Profit-driven Service Differentiation in Transient Environments*

Qi Zhang Evgenia Smirni Gianfranco Ciardo
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
{qizhang,esmirni,ciardo}@cs.wm.edu

Abstract

We focus on service differentiation policies for Web servers where clients have different QoS requirements and the environment has large fluctuations in the client arrival and service patterns and in the QoS class mix. We propose an adaptive admission control mechanism that offers service differentiation among client classes and maximizes the effectiveness of the server's operation. Using actual traces from the 1998 World Cup web site, we conduct a detailed analysis of the proposed policy and show that it meets the performance challenges of a robust QoS policy.
Keywords: *service differentiation, performance guarantees, transient workload*

1 Introduction

As Web technologies become the standard interface for most IT applications, there is an increasing need for service differentiation among different user classes. This has prompted a significant body of research that focuses on providing differentiated services in Web servers: [2] meets pre-specified class performance targets through an admission control mechanism; [4, 5] use queuing models analysis to formulate an optimization problem aiming at providing Quality-of-Service (QoS) guarantees based on measures of interest; [3, 6] dynamically partition critical resources while maintaining consistent performance spacings among different classes. Here, instead of optimizing classic performance measures such as throughput or response time, we focus on a cost model based on the per-class achieved *slowdown* [6].

We consider a scenario where a Web server admits *multiple* classes of customers. Classes are characterized by a QoS soft performance guarantee expressed by a service level agreement (SLA). We develop a Profit Aware QoS policy, PAQoS, aimed at maximizing the Web site's *profit* under SLA constraints. PAQoS uses a generalized processor sharing (GPS) discipline and continuously *adapts* to the workload by changing its sharing parameters as a function of the achieved profit and transient overload conditions. Using simulation, we perform a detailed sensitivity analysis that demonstrates the robustness of our policy in a changing environment. Comparisons with Neptune's QoS policy [6], which is designed to meet similar goals, indicate that the proposed mechanism is robust and effective, especially in highly variable environments.

2 Environment

We assume a commercial Web site, possibly organized as a cluster, providing static or lightly dynamic Web pages created on demand. We do not consider the problem of distributing the incoming requests across the servers in the cluster. We assume that the site employs a load balancing policy which has been shown to provide good performance in such setting and we concentrate instead on the problem of scheduling requests *within* each server.

Past studies point at conflicting bottlenecks in Web request processing, but we consider workloads with a single bottleneck, e.g., the CPU. We therefore assume systems that serve mainly static Web requests possibly using SSL encryption, which, due to the associated computational overhead in encrypting data, make the workload CPU-intensive but also provide the system with *a priori* knowledge of CPU demands. We also consider lightly dynamic workloads where a fraction of the requests results in database queries, which we assume do not require intensive use of server resources. Naturally, there is no *a priori* knowledge of CPU demands for these dynamic requests.

We consider a system that serves multiple classes of customers, such as an e-commerce site, and charges, or profits from, each customer *request*, according to a predefined SLA. Therefore, the purpose of the system is to satisfy as many requests as possible so as to maximize the site's rev-

*This work was partially supported by the National Science Foundation under grants EIA-9974992 and CCR-0098278.

enue, while at the same time provide services to the various customer classes within predefined SLAs. More specifically, performance is measured in terms of the *slowdown* for each request, defined as the ratio of the response time for that request over the minimum service time possible for that request. Given the vast variance in the size of the requests typically observed by Web servers, slowdown is a more meaningful measure of performance than the *response time*, which is instead appropriate when requests are comparable in size. This implies that a user requesting a large file is willing to tolerate a proportionally longer latency [1].

The amount charged by the Web server is determined by both the guaranteed and the actual performance, according to the following rule: "For each request of class i, the Web server charges an amount pay_i if the slowdown experienced by that request is no more than $agree_i$. If instead the requests experiences a slowdown above $agree_i$ or if it is dropped (i.e., not serviced at all), the web server does not charge anything for it." We sort service classes in decreasing order of performance (and priority), thus $agree_i < agree_{i+1}$ and, accordingly, $pay_i > pay_{i+1}$, for $1 \leq i < n$. Figure 1(a) illustrates our charge scheme, which follows this rule, and a closely related one where the payment drops off linearly, instead of abruptly (b). Any admission control policy aimed at maximizing the site's profit should selectively admit those requests that have the potential of yielding the highest profit, without starving the class of requests that results in lower profit. We return to this issue in Section 3, when discussing our differentiation policy.

