A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites

Sameh Elnikety
School of Computer and Communication Sciences
EPFL
CH 1015 Lausanne, Switzerland

Erich Nahum
Networked Systems Department
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598, USA

John Tracey
Networked Systems Department
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598, USA

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL
CH 1015 Lausanne, Switzerland

ABSTRACT

This paper presents a method for admission control and request scheduling for multiply-tiered e-commerce Web sites, achieving both stable behavior during overload and improved response times. Our method externally observes execution costs of requests online, distinguishing different request types, and performs overload protection and preferential scheduling using relatively simple measurements and a straightforward control mechanism. Unlike previous proposals, which require extensive changes to the server or operating system, our method requires no modifications to the host O.S., Web server, application server or database. Since our method is external, it can be implemented in a proxy. We present such an implementation, called Gatekeeper, using it with standard software components on the Linux operating system. We evaluate the proxy using the industry standard TPC-W workload generator in a typical three-tiered e-commerce environment. We show consistent performance during overload and throughput increases of up to 10 percent. Response time improves by up to a factor of 14, with only a 15 percent penalty to large jobs.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-communication networks—*Distributed systems*; C.4 [**Computer Systems Organization**]: Performance of systems; D.4.8 [**Operating Systems**]: Performance

General Terms

Measurement, performance

Keywords

Dynamic Web content, Web servers, admission control, load control, request scheduling

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

1. INTRODUCTION

E-Commerce is a growing phenomenon as consumers gain experience and comfort with shopping on the Internet [33]. E-Commerce Web sites are typically composed of a three-tiered architecture consisting of a front-end Web server, an application server and a back-end database. Online merchants desire to maintain a continuous, consistent presence on the Web in order to keep customers satisfied and maximize both revenues and returns on their infrastructure.

Two problems are typically encountered with deploying e-commerce Web sites. First is *overload*, where the volume of requests for content at a site temporarily exceeds the capacity for serving them and renders the site unusable. Second is *responsiveness*, where the lack of adequate response time leads to lowered usage of a site, and subsequently, reduced revenues. Both issues are instances of a larger problem: given the unpredictability of Web accesses, how can an e-commerce site provide responsive service to clients, even when user demand outstrips the capacity of the site?

This paper presents a method for providing admission control and request scheduling for multiply-tiered e-commerce Web sites. Our approach externally measures execution costs online, differentiating between different types of requests, enabling overload protection and dramatic improvements in response time. Our admission control scheme *accounts for variations in service costs*. By measuring service times online, our system is more robust to overload than approaches which assume that measurements taken under light load are applicable to heavy load situations. As will be seen, service costs can change as a function of load in the system. Our approach also accounts for variation in the execution times of different types of requests; other approaches only account for a single metric such as overall response time or queue length, or assume a simple linear model of service costs. In contrast, we track the amount of work generated by each request directly.

Other proposals require extensive modifications to the operating system or a complete re-write of the server. Our implementation requires no changes to the source code, server software, application programs, or to the database. The benefits of such an approach are clear: the use of unmodified commodity software components reduces development effort tremendously. As a result, we are able to demonstrate our approach using *standard software components*

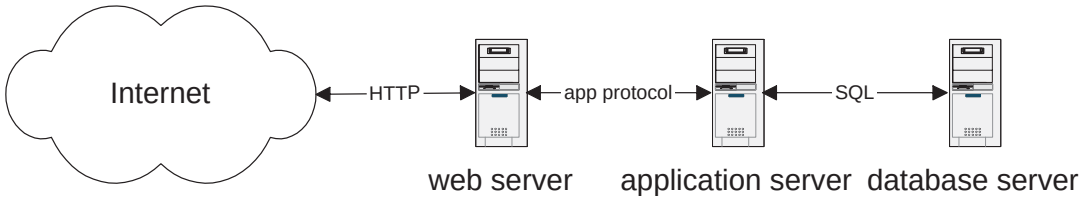

Figure 1: Architecture of an e-commerce Web site.

and workload generators.

Our method is embodied in a proxy, called *Gatekeeper*. A key feature is that it is *transparent to the database and application server*. For our evaluation, we use a standard test-bed environment of a multiple-tiered e-commerce Web site, with Linux, Apache, Tomcat, MySQL and DB2. In this environment, the database is the bottleneck, and thus Gatekeeper is placed so as to transparently intercept requests from the application server to the database. Driving the system with the industry-standard TPC-W benchmark, Gatekeeper achieves both stable behavior during overload and dramatically improved response times.

