic voltage scaling (DVS) to computing systems in order to save power while meeting time or performance constraints [13, 6, 12, 28, 27, 33].

In this section, we describe adaptive techniques for energy management in server farms based on optimization and feedback control. We specifically illustrate the importance of *joint* adaptation. We show that in large-scale systems, the existence of several individually stable adaptive components may result in a collectively unstable system. For example, a straightforward combination of two energy-saving policies may result in a larger energy expenditure than that with either policy in isolation. We illustrate this problem by exploring a combination of 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). We then provide a solution to avoid the unstable interaction between the two policies.

Figure 7.11 shows experimental results from a three-tier Web server farm testbed. 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 an optimized policy that we explain later in this section. It is clearly demonstrated that when the workload increases, the combined On/Off + DVS policy spends much more energy than all other policies.

The adverse interaction is because the DVS policy reduces the frequency of a processor, increasing system utilization, which increases end-to-end delay causing the On/Off policy to to turn more machines on.

7.6.1 Design Methodology for Integrating Adaptive Policies

In this section, we describe how to design feedback control mechanisms that are free of adverse interactions, optimize energy and respect end-to-end resource and timing constraints. Our solution methodology is divided into three steps:

1. Formulate the optimization problem: Optimization is performed with respect to the available feedback control knobs subject to (i) resource constraints, and (ii)

[1] Observe 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.

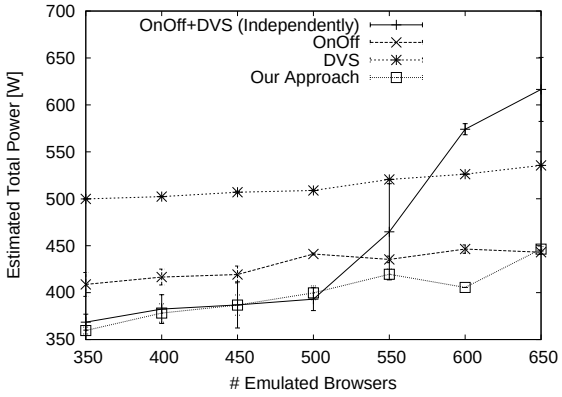


Fig. 7.11 Comparison of total system power consumption for different adaptive policies in the Web server case study.

performance specification constraints. Suppose there are a total of n different feedback control policies. For each feedback control policy i, a corresponding set of feedback control knobs is denoted as x_i, where $i = 1, \cdots, n$. We can formulate a constrained optimization problem as follows:

$$\begin{aligned} &\min_{x_1,\ldots,x_n} \quad f(x_1,\ldots,x_n) \\ &\text{subject to} \quad g_j(x_1,\ldots,x_n) \leq 0, \quad j = 1,\ldots,m, \end{aligned} \tag{7.20}$$

where f is the common objective function[2]; $g_j(\cdot)$, $j = 1,\ldots,m$ are the resource and performance constraints related to the application. Introducing Lagrange multipliers $\nu_1,\ldots,\nu_m$, the Lagrangian of the problem is given as:

$$\begin{aligned} L(x_1,\ldots,x_n,\nu_1,\ldots,\nu_m) = \quad & f(x_1,\ldots,x_n) + \\ & \nu_1 g_1(x_1,\ldots,x_n) + \\ & \ldots + \\ & \nu_m g_m(x_1,\ldots,x_n) \end{aligned} \tag{7.21}$$

2. Derivation of necessary conditions: Model inaccuracies (such as those in estimating actual computation times, exact energy consumption, or end-to-end delay in practical systems) are likely to render the expressions for functions $f(.)$ and $g_j(.)$ above inaccurate. Hence, we would like to combine optimization with feedback control to compensate for such inaccuracies.

Our approach is to derive only *approximate necessary conditions* for optimality, instead of exact necessary and sufficient conditions. This gives a locus of solution

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

points. A series of feedback loops is then used to traverse that locus in search of a maximum utility point.

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 $g(.)$ and $f(.)$ are differentiable), then using the Karush-Kuhn-Tucker (KKT) optimality conditions [5], $\forall i : 1,\ldots,n$:

$$\frac{\partial f(x_1,\ldots,x_n)}{\partial x_i} + \sum_{j=1}^{m} \nu_j \frac{\partial g_j(x_1,\ldots,x_n)}{\partial x_i} = 0 \tag{7.22}$$

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

$$\Gamma_{x_1} = ... = \Gamma_{x_n} \tag{7.23}$$

We then use a feedback control approach to find the maximum utility point on the locus that satisfies Equation (7.23). Our feedback control approach is described next. We find it useful for the discussion below to also define the average $\Gamma_x = (\Gamma_{x_1} + ... + \Gamma_{x_n})/n$. This average at time k will serve as the set point $r(k)$ for each individual Γ_{x_n}.