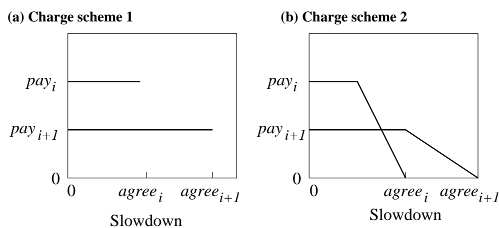

Figure 1. Slowdown-based request pricing.

3 Profit Aware QoS (PAQoS) policy

We assume a generalized processor sharing (GPS) discipline at each server, where each request is allocated a certain *fraction* of the resource. Other QoS policies based on processor sharing employ a pool of *threads* to provide differentiated services; a policy defines how a request is assigned to a specific thread but, once this is decided, each thread shares the resource with other threads in a round-robin fashion.

We adopt profit as a function of the slowdown as the measure of performance and do not use the thread pool approach for several reasons. First, it is difficult to decide the size of the thread pool: if it is too large, the average slowdown of the admitted requests might be excessive; if it is too small, requests wait too long before admission. Second, once a large request is admitted, it may negatively affect the performance of shorter requests admitted later, since the time a thread is busy with a request depends on the request size. Finally, admitting low-priority requests may violate the performance requirements of high-priority requests.

3.1 Policy Overview

The n different service classes of requests are organized in a multiple-queue structure used at each server node by our PAQoS policy. When a new request reaches the server, it joins the queue corresponding to its service class.

The server time slice allocation policy is organized into *rounds*. Within a round, each class can receive service. Let A_i be the the number of slices *allocated* to class i, that is, the maximum number of slices that can be assigned to serve requests of class i in the current round. Let C_i be the the number of slices that have been already *consumed* by requests of class i in the current round, and Q_i be the length of the *queue* for class i, that is, the number of requests of class i currently ready to be served. In PAQoS, a request from class i is scheduled only if $C_j = A_j$ or $Q_j = 0$ for all classes $j < i$. At the completion of one slice for class i, the system *does not* simply schedule another request in class i or in a low-priority queue. Instead it re-starts from class 1, searching for a nonempty queue ($Q_j > 0$) whose allocated slices have not yet been exhausted in this round ($C_j < A_j$). Thus, if a new higher-priority request for class $j < i$ arrived, it is served in this round, if $C_j < A_j$. A new round begins when $C_i = A_i$ for all non-empty queues i.

The value of A_i is therefore fundamental, as it enforces priorities and provides differentiated levels of service, but so is the scheduling of requests *within* a class. Since we seek to maximize profit according to a slowdown-threshold agreement, our policy considers the request size and its deadline as defined by the SLA in each scheduling decision. A longer request with an earlier deadline should be served before a shorter request with a later deadline, if by doing so both requests can still meet their respective deadlines. On the other hand, if attempting to let the longer request meet its deadline would force too many other requests to risk missing their deadline, the longer request should be dropped for the sake of maximizing the overall profit, especially under heavy load.

We define two variables associated to each request r in the queue for class i: T_r denotes the time of the *deadline* for request r; R_r denotes the *remaining* number of time slices required to serve request r in isolation. A new request k enters its queue in a position such that, for any request l ahead of it, $R_l < R_k$, or $R_l = R_k$ and $T_k \leq R_k$. Any request r that generates zero profit is dropped.

1. $sumPrevIncrs \leftarrow 0$;
2. for $i = (n-1)$ downto 1 do
3. if $drop_i/drop_{i+1} > d_i(1+\delta)$ then
4. $A_i \leftarrow A_i + incr + sumPrevIncrs$;
5. $sumPrevIncrs \leftarrow sumPrevIncrs + incr$;
6. else if $drop_{i+1}/drop_i > (1/d_i)(1+\delta)$ then
7. $A_i \leftarrow A_i - incr + sumPrevIncrs$;
8. $sumPrevIncrs \leftarrow sumPrevIncrs - incr$;
9. $norm \leftarrow \min_{1 \le i \le n}\{A_i\} - 1$;
10. for $i = 1$ to n do
11. $A_i \leftarrow A_i - norm$;

Figure 2. Adjusting time slice assignments.