Even though our approach is external and treats the system as a black box, we gain many of the benefits of admission control and request scheduling. Examples include:

- We show *consistent performance during overload*, across different database implementations and locking approaches.
- We demonstrate how admission control can *improve peak throughput* by up to 10 percent, by preventing thrashing and improving memory reference locality.
- We show how preferential scheduling in the form of shortest job first (SJF) can make *dramatic improvements* to response time for dynamic Web requests, while penalizing large jobs only slightly. Average response time improves up to 14-fold, while penalizing large jobs by only 15 percent.
- We present and evaluate an aging mechanism that *prevents starvation* of large jobs when preferential scheduling is used. Our results show a continuum of behavior between FIFO and SJF request scheduling; Web site operators can use the mechanism to implement a policy enforcing a particular behavior.

These results show that relatively invasive techniques that require extensive modifications to systems are not always necessary, particularly in the context of multiply-tiered e-commerce Web sites.

The remainder of this paper is organized as follows: Section 2 overviews the relevant background for our work. Section 3 presents our implementation of admission control and request scheduling in the Gatekeeper proxy. Section 4 describes our experimental environment, and Section 5 shows our results in detail. Section 6 discusses related work. Finally, section 7 summarizes our conclusions and offers possible directions for future work.

2. BACKGROUND

In this section, we provide a brief overview of dynamic content generation, admission control, overload control, and request scheduling, in the context of Web servers.

An e-commerce Web site is typically comprised of three components as depicted in Figure 1: a front-end Web server, application server and back-end database. The front-end Web server usually handles the static component of the workload, such as images and infrequently-changing HTML pages. The application server provides an environment to invoke methods that implement the business and presentation logic of the application. Examples of these methods include PHP scripts, Active Server Pages (ASPs) and Java Servlets. The application logic issues a number of queries to the database, which stores the true dynamic state of the Web site (for example, the number of copies of a book in stock). The application server formats the returned database query results as an HTML page, which is passed back to the front-end Web server. Finally, the Web server returns the aggregated content as an HTTP response to the client.

Admission control and coping with overload is used to prevent systems from being overwhelmed in the presence of persistent or transient overload. Research in this area can be roughly categorized under two broad approaches: reducing the amount of work required when faced with overload, and differentiating classes of customers so that response times of preferred clients do not suffer in the presence of overload.

Multiple proposals have been offered for using QoS techniques on Web servers [3, 11, 28, 44]. These have tended to include some form of classification in the form of client IP address, IP subnet, URL or cookie to identify requests as belonging to a particular differentiated level of service. To support these classes, these approaches also include admission control and request scheduling, in order to provide a particular server throughput, network bandwidth, or client response time. Some have advocated using observation-based measurements for providing QoS guarantees [35]; others have proposed a control theoretic approach [2].

Our work complements and extends the above results by applying admission control for Web sites with dynamic content, rather than simply static content. We use admission control to prevent overload and maintain peak aggregate throughput, rather than for guaranteeing response times for differentiated classes of service. We believe extending our system to guarantee response times should be straightforward.

In the last few years, a great deal of interest has arisen in applying a *scheduling policy* in the context of Web servers providing static content. Cherkasova [16] proposed using shortest job first scheduling for static content Web sites. By using the URL in an HTTP request to identify a file, the cost of servicing that file (that is, the "job size") could be well approximated by the size of the file. Cherkasova evaluated her proposal via trace-driven simulation. Crovella et al. [19] independently proposed a similar approach, scheduling outbound responses according to an "shortest connection first" policy similar to shortest remaining processing time first (SRPT). They demonstrated experimental results showing that response time could be improved by up to a factor of 5. Schroeder and Harchol-Balter [37] demonstrate an additional benefit from performing SRPT scheduling for static content Web requests. Under a wide breadth of networking and server conditions,