3. Feedback control: The purpose of feedback control is to find knob values x_i such that the condition in Equation (7.23) 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 the increase in utility (i.e., optimizes the objective function). Hence, we define the control error, $e(k)$, at time k, as $\Gamma_x - \Gamma_{x_i}$ (we omit index k for notational simplicity) and find the set of neighboring points to the current x_i vector that reduces the error in the direction that involves a maximum increase in utility. The algorithm will dynamically guide the system toward a better configuration.

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

1. Formulating the optimization problem: The decision variables in the optimization problem are the tuning knobs for each individual feedback control 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). They are optimized subject to resource and delay constraints. For simplicity, let us use a queuing-theoretic M/M/1 model for each server machine to predict delay. 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. Assuming a load-balanced tier i of m_i machines and of total arrival rate λ_i, the arrival rate per machine is λ_i/m_i and the service rate is proportional to frequency f_i. Expressing λ_i in clock cycles, the utilization of a machine at tier i, denoted U_i, becomes $U_i = \frac{\lambda_i}{m_i f_i}$, We further approximate power consumption P_i by a

function of CPU frequency f_i for each machine at tier i, namely $P_i(f_i) = A_i \cdot f_i^p + B_i$, where A_i and B_i are positive constants. In realistic systems p varies between 2.5 and 3 [12]. A_i, B_i, and p can be obtained by curve fitting against empirical measurements when profiling the system off-line. Merging the above two equations, we get $P_i(U_i, m_i) = A_i \cdot \left(\frac{\lambda_i}{U_i m_i}\right)^p + B_i = \frac{A_i \lambda_i^p}{U_i^p m_i^p} + B_i$. The total power consumption can be obtained by summing over N tiers as $P_{tot}(U_i, m_i) = \sum_{i=1}^{N} m_i \cdot P_i(U_i, m_i)$. 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, this translates to $\sum_{i=1}^{N} \frac{m_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} \leq K$, where K is some constant. 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:

$$\begin{aligned} \min_{U_i \geq 0,\ m_i \geq 0} \quad & P_{tot}(U_i, m_i) = \sum_{i=1}^{3} m_i \left(\frac{A_i \lambda_1^3}{U_i^3 m_i^3} + B_i \right) \\ \text{subject to} \quad & \sum_{i=1}^{3} \frac{m_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} \leq K, \\ & \sum_{i=1}^{3} m_i \leq M \end{aligned} \qquad (7.24)$$

2. Derivation of necessary conditions: To derive necessary conditions, let $x = \left[U_1\ U_2\ U_3\ m_1\ m_2\ m_3\right]^T$ be the vector of decision variables. Observe that we could have alternatively chosen frequency f_i instead of utilization U_i for the decision variables, since utilization cannot be set directly. Since we assume an algebraic relation between utilization and frequency, the two choices are mathematically equivalent. Expressing control knobs in terms of utilization could be more intuitive in that it directly quantifies a measure of server load. Introducing the Lagrange multipliers $\nu_1, \nu_2 \geq 0$, we can write the Lagrangian function as:

$$\begin{aligned} L(x, \nu_1, \nu_2) = & \sum_{i=1}^{3} m_i \left(\frac{A_i \lambda_i^3}{U_i^3 m_i^3} + B_i \right) + \\ & + \nu_1 \cdot \left(\sum_{i=1}^{3} \left(\frac{m_i}{\lambda_i} \cdot \frac{U_i}{1-U_i} \right) - K \right) + \nu_2 \cdot \left(\sum_{i=1}^{3} (m_i) - M \right). \end{aligned} \qquad (7.25)$$

The Karush-Kuhn-Tucker (KKT) conditions [5] associated with the optimization problem are:

$$\begin{aligned}
\frac{\partial L}{\partial U_i} &= -\frac{nA_i\lambda_i^3}{m_i^2U_i^4} + \frac{\nu_1 m_i}{\lambda_i(1-U_i)^2} = 0 \ \forall i,\\
\frac{\partial L}{\partial m_i} &= -\frac{2A_i\lambda_i^3}{m_i^3U_i^3} + B_i + \frac{\nu_1}{\lambda_i}\cdot\frac{U_i}{1-U_i} + \nu_2 = 0 \ \forall i,\\
\nu_1 \cdot \left(\sum_{i=1}^{3}(\frac{m_i}{\lambda_i}\cdot\frac{U_i}{1-U_i}) - K\right) &= 0,\\
\nu_2 \cdot \left(\sum_{i=1}^{3}(m_i) - M\right) &= 0.
\end{aligned} \tag{7.26}$$