3.2 Slice Allocation

Key to the effectiveness of providing differentiated services is the slice allocation scheme used by PAQoS, which is in turn guided by the percentage of dropped requests within each class. Let $drop_i$ be the instantaneous percentage of dropped requests for class i and define $p_i = pay_i/pay_{i+1}$ and $a_i = agree_i/agree_{i+1}$. Considering a simple system where the resource is equally distributed among all the classes, the drop ratio should be inversely proportional to its agreed slowdown (i.e., $drop_i/drop_{i+1} \sim 1/a_i$). On the other hand, if all classes have the same agreed slowdown and we aim at providing a quality of service proportional to the price paid, the drop ratio of each class should be inversely related to its price (i.e., $drop_i/drop_{i+1} \sim 1/p_i$). Therefore, in a general system with various prices and agreed slowdowns, the drop ratio should depend on both a_i and p_i:

$$drop_i/drop_{i+1} \sim 1/(p_i a_i) \stackrel{\text{def}}{=} d_i, \quad 1 \le i \le n-1. \quad (1)$$

To promptly adapt to sudden load changes, the system maintains a monitoring window during which it records drop statistics. The number of slices allocated to each class i, A_i, is then adaptively updated to satisfy Eq. (1), as showed in Figure 2, where δ is a *tolerance* parameter and $incr$ is a constant.

4 Performance analysis

To evaluate the proposed PAQoS policy we use the workload traces of the 1998 World Soccer Cup Web site, Available on the Internet Traffic Archive at `http://ita.ee.lbl.gov/`. Although this is not an e-commerce workload, it is characterized by high variability in its arrival pattern (i.e., arrival intensities vary significantly over time) and service demands. We use day June 27 of this trace, which we selected as representative.

We compare PAQoS with our implementation of Neptune's resource management QoS framework [6], which also focuses on maximizing the site's *profit* according to the charge scheme illustrated in Figure 1(a) (we also conducted experiments using the charge scheme of Figure 1(b), which resulted in qualitatively similar results). As Neptune's performance is sensitive to the size of the thread pool, we use the size that experimentally yields the highest profit.

Experiment 1: Exponential Arrival Rate
The first experiment concentrates on PAQoS's ability to adapt to highly variable request sizes by enforcing a stationary arrival process with exponentially distributed request interarrival time, while the service times are given by the trace data. We assume a fixed workload mix consisting of 30% high-priority requests (class 1) and 70% low-priority requests (class 2). To study the policy under high load conditions, we choose an arrival rate resuting in nearly 80% utilization. We also experimented with lower levels of utilization; the results are not reported here for lack of space, but they are consistent across different load conditions.

PAQoS provides better differentiation for the QoS levels. Its average slowdown is about half that of Neptune, thus PAQoS achieves significantly higher profit, see Figure 3. The drop ratios for PAQoS are also significantly less than Neptune's. This can be attributed to non-preemptiveness in Neptune's policy: admitting long low-priority requests may hurt the performance of short high-priority requests.

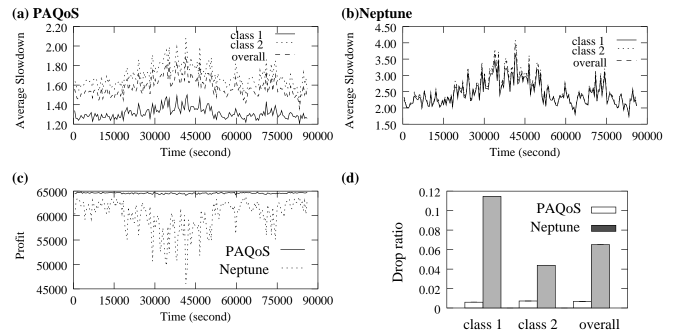

Figure 3. Performance under heavy load (exponential arrivals).

Experiment 2: Variability in Arrival Rate and Workload Mix
Having established the effectiveness of PAQoS for workloads that are highly variable with respect to their service process, we now present the experiments using the arrival time stamps of the actual trace. This forces the system to operate at high load during the late evening hours but at much lower load during the morning hours. Furthermore, we assume that the class mix is not persistent during the experiment, but controlled by a Markov modulated Poisson process (MMPP) where (exponentially distributed) periods

with 20% high-priority requests alternate with periods with 50% high-priority requests, see Figure 4(a).

Figures 4(c) and 4(d) show the average slowdowns of the two policies as a function of the time of the day. Figure 4(b) illustrates that the profit of the two policies are nearly identical at low loads, while PAQoS outperforms Neptune at the more challenging high loads.

