Managing Web server performance with AutoTune agents

by Y. Diao
J. L. Hellerstein
S. Parekh
J. P. Bigus

Managing the performance of e-commerce sites is challenging. Site content changes frequently, as do customer interests and business plans, contributing to dynamically varying workloads. To maintain good performance, system administrators must tune their information technology environment on an ongoing basis. Unfortunately, doing so requires considerable expertise and increases the total cost of system ownership. In this paper, we propose an agent-based solution that not only automates the ongoing system tuning but also automatically designs an appropriate tuning mechanism for the target system. We illustrate this in the context of managing a Web server. There we study the problem of controlling CPU and memory utilization of an Apache® Web server using the application-level tuning parameters MaxClients and KeepAlive, which are exposed by the server. Using the AutoTune agent framework under the Agent Building and Learning Environment (ABLE), we construct agents to fully automate a control-theoretic methodology that involves model building, controller design, and run-time feedback control. Specifically, we design (1) a modeling agent that builds a dynamic system model from the controlled server run data, (2) a controller design agent that uses optimal control theory to derive a feedback control algorithm customized to that server, and (3) a run-time control agent that deploys the feedback control algorithm in an on-line real-time environment to automatically manage the Web server. The designed autonomic feedback control system is able to handle the dynamic and interrelated dependencies between the tuning parameters and the performance metrics with guaranteed stability from control theory. The effectiveness of the AutoTune agents is demonstrated through experiments involving variations in workload, server capacity, and business objectives. The results also serve as a validation of the ABLE toolkit and the AutoTune agent framework.

The increasing complexity of computing systems and applications demands a correspondingly larger human effort for system configuration and performance management. This manual effort can be time-consuming and error-prone, and requires highly skilled personnel, making it costly. Autonomic computing[1] uses the analogy of the human autonomic nervous system to suggest the use of a higher level of automation and self-management capability in computing systems.

The complexity and importance of developing autonomic computing systems has attracted research

0018-8670/03/$5.00 © 2003 IBM

efforts of a theoretical as well as an applied nature. In particular, control theory is emerging as a promising cornerstone to provide a rigorous mathematical foundation for designing and analyzing autonomic controllers, which can reduce the work of system administrators and provide guaranteed control performance. Some applications of control theory to computing systems include flow and congestion control,[2–5] differentiated caching and Web service,[6,7] multimedia streaming,[8] Web server performance,[9] e-mail server control,[10,11] and distributed resource allocation on the grid.[12] These applications all provide a degree of autonomic behavior by providing algorithms to automatically control some aspect of a computing system's operation. However, a common theme in all this work is that applying control techniques requires significant modeling and design work, which is typically a manual process conducted by the system designer. In the spirit of reducing all manual intervention, we propose to automate this phase of the deployment of control systems as well.

In this paper, we describe an agent-based autonomic feedback control system that uses high-level inputs from the human system administrators to not only control a computing system, but also to automatically design a controller suitable for that system. We illustrate this in the context of an Apache** Web server. Using the Agent Building and Learning Environment (ABLE),[13] AutoTune[14] agents are built for (1) modeling the behavior of an Apache Web server, (2) designing the feedback control law, and (3) in on-line operation, adjusting the server parameters in response to workload variations. These agents cooperate at different phases of the life cycle of the autonomic control system to achieve the function of automatically controlling the Web server.

The remainder of the paper is organized as follows. The next section describes the background of server self-tuning in the context of Apache Web servers. The section "Server self-tuning with AutoTune agents" introduces ABLE-based AutoTune agents and details the architecture and algorithms used to automate server tuning. The experimental results are described in the section "Experimental assessment," comparing the performance of the Apache server controlled by the proposed AutoTune controller and a heuristic manual controller, particularly when significant workload variations exist. Finally, our conclusions are presented.

Apache Web server and performance tuning

The Apache[15] Web server is the most popular Web server in use today,[16] making resource management for such a server an important problem. Version 1.3.x of the server on UNIX** is structured as a master process and a pool of worker processes. The master process monitors the health of the worker processes and manages their creation and destruction. The worker processes are responsible for communicating with Web clients and generating responses, and one worker process can handle at most one connection at a time. The number of worker processes is limited by the parameter MaxClients, thereby throttling the Web server's throughput. Worker processes cycle through three states: idle, waiting, and busy. A worker is in an idle state if no Transmission Control Protocol (TCP) connection from the client has been made to it.

Once a TCP connection is accepted, the worker process is either waiting for a HyperText Transfer Protocol (HTTP) request from the client, or is busy in processing the client request. According to persistent connections in HTTP/1.1,[17] the established TCP connection remains open between consecutive HTTP requests (which eliminates the overhead for setting up one connection for each request as in HTTP/1.0). This persistent connection can either be terminated by the client or by the master process, if the waiting time of a worker process exceeds the maximum allowed time specified by the parameter KeepAlive.

The performance of the Apache Web server can be measured by different metrics, such as end-user response times or utilization of various resources on the server. Selection of appropriate performance metrics depends not only on management objectives but also on metric availability. From the point of view of guaranteeing quality of service, bounding the end-user response times is desired. However, end-user response time is a client-side metric, and additional instrumentation such as a probing station needs to be added. This would increase server load and raise other issues, including accuracy and recentness of the available measurements. (Refer to Reference 18 for a discussion of and control strategies for managing the end-user response time.) In this paper, we quantify the server performance using server-side metrics, CPU utilization and memory utilization, which are easy to measure on the server and associated with business needs as well.

System administrators typically maintain a certain utilization level on the server, high enough to efficiently utilize system resources but not so high that it causes thrashing and failures as a result of overutilization. Good end-user response times are ensured by reserving sufficient capacity to handle workload surges. However, the system utilization cannot be directly set in an Apache server. Instead, the administrators must operate indirectly by adjusting certain tuning parameters, among which MaxClients and KeepAlive are commonly used. A higher MaxClients value allows the Apache server to process more client requests, and increases both CPU and memory utilizations. Also, decreasing the value of KeepAlive potentially allows worker processes to be more active, which directly results in higher CPU utilization and indirectly increases memory utilization, since more clients can connect to the server.[9]

In principle, the desired and feasible CPU and memory utilizations can be achieved by properly selecting the tuning parameters MaxClients and KeepAlive, but in practice, it is time-consuming, error-prone, and skills-intensive to adjust these parameters manually. Moreover, this tuning work has to be repeated as the workload changes or the server is reconfigured for more CPU and memory. A change of Web site contents may also affect the CPU and memory usage per request and can also require different MaxClients and KeepAlive settings. We illustrate the drawbacks of manual tuning using modified versions of our agents and the testbed, both of which are described later. In essence, our modifications to the Apache server allow us to change the MaxClients and KeepAlive values without restarting the server, and the agents provide the graphical user interface (GUI) for manually setting the values.

Suppose the administrator wants to have the desired CPU level at 0.5 and memory at 0.6. Manually tuning the Apache server is a trial-and-error process and can be quite time-consuming. Due to the interrelationships between the tuning parameters and performance metrics, it may not be easy to find the proper KeepAlive and MaxClients settings.