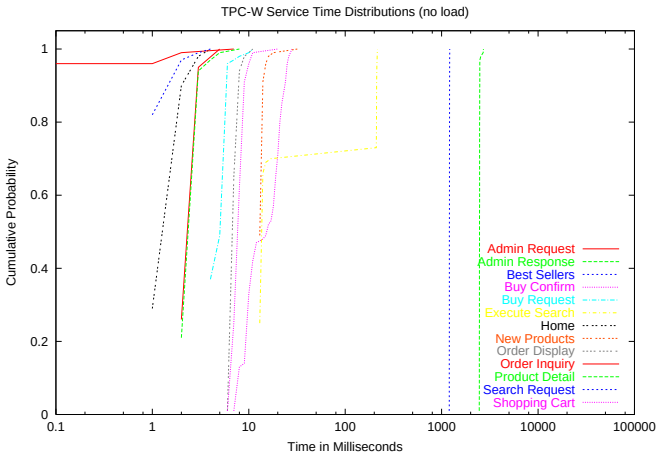

Figure 2: Execution times of TPC-W servlets (no load).

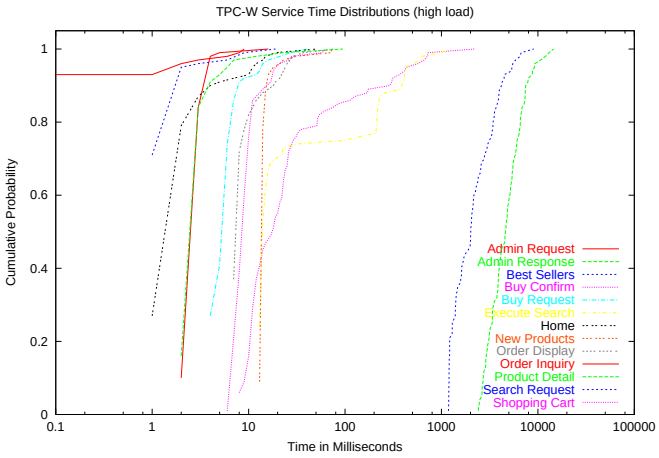

Figure 3: Execution times of TPC-W servlets (peak load).

they show that SRPT scheduling can be used to mitigate the response time effects of transient overload conditions.

Our work advocates using preferential scheduling for dynamic content Web sites in a transparent fashion, and evaluates its benefits. We also address the starvation question by using an aging mechanism to prevent starvation.

3. THE GATEKEEPER PROXY

In this Section we describe the design of our method and its implementation in the *Gatekeeper* proxy; which transparently intercepts requests for dynamic Web Content. In this work, we use Java servlets for application functionality, due to their frequent use in commercial dynamic content Web sites and their competitive performance [13].

3.1 Admission Control

Admission control generally requires two components: knowing the *load* that a particular job will generate on a system, and knowing the *capacity* of that system. By keeping the maximum amount of load just below the system capacity, overload is prevented and peak throughput is achieved. In a dynamic content generation system, we satisfy the above requirements in the following manner.

For the first requirement, Gatekeeper identifies different request types (i.e., servlets) and maintains online estimates of their expected service times, based on measurements of recent executions. The measurement interval is taken from when the servlet issues the first query until the response from the last query is received. The execution times are "learned" in an online fashion through a simple moving average, and used as the estimated *load* that each request imposes on the system. Given that any dynamic Web site has a finite number of interactions, it is simple to maintain per-servlet estimates. For example, the TPC-W workload has only 14 interactions, each of which is embodied by a single servlet.

While different servlets issue different sets of queries, any one servlet will typically issue the same set of database queries, albeit with different parameters. Our experience is that the load generated by a particular request is generally consistent; namely, that service times depend primarily on *which* servlet is being executed rather than on the *parameters* to that servlet. This is illustrated in Figures 2 and 3, which show the service time distributions of the 13 TPC-W interactions that communicate with the database. Figure 2 shows the times when measured in isolation (i.e., under no load) and Figure 3 shows the times when measured at peak load. The X-axis in Figures 2 and 3 is in log scale. Also, we show the averages of the same data in Table 1.

Two significant observations can be made from these Figures. First, variation *across* servlets is much greater than variation *within* servlets. Service times between servlets vary significantly as they range from less than 1 ms to over 5 sec, a range covering four orders of magnitude. Variation within the servlet is much less, as shown by the relatively vertical straight lines in the distributions. While in some cases variation within the script can be as much as a factor of 10, it is still orders of magnitude smaller than the variation across scripts. Second, service times under heavy load are very different from those measured in isolation; most are much higher than in the no-load case. This shows the importance of measuring costs online rather than using offline measurements.