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

$$\frac{\lambda_1^4(1-U_1)^2}{m_1^3U_1^4} = \frac{\lambda_2^4(1-U_2)^2}{m_2^3U_2^4} = \frac{\lambda_3^4(1-U_3)^2}{m_3^3U_3^4}. \tag{7.27}$$

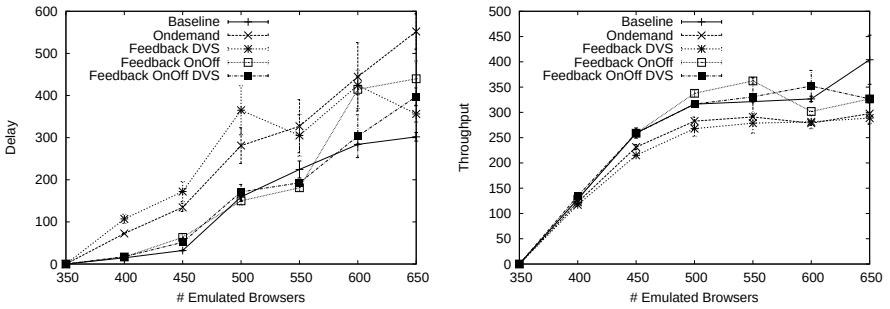
To simplify the notations, we will use $\Gamma(m_i,U_i)$ to denote $\frac{\lambda_i^4(1-U_i)^2}{m_i^3U_i^4}$ in the following discussions. Then the necessary condition for optimality is expressed as

$$\Gamma(m_1,U_1) = \Gamma(m_2,U_2) = \Gamma(m_3,U_3). \tag{7.28}$$

3. 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 can design a feedback loop in which the utilization and the number of machines are adjusted (using traditional control-theoretic analysis techniques described earlier in this tutorial) in the direction that reduces error (i.e., enforces Equation (7.28)) while minimizing the energy objective function.

7.6.2 Evaluation

Next, we evaluate five different energy saving approaches: a baseline (no power management), the Linux On-demand governor [25], and the three control algorithms mentioned above (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. 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). We used the shopping mix workload consisting of 80% browsing and 20% ordering, which is considered the primary performance metric by the Transactional Processing Council [30]. The user think time was set to 1.0 seconds. We used 450 ms as the delay set-point for all experi-



(a) Average Delay (b) Throughput

Fig. 7.12 Other Metrics: Average Delay, Deadline Miss Ratio, and Throughput

ments. 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. Figure 7.11 shows that our approach improves energy consumption (baseline and Linux governor are not shown). Figure 12(a) depicts the average delay of the five algorithms. Figure 12(b) depicts throughput.

7.7 Conclusions And Research Challenges

Current trends in computing systems are challenging our ability to engineer systems that adapt quickly to changes in workloads and resources. Examples addressed in this paper include: self-tuning memory management in database systems that adapts to changes in queries and disk contention, dynamic control of resources in real-time embedded systems that control variations in task resource demands to meet real time objectives, adapting CPU allocations of virtualized servers in data centers in response to variations in the user requests, and addressing interactions between control loops for power management in response to workload variations. Such adaptation is usually addressed by building a closed loop system that dynamically adjusts resource allocations and other factors based on measured outputs. Control theory provides a formal approach to designing closed loop systems that is used in many other fields such as mechanical engineering, electrical engineering, and economics.

This paper provides a brief introduction to key concepts and techniques in control theory that we have found valuable in the design of closed loops for computing systems. There has been considerable success to date with applying control theory to computing systems, including impact on commercial products from IBM, Hewlett Packard, and Microsoft. However, many research challenges remain. Among these are the following.

- Benchmarks for assessing closed designs. While there are well established benchmarks for steady state workloads of web servers, database systems, and other widely used applications, assessing the ability of closed loop systems to adapt to changes in workloads and resources requires the characterizations of transients. Examples of such characterizations include the magnitude of changes in arrival rates and/or service times, how quickly changes occur, and how long they persist. Further, we need efficient ways to generate such workload dynamics that permit the construction of low cost, low noise benchmarks. Good insights into workload characteristics will allow us to incorporate more sophisticated techniques, such as model based predictive control that is discussed in Section 4.
- Control patterns for software engineering. To make control design accessible to software practitioners, we need a set of "control patterns"' that provide a convenient way to engineer resource management solutions that have good control properties. By good control properties, we mean considerations such as the SASO properties (stability, accuracy, settling time, and overshoot) discussed in Section 2. Two starting points for such patterns are contained in this paper: self-tuning memory in Section 3, which shows how to use control theory to do load balancing, and the optimal design of interacting control loops in Section 6.
- Scalable control design for distributed systems. Traditionally, control engineering deals with complex systems by building a single Multiple Input, Multiple Output closed loop. This approach scales poorly for enterprise software systems because of the complexity and interactions of components. Helpful here are decomposition techniques such as those in Section 5 that address virtualized servers for enterprise computing.
- Analysis tools to address interactions between control loops. Feedback control introduces a degree of adaptive behavior into the system that complicates the construction of component based systems. Analysis tools are needed to understand and quantify the side-effects of interactions between individually well-optimized components, as well as any emergent behavior that results from component compositions.
- Dynamic verification of design assumptions. Feedback loops make assumptions about causal relations between systems variables, such as an admission controller assuming that request rate and utilization change in the same direction. There is considerable value in dynamically verifying design assumptions. For example, one could have a "performance assert" statement that tests that system variables change in the expected direction in relation to one another. When violations of these assumptions are detected, appropriate actions must be taken.
- Control of multiple types of resources. Most of the existing applications of control theory deal with one resource type, for instance, memory in Section 3, and CPU in Sections 4 and 5. In practice, the performance of applications running in computing systems depends on multiple resources, such as CPU, memory, network bandwidth and disk I/O. From a control perspective this creates challenges with interactions between multiple controllers and target systems with different time constants, delay characteristics, and software interfaces.