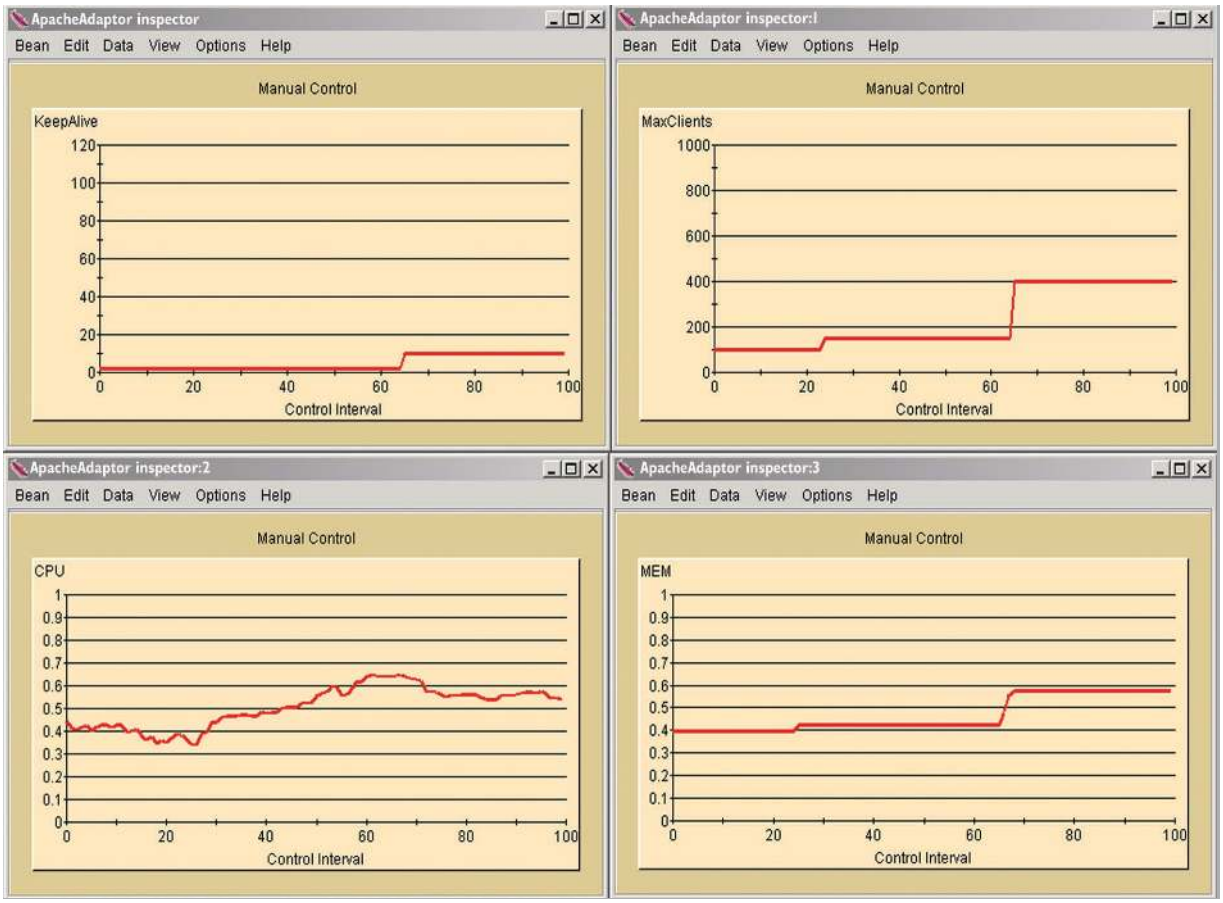
In Figure 1, the values for the tuning parameters KeepAlive and MaxClients are shown in the top two Inspector windows, and the bottom two Inspector windows show the corresponding effects on the performance metrics CPU and memory utilization. The system is running in our testbed and is subjected to a synthetic workload, both of which are described in the section "Apache testbed and workload generator." The y-axis shows the measured values, and the x-axis indicates the time, which is measured by control intervals (i.e., sampling intervals). The control interval is five seconds; every five seconds the tuning parameters (if they are changed) are sent to the Apache server, and the CPU and memory values of the Apache server are also provided to the Apache adaptor for the system administrator to check the control results.

As Figure 1 indicates, arbitrarily selecting KeepAlive = 2 and MaxClients = 100 will not yield CPU and memory utilization close to the desired values. The tuning parameters and the performance metrics are interrelated; for instance, increasing MaxClients to 150 causes increases in both CPU and memory utilizations. Manually tuning the Apache Web server using these controls is possible by using the following heuristic. If we increase MaxClients, both CPU and memory utilization will increase. If we increase KeepAlive, CPU utilization will decrease. Thus, we can use MaxClients to adjust utilization until the memory is at the desired level, and then get the desired CPU utilization by adjusting KeepAlive. Using these tuning heuristics, in order to achieve CPU = 0.5 and MEM = 0.6, after several tries the values MaxClients = 400 and KeepAlive = 10 are found, which drive CPU and memory close to the desired values.

As mentioned earlier, a variation in workloads may affect this relationship between the tuning parameters and system utilizations, and thus change the "optimal" values for KeepAlive and MaxClients. For example, as shown in Figure 2, when additional workload (requests for dynamic Web pages) is included around the twentieth control interval, we observe that the previous setting of tuning parameters causes both CPU and memory utilizations to deviate from the desired levels. Therefore, the system administrator will need to tune the server parameters again, and this can become quite tedious if the workload is changing frequently.

It is worth mentioning that in this paper we use a desired CPU level of 0.5 and memory of 0.6 simply for the purpose of illustrating the tuning process. How to choose the desired utilization level practically depends on business needs and system configurations, and thus is out of the scope of this paper. In this paper, we focus on how to automatically construct quantitative models and conduct tuning control. This is important, since the relationship between the performance metrics and tuning parameters is