Per-servlet estimates converge relatively quickly; in our experiments, 11 out of the 13 servlets converge to within 90 percent of their steady state values in under 50 seconds. We then define the *load* produced by the servlet to be the average execution time (in milliseconds) that the servlet requires.

The second requirement is to determine the capacity of a system, which is a single numeric value. For our current environment, we determine the capacity of the system offline rather than online. Our approach is to use the method of incremental steps [24]. Briefly, this method assumes that the functional relationship of the load and resulting performance is monotonically increasing up to a peak and then decreasing; in other words, it assumes the throughput function is relatively concave, with the existence of a local maximum that is also the global maximum. We define *capacity* as the maximum load level that produces the highest throughput. Capacity is essentially a "pool" of execution units that are allocated to requests. To determine a particular system's capacity, we run experiments using several relatively large candidate values (e.g., 20,000 or 40,000 milliseconds) for the capacity. We then perform a binary search process of changing the candidate numbers and measuring the resulting performance, seeking the proper candidate value that provides the maximum throughput. When the binary search converges, we use the resulting value as our estimate of capacity.

Because we consider only the functional relationship between the load and resulting performance, our technique is oblivious to the bottleneck resource (which could be a physical resource such as CPU cycles or a logical resource such as database locks). The capacity of the system should be recomputed if there is a change

Servlet Name	Admin Req	Admin Resp	Best Seller	Buy Conf	Buy Req	Exec Search	Home	New Prod	Order Disp	Order Inq.	Prod Detail	Search Req	Shop Cart
No Load	2.84	2476.75	1205.01	15.71	5.26	73.27	1.83	13.82	7.43	0.75	0.91	0.22	0.34
High Load	3.19	5072.42	2441.86	92.80	6.66	108.73	3.22	15.66	10.48	0.05	0.30	0.65	0.92
Frequency	0.1	0.09	5	1.2	2.6	17	16	5	0.66	0.75	17	20	11.6

Table 1: Servlet average execution costs (ms) and servlet execution frequencies (percentage).

in the system, for example hardware upgrade, or a change in the workload. Techniques with more elaborate adaptive algorithms for estimating the online capacity of a running system could also be used, and presumably control theoretic approaches as well.

With these two estimates, admission control is straightforward. In the implementation, the Gatekeeper proxy maintains a running estimate of system load, initially set to zero. When a servlet requests a database connection, the work estimate for that servlet is examined. If admitting the servlet will not exceed the capacity, then the servlet is allowed to proceed, and the running load value is incremented accordingly. Otherwise, the servlet is deferred to execute later, and is placed in the *admission queue* in FIFO order. As jobs finish, Gatekeeper decrements the running load value appropriately, and if jobs are waiting in the queue, they are admitted in turn as long as they do not overload the system. Gatekeeper does not shed load by dropping requests; in persistent overload conditions, the request waiting time will eventually become infinite and pending requests will time out indicating the need to upgrade the system. Other admission control policies, such as dropping certain requests, could easily be provided depending on the requirements of the Web site operator.

In summary, we use measurement-based admission control. We maintain online estimates of request service times based on recent request executions. The estimated request service time indicates the load that a request imposes on the system. We estimate the current load on the system as the sum of the estimated service times of all executing requests. The system capacity is measured offline and it limits the number of requests admitted to the system; excess requests are queued.

3.2 Request Scheduling

The Gatekeeper proxy uses request scheduling to reduce the average response time of dynamic Web site interactions, in the form of a shortest-job first (SJF) policy. As requests arrive, they are placed in the admission queue sorted based on their expected processing times. When requests finish, new requests are admitted from the front of the queue until the capacity threshold is reached. After jobs are admitted, they run to completion, i.e., they are not pre-empted.