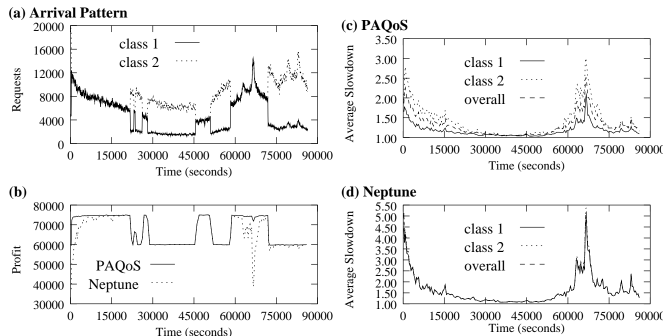

Figure 4. Performance (actual arrival times).

Experiment 3: Dynamic workload
Since PAQoS is based on *a priori* knowledge of service times (which, in turn, define the deadlines), it cannot be used without modification for dynamic workloads. In this experiment, we still assume that the workload mix is controlled by the MMPP as in *Experiment 2* but now the workload contains a portion of requests for dynamically generated pages. We further assume that dynamic requests cause no contention at the application server and that the time to generate and process dynamic content is exponentially distributed with rate μ. For each dynamic request we "guess" its duration to be the mean $1/\mu$ of the exponential distribution. If the guessed value is smaller than the actual duration of the request, then we augment the anticipated request duration by one standard deviation (which, for the exponential distribution, also equals $1/\mu$). We repeat this process at most twice and, if the request is still not completed, we drop it (this means that it required more than $3/\mu$ service time; given our exponential distribution assumption, this happens only for approximately 5% of the dynamic requests).

Figure 5 shows the average profit loss ratios (defined as the ratio of PAQoS's profit vs. the profit achieved in the ideal case where the duration of *all* requests is known *a priori*) as a function time for workloads with different percentages of dynamic requests. The worst case occurs during peak load, but performance is still nevertheless good even when a substantial portion of the workload is dynamic. We conclude that PAQoS is well suited for mixed workloads.

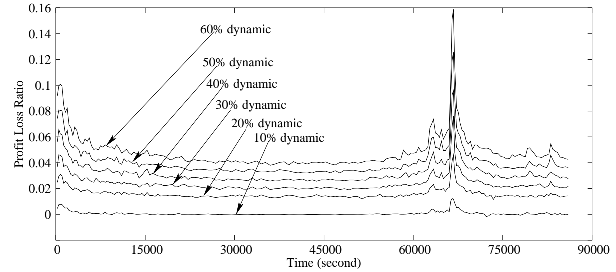

Figure 5. Profit loss ratios with workloads containing different percentages of dynamic requests in a variable environment.

5 Conclusions

We introduced PAQoS, a new resource allocation policy that enforces quality of service differentiation among classes of customers having different service level agreements, with the goal of maximizing profit for the service center operator. Our policy uses a dynamic choice for the allocation of time slices to classes of service, to balance the request drop ratio with their profit potential and performance guarantees, even when the arrival and service processes are highly variable. Experimentally, we show that PAQoS achieves consistently higher profit than Neptune, a previously proposed policy, especially under the severe transient loads common in e-commerce Web servers.

References

[1] A. Bouch, A.Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *Proceedings of ACM SIG-CHI*, pages 297–304, The Hague, Netherlands, Apr. 2000.

[2] W. Chen and P. Mohapatra. Session-based overload control in QoS-aware Web servers. In *Proceedings of INFOCOMM 2002 Conference*, New York, NY, June 2002.

[3] S. C. M. Lee, J. C. S. Lui, and D. K. Y. Yau. Admission control and dynamic adaptation for a proportional-delay DiffServ-enabled Web server. In *Proceedings of ACM SIGMETRICS Conference*, pages 172–181, Marina Del Ray, CA, June 2002.

[4] Z. Liu, M. S. Squillante, and J. L. Wolf. On maximizing server-level-agreement profi ts. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 213–223, Tampa, FL, Oct. 2001.

[5] D. A. Menasce, D. Barbara, and R. Dodge. Preserving QoS of E-commerce sites through self-tuning: a performance model approach. In *Proceedings of the ACM Conference on Electronic Commerce*, pages 224–234, Tampa, FL, Oct. 2001.

[6] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the Fifth Symposium on Operating Systems Designand Implementation*, Boston, MA, Dec. 2002.