Figure 1 Results of manually tuning the Apache Web server



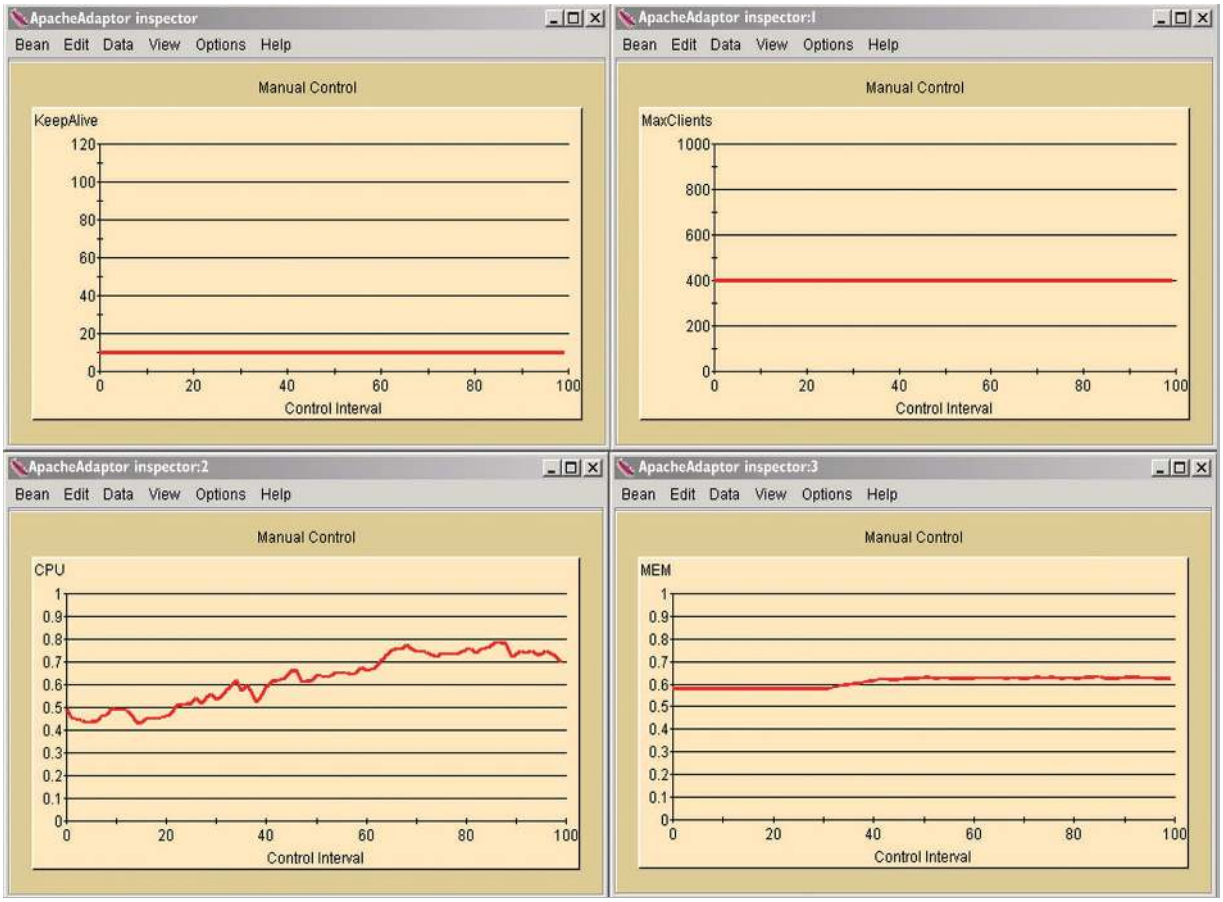
only qualitatively known, and extensive manual tuning is required and needs to be repeated when workload changes. In other words, the proposed AutoTune agents increase the automation level of server tuning by automating the lower-level tuning (e.g., adjusting server parameters MaxClients and KeepAlive). This will not eliminate the role of the system administrators, but can reduce their work and help them focus on higher-level performance metric goals (e.g., desired CPU and memory utilization).

Server self-tuning with AutoTune agents

Based on the previous section, it is clearly desirable to automate the adjustment of the MaxClients and KeepAlive values, both at system startup and on an ongoing basis in response to changing workload. More broadly, we would like to reduce the work of the system administrator by performing such tuning work with an autonomic agent. Rather than construct a special-case agent for this particular set of tuning controls and this particular software system, we propose a more general architecture that can be applied to many other systems as well. We use the Apache server as an illustration for this general framework.

Our solution consists of multiple agents that automate the entire methodology of controller design and also perform the on-line system control. These agents are implemented using the Agent Building and Learning Environment (ABLE), and in particular by implementing specialized AutoTune agents. Below, we present an overview of ABLE and the basic AutoTune architecture, followed by a description of our agents and the control-theory-based algorithms that

Figure 2 Effects of dynamic workloads on manually tuned Apache server

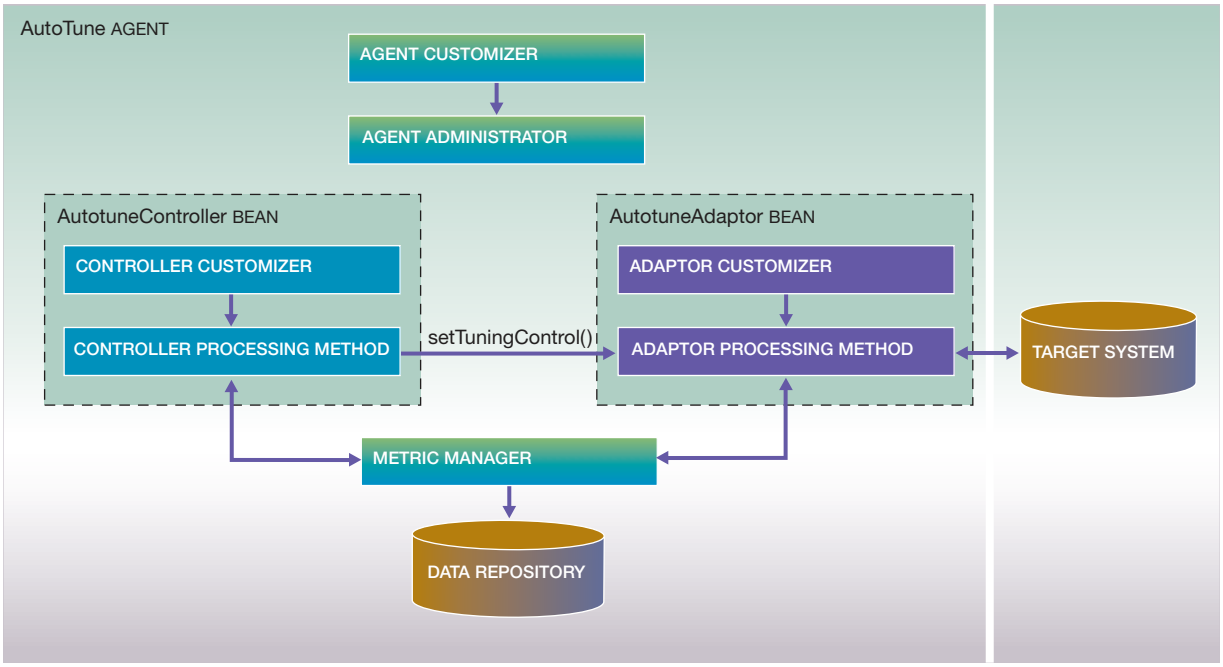


are essential to exploit the learning and control abilities of the autonomic agents.

ABLE framework and base AutoTune agent. The Agent Building and Learning Environment (ABLE) is a Java**-based toolkit for developing and deploying hybrid intelligent agent applications.[19] It provides a comprehensive library of intelligent reasoning and learning components packaged as Java beans (known as AbleBeans) and a lightweight Java agent framework to construct intelligent agents (known as AbleAgents). The AbleBean Java interface defines a set of common attributes (name, comment, state, etc.) and behaviors (standard processing methods such as init( ), reset( ), process( ), and quit( )), which allows AbleBeans to be connected to form AbleAgents. A Java Swing-based GUI, AbleEditor, is also provided for creating and configuring AbleBeans, and for constructing and testing the AbleAgents built from them. For most AbleBeans, the user-interface is through a GUI component known as a "Customizer" that allows the user to set and view parameters related to the bean.