- Extending the application of control theory beyond performance management. While control theory provides a systematic approach to designing feedback systems for performance management, computing solutions also involve considerations such as user interface design, security, installation, and power. To what extent can feedback control be applied to these areas? Also, to what extent can other technologies, such as machine learning, be applied to performance management?

References

1. C. Amza, A. Ch, A. Cox, S. Elnikety, R. Gil, K. Rajamani, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of WWC-5: IEEE 5th Annual Workshop on Workload Characterization*, Oct. 2002.
2. K. Astrom. Challenges in Control Education. *Advances in Control Education*, 2006.
3. K. J. Astrom. *Introduction to Stochastic Control Theory*. Academic Press, 1970.
4. K. J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition, Jan. 1995.
5. D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1995.
6. R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–74, 2004.
7. C. Corp. XenServer.
8. H. P. Corporation. HP Integrity Essentials Global Workload Manager.
9. Y. Diao, J. L. Hellerstein, A. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the American Control Conference*, pages 2045–2050, June 2004.
10. Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *Proceedings of the Real-Time and Embedded Technology and Application Systems Symposium, Toronto, Canada*, pages 376–387, 2004.
11. Y. Diao, C. W. Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco. Comparative studies of load balancing with control and optimization techniques. In *Proceedings of the American Control Conference, Portland, OR*, pages 1484–1490, 2005.
12. E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Power Aware Computing Systems*, pages 179–196, 2002.
13. E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.
14. G. F. Franklin, J. D. Powell, and A. Emani-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley, Reading, Massachusetts, third edition, 1994.
15. G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, Reading, Massachusetts, third edition, 1998.
16. J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
17. C. V. Hollot, V. Misra, D. Towsley, and W. B. Gong. A control theoretic analysis of RED. In *Proceedings of IEEE INFOCOM*, pages 1510–1519, Anchorage, Alaska, Apr. 2001.
18. S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM*, pages 3–15, Sept. 1991.
19. X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proceedings of the IEEE Conference on Decision and Control*, Dec. 2007.

20. L. Ljung. *System Identification: Theory for the User*. Prentice Hall, Upper Saddle River, NJ, second edition, 1999.
21. C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Markley. Performance specifications and metrics for adaptive real-time systems. In *Proceedings of the IEEE Real Time Systems Symposium*, Orlando, 2000.
22. C. Lu, X. Wang, and X. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):550–561, 2005.
23. J. Maciejowski. *Predictive Control with Constraints*. Prentice Hall, 1 edition, 2002.
24. K. Ogata. *Modern Control Engineering*. Prentice Hall, 3rd edition, 1997.
25. V. Pallipadi and A. Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, 2006.
26. S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, J. Bigus, and T. S. Jayram. Using control theory to acheive service level objectives in performance management. *Real-time Systems Journal*, 23:127–141, 2002.
27. P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Sstems Principles*, pages 89–102, New York, NY, USA, 2001. ACM Press.
28. 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.
29. J.-J. E. Slotine and W. Li. *Applied Nonlinear Control*. Prentice-Hall, 1991.
30. Transaction Processing Performance Council. TPC Benchmark W (Web Commerce).
31. X. Wang, Y. Chen, C. Lu, and X. Koutsoukos. FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization control. *Journal of Systems and Software*, 80(7):938–950, 2007.
32. W. Xu, X. Zhu, S. Singhal, and Z. Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *Proceedings of the IEEE/IFIP Network Operations & Management Symposium*, Apr. 2006.
33. W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 149–163, New York, NY, USA, 2003. ACM Press.
34. X. Zhu, Z. Wang, and S. Singhal. Utility driven workload management using nested control design. In *Proceedings of the American Control Conference*, June 2006.