The base AutoTune agent is a function-specific AbleAgent for autonomic computing. Inspired by human biology, the AutoTune agent is based on an architecture that combines several elements that are useful in building systems to react to a dynamic environment. The AutoTune agent contains two basic building blocks (AbleBeans): the AutotuneController bean and the AutotuneAdaptor bean, as shown in Figure 3. The AutotuneController bean defines control strategies (such as learning the behavior of the target system or providing actions to amend abnormal situations). Its Customizer GUI allows the

Figure 3 Architecture of the base AutoTune agent



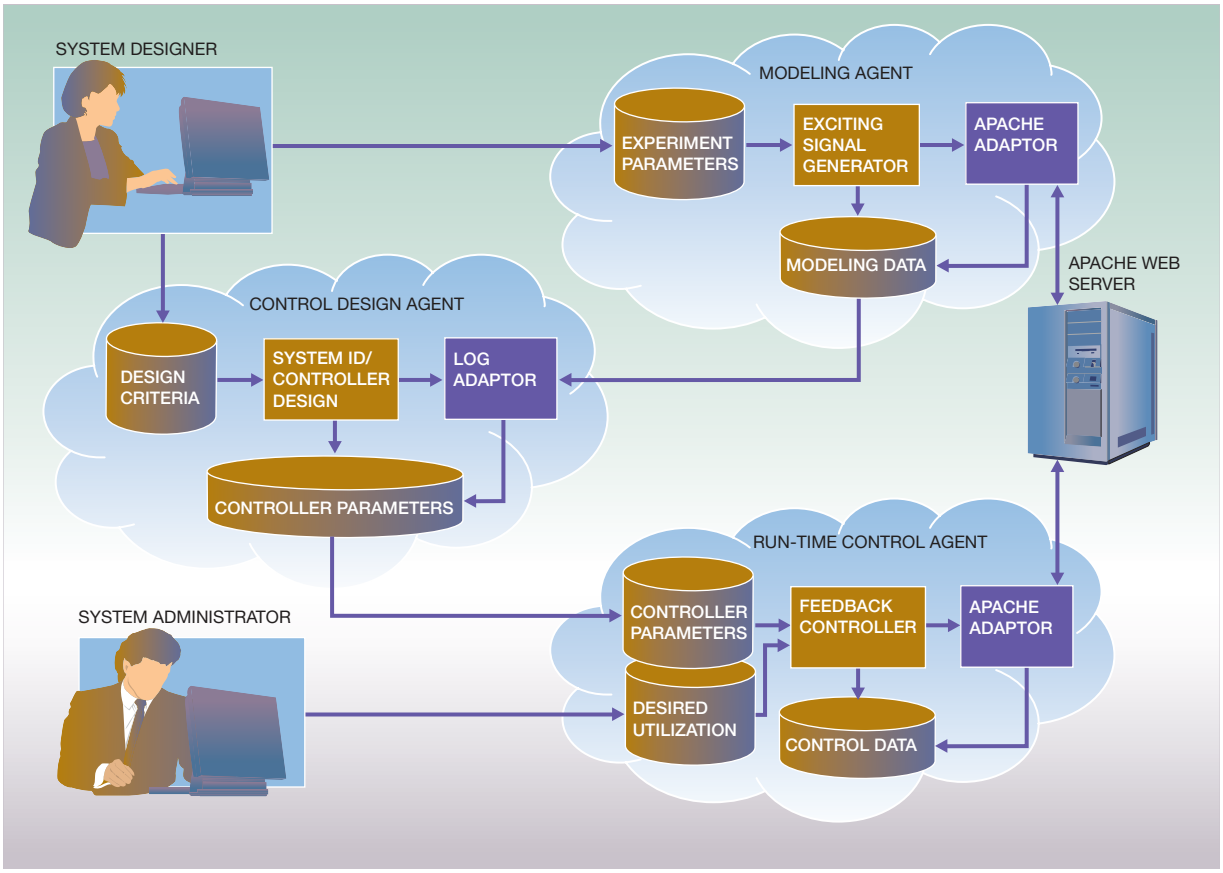
system administrator to configure the control strategy in advance or on the fly. The AutotuneAdaptor bean interfaces with the target system to get the service level metrics and to set the tuning parameters. Another Customizer GUI is provided for the system administrator to manually set the tuning parameters (for example, for testing purposes, or when the AutotuneController is inactive). This decoupling allows the same AutotuneController to be used with a variety of systems, simply by choosing the appropriate AutotuneAdaptor to interface with the respective system.

The execution of the AutotuneController and AutotuneAdaptor beans is managed by the AutoTune agent through the Agent Administrator. In particular, the Agent Administrator handles the timer facility and asynchronous event processing function that allow the AutotuneController and AutotuneAdaptor to run autonomously, by periodically processing control functions and communicating with the target system. The AutoTune agent-level Customizer allows the system administrator to separately set the control interval (used by the AutotuneController bean) and sample interval (used by the AutotuneAdaptor bean), which can be different from each other. A set of AutotuneMetric classes is defined to represent the state/performance of the target system (service-level metrics), the tuning parameters of the target system (tuning control metrics), and the parameters of the control strategy (configuration metrics). These metrics can be read or written by the AutotuneController or AutotuneAdaptor. They are managed as a collection by the AutoTune Metric Manager for interactions between the two component beans, and can also be selectively saved to a historical data repository.

ABLE and the AutoTune architecture were designed to be extensible and to allow rapid deployment of agent-based solutions. Our experience with using this infrastructure validates the usefulness of this architecture and the ABLE toolkit.

AutoTune agents for server self-tuning. For automatic tuning of server configuration parameters, an algorithm is needed. A complete self-tuning solution both automates the algorithm design and includes an executable on-line algorithm for doing the param-

Figure 4 Architecture of the AutoTune agents



eter adjustments. We follow a standard control-theoretic methodology for the design process, consisting of (1) system modeling and (2) controller design. Thus, our solution consists of three AutoTune agents that implement the phases of automatic feedback controller design and deployment: modeling, controller design, and run-time control, as shown in Figure 4. The modeling and design phases are performed in a "testing" (or nonproduction) mode, whereas the run-time control is active when the system is "live" (or in production mode).

In order to gain understanding of the dynamic behavior of the server, a modeling agent is first applied. The modeling agent varies the tuning parameters MaxClients and KeepAlive of the Apache Web server and records the resulting server performance metrics (CPU and memory utilization). The collected time-series data contain information about the dynamic relationship between the tuning parameters and the performance metrics. Using this modeling data, a first-order dynamic model is automatically built by utilizing system identification techniques (see the section "Modeling agent," below).

The system model is passed to the controller design agent to conduct model-based controller design. Based on this model, a linear quadratic regulation (LQR) controller is synthesized using design criteria that are specified through the Customizer GUI. The output of the controller design agent is a set of controller parameters that are passed to the run-time control agent.

These controller parameters are used by the run-time control agent to dynamically adjust the MaxClients and KeepAlive tuning parameters to achieve the de-

sired utilization levels. These desired levels are specified by the system administrator (through the runtime agent's Customizer GUI).

Modeling agent. A good design for the feedback controller relies on a mathematical model of the target system. One approach to building this model is to use first principles (e.g., creating a queuing model of the Apache server). However, this approach requires considerable sophistication as well as detailed knowledge about the inner working mechanism of the server. Instead of proceeding from first principles, an empirical approach is taken in the modeling agent for quantifying the relationship between the tuning parameters and performance metrics. This empirical modeling approach is referred to as the system identification technique,[20] also known as the "black box" approach, and is particularly suitable for use with autonomic agents. Since the model is built from the collected input/output data without requiring knowledge of the internals of the target system (i.e., the Apache server), this technique is easily applied to a wide variety of systems.

To perform system identification, the tuning parameters are varied in a deliberate, predetermined pattern so that the dynamics of the systems (i.e., the relationship between the tuning parameters and system utilization) are "excited" and represented in the input/output data pairs. This means that the parameter variations should satisfy two properties: uniform coverage (covering the space of possible variation of tuning parameters) and persistent excitation (containing enough frequency components to excite all of the system modes).[20] Our experience shows that a first-order model constructed by varying the tuning parameters in a (discrete) sinusoid pattern is usually sufficient for modeling the dynamics of queuing-based computing systems. Note that sine wave signals are used because any signal can be approximated with a combination of sine waves of different magnitude and frequency. Also note that in order to build the model correctly, the number of frequencies in the exciting signal should be at least equal to the order of the model (this is referred to as "persistent excitation" in control theory, and the necessity of exciting all the modes and dynamic behaviors of the system has been proven[20]).

We can build the system model by fitting the input/output data into the following model form:

$$\begin{bmatrix} \mathrm{CPU}_{k+1} \\ \mathrm{MEM}_{k+1} \end{bmatrix} = A \cdot \begin{bmatrix} \mathrm{CPU}_{k} \\ \mathrm{MEM}_{k} \end{bmatrix} + B \cdot \begin{bmatrix} \mathrm{KeepAlive}_{k} \\ \mathrm{MaxClients}_{k} \end{bmatrix} \quad (1)$$

where CPU_k and MEM_k denote the values of CPU and memory utilizations at the k-th time interval, and $\mathrm{KeepAlive}_k$ and $\mathrm{MaxClients}_k$ denote the values of KeepAlive and MaxClients at the k-th time interval. The 2×2 matrices A and B include modeling parameters and can be identified using the least squares method.[20] Note that this model is a linear model and, moreover, that the modeling work is only conducted once for a representative workload. Whereas the assumption of linearity can affect the model's accuracy, using linear models reduces modeling complexity and is easier to extend for modeling high-dimension systems with more metrics. Moreover, using linear models also facilitates the design of robust controllers, which can better tolerate model inaccuracy (as shown in the section "Experimental assessment"), so that there is no need to rebuild the model once the workload changes.

The above modeling procedure is wrapped into the modeling agent for automatically building the system model. The user is only required to answer two system-related questions (via the Customizer GUI): (1) the effective ranges of the tuning parameters, and (2) the maximum delay required for the tuning parameters to take full effect on the performance metrics. Generally, the effective range is known from server parameter definitions. The maximum delay refers to how long it will take for the performance metrics to reach the new steady state if the tuning parameters are changing from one end of the effective range to the other end. This can be easily measured by conducting a test run. The answers to these two questions are used to determine the magnitudes and time periods of the sinusoid variation pattern. The magnitudes are chosen to cover the effective ranges of the tuning parameters, and the time periods are chosen to be greater than twice the maximum delay, to ensure the coverage of the tuning parameter space. The answers to these questions need not be very accurate in order to get a good model. This is because the model's inaccuracy can be compensated for by a good feedback control scheme, which will be discussed in the next section. The modeling agent will automatically run the target system (the Apache server in our case) under a given workload, vary the parameters according to this pattern, collect the performance metrics from the server, and construct the model from the data.

Due to the process limit of the operating system of our server, MaxClients can only take an integer value in the range of [1, 1024]. The tuning parameter KeepAlive is in integral seconds with a minimum of 1 second. No maximum value is enforced for KeepAlive, but KeepAlive values larger than 50 have very small effects on system utilization. Thus, the range of Keep Alive can be specified as [1, 50]. The maximum time for the tuning parameters to take effect is roughly estimated from test run results; we use 10 minutes for MaxClients and 20 minutes for KeepAlive. Since the Apache server model is a second-order model, two sine waves are constructed for varying these parameters.

Run-time control agent. The run-time control agent implements a state feedback controller[21] that makes control decisions based on feedback of error, which is defined as the difference between the desired and measured system utilization. The accumulated error over previous time intervals is also used to increase the robustness of the controller. The feedback control law is of the following form:

$$\begin{bmatrix} \text{KeepAlive}_k \\ \text{MaxClients}_k \end{bmatrix} = K_P \cdot \begin{bmatrix} e_{\text{CPU},k} \\ e_{\text{MEM},k} \end{bmatrix} + K_I \cdot \sum_{j=1}^{k-1} \begin{bmatrix} e_{\text{CPU},j} \\ e_{\text{MEM},j} \end{bmatrix} \tag{2}$$

where $e_{\text{CPU},k}$ and $e_{\text{MEM},k}$ are the differences between the desired CPU and memory utilizations and the measured values at the k-th time interval. This control law is characterized by the controller parameters, the 2×2 matrices K_P (proportional control gain for fast response) and K_I (integral control gain for removing steady-state error). These matrices are automatically derived by the controller design agent discussed in the next section. The performance of the controller, such as the settling time (the time to recover from a disturbance) can be determined analytically from the closed loop system model (a model composed of both the system model generated by the modeling agent and the feedback control law).[21]

Controller design agent. The controller design agent uses the linear quadratic regulator approach from optimal control theory[22] to design the parameters K_P and K_I of the run-time control agent. This design relies on the system model obtained from the modeling agent. In particular, the controller design agent chooses the controller parameters based on minimizing the following quadratic cost function,

$$J(K_P, K_I) = \sum_{k=1}^{\infty} [e_{\text{CPU},k}\ e_{\text{MEM},k}\ v_{\text{CPU},k}\ v_{\text{MEM},k}] \cdot Q \cdot \begin{bmatrix} e_{\text{CPU},k} \\ e_{\text{MEM},k} \\ v_{\text{CPU},k} \\ v_{\text{MEM},k} \end{bmatrix} + [\text{KeepAlive}_k\ \text{MaxClients}_k] \cdot R \cdot \begin{bmatrix} \text{KeepAlive}_k \\ \text{MaxClients}_k \end{bmatrix}$$

subject to the dynamic system model in Equation 1 and the control law in Equation 2. The variables $v_{\text{CPU},k}$ and $v_{\text{MEM},k}$ are defined as the accumulated error of CPU ($v_{\text{CPU},k} = \sum_{j=1}^{k-1} e_{\text{CPU},j}$) and memory ($v_{\text{MEM},k} = \sum_{j=1}^{k-1} e_{\text{MEM},j}$) utilizations, respectively. A numerical algorithm for solving the Riccati equation is included in the controller design agent, and it allows us to compute the "optimal" K_P and K_I that minimize the above cost function. This raises the question as to what values to use for the weighting matrices Q and R of the above cost function. In principle, Q and R perform some scaling functions in addition to determining a trade-off between control error and control variability.

Since it is not very intuitive or user-friendly to have the system designer specify these Q and R matrices directly, we use some more meaningful inputs to derive the proper Q and R matrices. The system designer is required only to answer two system-related questions through the Customizer GUI: (1) what are the valid ranges for the tuning parameters and performance metrics (e.g., [0, 1] for CPU and memory, [1, 50] for KeepAlive, and [1, 1024] for MaxClients), and (2) what are the ranges of random fluctuations for the performance metrics (e.g., 10 percent for CPU and 2 percent for memory). Like the two questions asked for the modeling agents, the answers to the above two questions need not be very accurate, due to the robustness of the controller. We use the following heuristics to determine Q and R.

First, it is sufficient to use only the diagonal forms of the 4×4 weighting matrix Q and the 2×2 matrix R to define the above trade-off; that is $Q = diag(q_1, q_2, q_3, q_4)$ and $R = diag(r_1, r_2)$. Second, specify the error weights q_1, q_2 and tuning parameter weights r_1, r_2 in such a way that the errors $e_{\text{CPU},k}$ and $e_{\text{MEM},k}$ and tuning parameters (KeepAlive_k and MaxClients_k) have the same order of magnitude in the cost function. For example, since the ranges for CPU and MEM are both [0, 1], for KeepAlive [1, 50],

and for MaxClients [1, 1024], we choose $q_1 = 1, q_2 = 2, r_1 = 1/50^2$ and $r_2 = 1/1000^2$. Third, determine the accumulated error weights q_3, q_4 according to the variability of the performance metrics: the higher the variability of the metrics, the lower the accuracy of their control effects. Since the CPU utilization usually has 10 percent random fluctuation and the fluctuation for the memory utilization is usually quite small (less than 2 percent), we choose $q_3 = 1/10^2$ and $q_4 = 1/2^2$. Although these rules of thumb for determining Q and R may be a little involved, we have wrapped them into the controller design agent, so that the system designers are not exposed to them.

Implementation of self-tuning agents. Note that all of the above three agents have the same basic structure, and that they are constructed by extending the base AutoTune agent discussed earlier. In the modeling agent, the exciting signal generator bean is extended from the AutotuneController bean in Figure 3, by overriding the process( ) method to implement the specific exciting signal function and the system identification function (discussed in the section "Modeling agent"). Similarly, the Apache Adaptor bean is an extension of the AutotuneAdaptor bean, and it implements the socket connection (see the section "Apache testbed and workload generator") with the Apache Web server being used both for setting the tuning parameters and getting the performance metrics. We use the Customizer GUI facility to allow the system designer to specify the experimental parameters (for details, see the section "Modeling agent"). The run-time control agent has its own extension of the AutotuneController and a correspondingly different Customizer GUI. This agent can, in fact, reuse the exact same ApacheAdaptor bean for communicating with the Apache server. For the controller design agent, the AutotuneController implements the design automation, and it uses a different Adaptor to read the model parameters from the modeling agent.

The ABLE framework and Autotune Agent architecture have thus proved to be general and sufficient to have allowed us to easily construct all three agents from the same framework. Further, by allowing reuse of the ApacheAdaptor bean between the modeling and the run-time agents, it reduces the amount of work that needs to be done. The server self-tuning agents increase the automation level of server tuning and require minimal human intervention, from the system designer or from the system administrator. The system designer must provide some high-level information to guide the design process: the experiment parameters and the controller design criteria. The system administrator needs only to specify the control objective (the desired system utilization levels). Once this information is given, the agents can automatically build the system model, design the feedback controller, and apply the controller to dynamically tune the server.

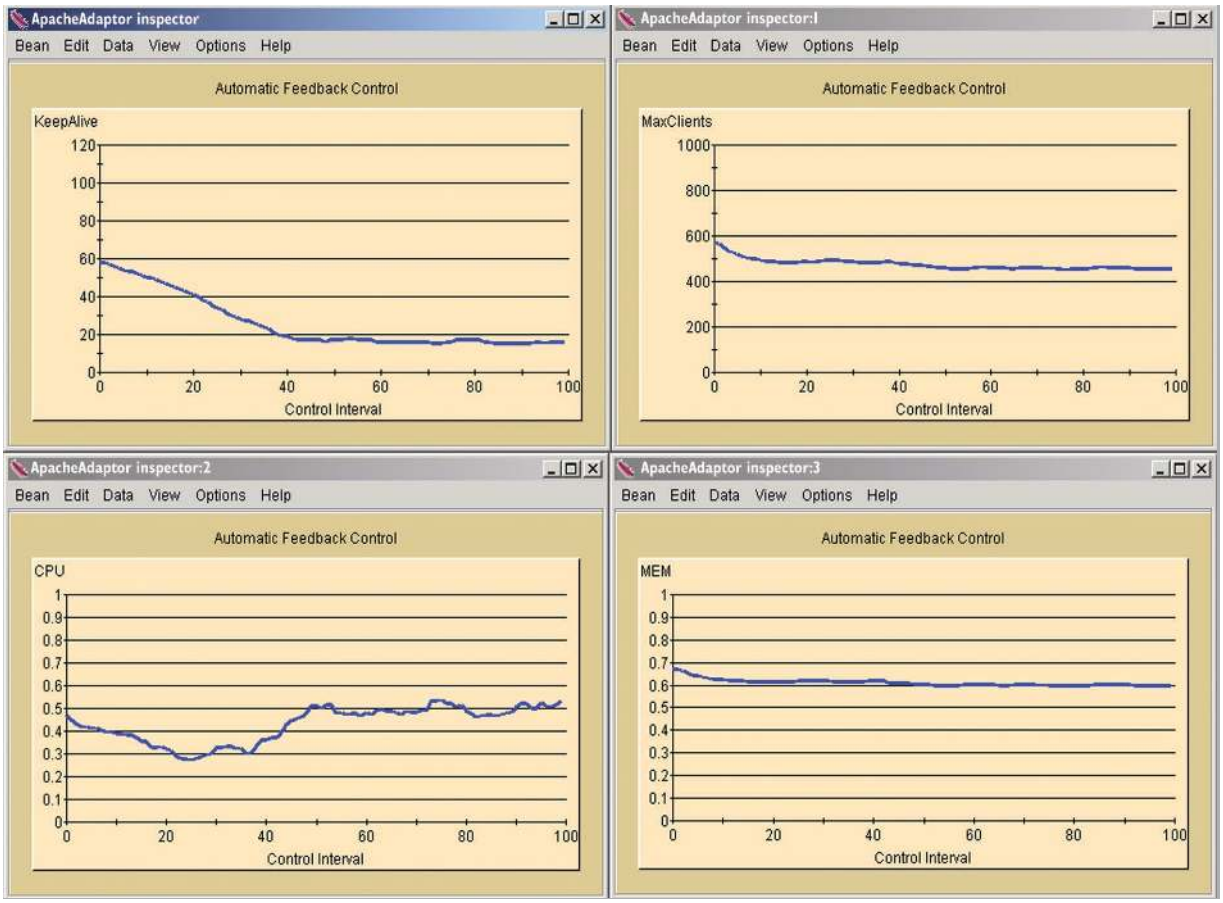
Experimental assessment

In this section, we describe our testbed and synthetic workload generator and present our experimental results.

Apache testbed and workload generator. Our Apache testbed consists of one server machine running Linux** (kernel v2.2.16) and the Apache HTTP server v1.3.19, and one or more client machines running synthetic workload generators. The Apache server code was modified in order to enable dynamic control. In particular, the tuning parameters MaxClients and KeepAlive were moved from the static variables (read from a configuration file at startup) into the in-memory scoreboard so that they could be modified on the fly and picked up by the master process. Further, a metric access functionality was implemented as a separate process that communicates with other Apache processes through an in-memory scoreboard (to minimize Apache source code changes and performance overheads). Also, a GET/SET interface over a socket connection was added to receive the tuning parameters and send performance metrics to the external autonomic agents (so as not to compete with normal HTTP traffic).

The synthetic workload generator simulates the activity of many clients. The workload model used to generate synthetic transactions was based on the WAGON (Web trAffic GeneratOr and beNchmark) model,[23] which has been validated in extensive studies of production Web servers. The "httperf" program[24] was used to generate synthetic HTTP requests that conform to this model. The Web site file access distributions are from the Webstone 2.5 reference benchmark.[25] Both static and dynamic workloads were used. The static workload clients requested static Web pages, and the session arrivals followed a Poisson distribution with a rate of 20 sessions per second. For the dynamic workload, the clients requested dynamic Web pages generated through CGI (Common Gateway Interface), and the session arrivals also followed a Poisson distribution but with a rate of 10 sessions per second. (A detailed descrip-

Figure 5 Results of autonomically tuning the Apache Web server

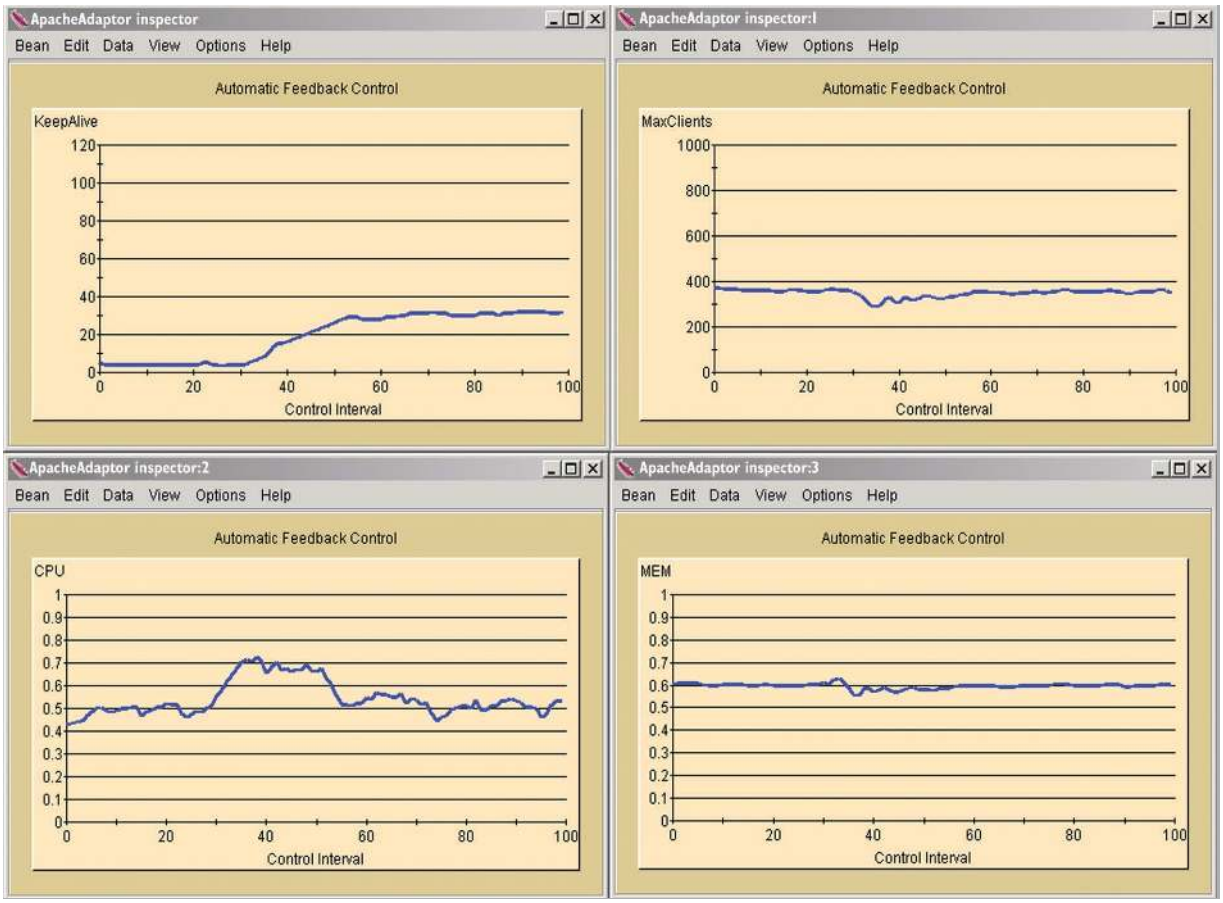


tion of the Apache testbed and workload generator can be found in Reference 9.)

Experimental results. We assess the performance of the autonomic control system by comparing the performance achieved through automation with that achieved by manual tuning as presented in the section "Apache Web server and performance tuning." We use different workload scenarios to validate the tuning results against workload variations.

First, the modeling agent was deployed under the static workload to build a dynamic system model. This was followed by running the controller design agent to come up with K_P and K_I for the run-time controller. Finally, the run-time feedback controller agent was started to dynamically tune the server parameters. The control interval (adaptation period) was 5 seconds. Note that the adaptation period was much shorter than the maximum times (i.e., 10 minutes for MaxClients and 20 minutes for KeepAlive). Usually, this parameter should be smaller than one-twentieth of the maximum time, so that the system dynamics can be captured and the controller can have time to react. Here, we made it even smaller, in order to achieve faster tuning. Experiments have shown that changing KeepAlive does not introduce any performance overhead, but changing MaxClients will cause a slight increase in CPU utilization, due to the overhead of adding or removing worker processes. However, these overheads are negligible, especially when the change in MaxClients is not large (smaller than 100). This is usually the case, and is guaranteed by putting a bound on the controller output. In addition, from the perspective of system stability, having a small control interval will not lead to an

Figure 6 Performance of the AutoTune controller for the Apache Web server under dynamic workloads



oscillatory response or cause instability problems because each adaptation step (the change of tuning parameters) is calculated based on the model, which is also obtained at the same control interval, and because the controller is designed to guarantee the stability of the adaptation and optimize the adaptation performance.

In Figure 5, we show the results of using the run-time control agent. The tuning parameters start at MaxClients = 600 and KeepAlive = 60. Without human intervention, it takes around 50 control intervals for the tuning parameters to converge to the values which result in 0.5 CPU utilization and 0.6 memory utilization.

The robustness and autonomy of the feedback control system allows the run-time control agent to adapt to different workloads without having to rerun the modeling and controller design agents to come up with new run-time controller parameters. Figure 6 shows the scenario where the same run-time control agent as in Figure 5 is used, but now the dynamic workload is added around the 20th control interval (similar to Figure 2). The added HTTP requests for dynamic Web pages consume more system resources, causing large increases in CPU utilization, and slight increases in memory utilization as well. This results in differences between the desired and observed utilizations and causes the run-time control agent to start changing the tuning parameters to compensate. In particular, a larger KeepAlive value is used to decrease the CPU level and the MaxClients value is adjusted temporarily according to the dynamics of the server. The CPU and memory utilizations come back to the desired values after 20 control intervals. (The

performance of the autonomic feedback control system has also been validated through some other experiments such as using different workload patterns, with different Web server configurations, and having different CPU and memory objectives; however, in the interest of brevity we do not include those results here.)

Conclusions

Complex information technology (IT) systems that require manual intervention for configuration and tuning increase the cost of ownership. Specifically, managing the performance of e-commerce sites is challenging, especially with dynamically varying workloads. To maintain good performance, system administrators must tune their IT environment on an ongoing basis. The autonomic computing drive aims to reduce costs by increasing the level of automation for such tasks, thereby reducing the manual intervention required.

In this paper, we have proposed an agent-based solution for not only automating the ongoing system tuning but also for automatically designing an appropriate tuning mechanism for the target system. We use the ABLE toolkit and the AutoTune agent framework to facilitate the construction of autonomous agents for autonomic performance management. These agents automate a control-theoretic methodology of controller design, that is, the AutoTune agents automate the procedure of model building, controller design, and run-time feedback control. We described the methodology used to design a model-based feedback controller, which can handle the dynamic and interrelated dependencies between tuning knobs and performance metrics. The effectiveness of the design automation as well as the resulting tuning mechanism has been demonstrated through experiments showing the feedback-driven controller to be robust and adaptable to situations other than the one for which it was designed. Our system thus allows a system administrator to automate the process of designing a tuning mechanism and provides a readily available run-time agent to perform the real-time system monitoring and tuning. Thus, it is clearly preferable to the manual tuning approach that is commonly used today.

The component-based toolkit of ABLE and the general framework provided by the AutoTune agent architecture were invaluable in allowing rapid construction of these three agents. In the future, we look to leverage this plug-and-play capability to automate tuning for a variety of systems, ranging from single database and application servers to distributed server farms. We would like to verify that the automated methodology we have incorporated in these agents is indeed applicable across this wide variety of systems. Further, we have not yet exploited the hierarchical and inter-agent capabilities of AutoTune, and much work remains in designing and automating effective control mechanisms for enterprise-scale distributed systems.

**Trademark or registered trademark of Apache Digital Corporation, The Open Group, Sun Microsystems, Inc., or Linus Torvalds.

Cited references

1. P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology*, IBM Corporation (October 15, 2001); available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
2. D.-M. Chiu and R. Jain, "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks," *Computer Networks and ISDN Systems* **17** (1989).
3. S. Keshav, "A Control-Theoretic Approach to Flow Control," *Proceedings of the Conference on Communications Architecture and Protocols* (ACM SIGCOMM '91), ACM, New York (1991).
4. S. Mascolo, "Classical Control Theory for Congestion Avoidance in High-speed Internet," *Proceedings of the 38th Conference on Decision and Control*, IEEE (1999).
5. C. V. Hollot, V. Misra, D. Towsley, and W. B. Gong, "A Control-Theoretic Analysis of RED," Infocom 2001, IEEE, Anchorage, AK (2001), pp. 1510–1519.
6. Y. Lu, A. Saxena, and T. F. Abdelzaher, "Differentiated Caching Services: A Control-Theoretic Approach," *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, AZ (2001).
7. C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, *A Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers*, Technical Report CS-2001-06, University of Virginia, Department of Computer Science (2001).
8. B. Li and K. Nahrstedt, "Control-based Middleware Framework for Quality-of-Service Applications," *IEEE Journal on Selected Areas in Communications* **17**, No. 9, 1632–1650 (1999).
9. Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server," *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium*, Florence, Italy (2002).
10. S. Parekh, N. Gandhi, J. L. Hellerstein, D. M. Tilbury, and J. P. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *Proceedings of the 7th IEEE/IFIP Symposium on Integrated Network Management*, Seattle, WA (2001), pp. 841–854.
11. Y. Diao, J. L. Hellerstein, and S. Parekh, "A Business-Oriented Approach to the Design of Feedback Loops for Performance Management," 12th International Workshop on Distributed Systems: Operations and Management (2001).
12. R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed, "Autopilot: Adaptive Control of Distributed Applications," *Pro-*

ceedings of the 7th IEEE Symposium on High-Performance Distributed Computing (1998).
13. J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A Toolkit for Building Multiagent Autonomic Systems," *IBM Systems Journal* **41**, No. 3, 350–371 (2002).
14. J. P. Bigus, J. L. Hellerstein, and M. S. Squillante, "Auto Tune: A Generic Agent for Automated Performance Tuning," *Proceedings of the International Conference on Practical Application of Intelligent Agents and Multi-Agents* (PAAM) (2000).
15. Apache Software Foundation. See http://www.apache.org.
16. Netcraft, "Netcraft Web Server Survey," http://www.netcraft.com/survey.
17. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol—HTTP/1.1*, RFC 2616, Internet Engineering Task Force (IETF), June 1999; see http://www.ietf.org/rfc/rfc2616.txt.
18. Y. Diao, J. L. Hellerstein, and S. Parekh, "Optimizing Quality of Service Using Fuzzy Control," *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems Operations and Management* (2002).
19. J. P. Bigus, "The Agent Building and Learning Environment," *Proceedings of the Fourth International Conference on Autonomous Agents* (2000), pp. 108–109.
20. L. Ljung, *System Identification: Theory for the User*, Prentice Hall, Upper Saddle River, NJ (1999).
21. G. F. Franklin, J. D. Powell, and M. L. Workman, *Digital Control of Dynamic Systems*, Addison-Wesley Publishing Co., Reading, MA (1998).
22. W. H. Fleming and R. W. Rishel, *Deterministic and Stochastic Optimal Control*, Springer Verlag (1996).
23. Z. Liu, N. Niclausse, C. Jalpa-Villanueva, and S. Barbier, *Traffic Model and Performance Evaluation of Web Servers*, Technical Report 3840, Institute National de Recherche en Informatique et en Automatique (INRIA) (1999).
24. D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Proceedings of the First Workshop on Internet Server Performance* (WISP 98), ACM (1998), pp. 59–67.
25. I. Mindcraft, "Webstone 2.5 Web Server Benchmark" (1998). See http://www.mindcraft.com/webstone/.

Accepted for publication October 16, 2002.

Yixin Diao *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: diao@us.ibm.com).* Dr. Diao is a research staff member at the IBM T. J. Watson Research Center. He received his Ph.D. degree in electrical engineering from Ohio State University in 2000. He has authored more than 20 papers, and his research interests include computer performance management, intelligent systems and control, adaptive systems, and stability analysis.

Joseph L. Hellerstein *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: hellers@us.ibm.com).* Dr. Hellerstein is a research staff member at the IBM T. J. Watson Research Center, where he manages the adaptive systems department. He received his Ph.D. degree from the University of California in Los Angeles. Since then, his research has addressed various aspects of managing service levels, including: predictive detection, automated diagnosis, expert systems, and the application of control theory to resource management. Dr. Hellerstein has published approximately 60 papers and an Addison-Wesley book on expert systems.

Sujay Parekh *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: sujay@us.ibm.com).* Mr. Parekh is a research associate at the IBM T. J. Watson Research Center. He received his B.S. degree in computer science and B.A. in mathematics at Cornell University in 1993, and his M.S. degree in computer science at the University of Washington in 1996. His research interests center around automating both simple and complex computing systems, and have included work in artificial intelligence planning, machine learning, computer architecture, scheduling algorithms, and feedback-driven control.

Joseph P. Bigus *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: bigus@us.ibm.com).* Dr. Bigus is a Senior Technical Staff Member at the IBM T. J. Watson Research Center, where he is the project leader on the ABLE research project. He is a member of the IBM Academy of Technology and an IBM Master Inventor, with over 20 U.S. patents. Dr. Bigus was an architect of the IBM Neural Network Utility and Intelligent Miner for Data products. He received his M.S. and Ph.D. degrees in computer science from Lehigh University and a B.S. degree in computer science from Villanova University. Dr. Bigus's current research interests include learning algorithms and intelligent agents, as well as multiagent teams and their applications to adaptive system modeling and control, data mining, and decision support.