Scheduling cannot improve response times if the workload is completely homogeneous. For example, if each request required the same service time, changing the order of execution would be pointless. However, large variability has been found in many Internet workloads (e.g., [18]). In TPC-W , for example, execution times vary by as much as five orders of magnitude, as was seen in Figures 2 and 3. Most service times are small, less than 54 milliseconds. However, two interactions, the “Best Seller” and “Administrative Control” servlets, have service times of over one and two seconds, respectively, in the no-load case. Their cost is high because they issue complex database queries. Luckily, the expensive queries tend to be infrequent. Table 1 shows the frequency distribution for the thirteen interactions that invoke the database using the shopping mix of TPC-W. The fourteenth TPC-W interaction requires no database access and it appears 3% of the time.

A frequent concern with preferential scheduling is that long jobs may starve in the presence of a large number of small requests. To prevent starvation, we implemented an aging mechanism, which is similar to Alpha scheduling [16]. The aging mechanism enforces an upper bound on the amount of time a request is delayed in the queue. That upper bound is defined as a multiple of the expected service time for the request. The upper bound is configurable. For example, a Web site operator can choose a policy such that a request will not wait more than twice its expected service time. Shorter requests can be promoted towards the head of the queue only if this promotion does not cause any pending request to be delayed more than its bound. Aging only affects request scheduling, it does not affect the average system throughput, as will be seen in Section 5.

4. EXPERIMENTAL ENVIRONMENT

In this section, we describe our environment for doing our experimental evaluation, including the hardware and software used, metrics, and experimental methodology.

4.1 Hardware and Software

Our testbed consists of a client PC, two server PCs, and a 100 Mbps Ethernet switch. Each PC has a 1.33 GHz AMD Athlon, 768 MB RAM, and a 60 GB 5400 RPM disk. All machines have a 100 Base-T Ethernet interface connected point-to-point full duplex with the switch. One server machine runs the Web server and application server software, while the other contains the database. The client machine drives the system with a workload generator, described in more detail in section 4.4 below. All machines run Red Hat Linux with the Linux kernel 2.4.18. We use Apache version 1.3.27 for the front-end Web server, and Jakarta Tomcat version 3.2.4 as the application server. For the relational database, we use two versions: MySQL version 3.23.53-max for most experiments and DB2 for Linux version 7.2. for one experiment. To measure the load on each machine, we used the sysstat utility [40] that collects CPU, memory, network and disk usage from the Linux kernel every second. The AMD Athlon processor has four performance monitoring counters [4] that count processor events (e.g., level 1 data cache misses, level 1 and level 2 data TLB misses). We also use Rabbit [25], a library and kernel module, to access these performance monitoring counters.

Since we are evaluating Gatekeeper in the context of an e-commerce Web site, we place the proxy between the application server and the database, as shown in Figure 4. To access a database, servlets produce SQL queries using the JDBC API. In turn, the JDBC driver invokes the vendor-specific connector code to interact with the database server. Gatekeeper thus intercepts requests to communicate with the database through the JDBC API.

4.2 Locking Options

In e-commerce Web sites, locking can be performed in either of two places: on the database itself, or in the servlet that invokes the particular transaction. The advantage of locking in the application server is that performance is greatly improved, as will be seen in Section 5.1. The disadvantage is that all access to the database

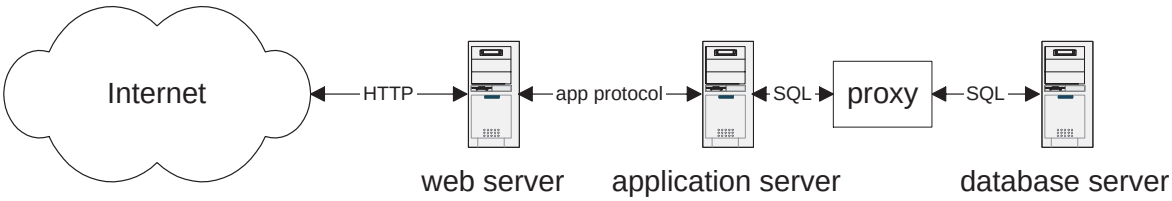

Figure 4: Placement of the Gatekeeper proxy within an e-commerce Web site.

must be directed through the application server, which might not be feasible. Since the decision of where locking is done will depend on many factors, such as the load on the site, architecture of the system, and perhaps even the inventory management technique (e.g., just-in-time, inexact inventory), we evaluate both scenarios. When locking is done in the database, we use strict two phase locking. For each servlet that accesses the database, we insert database read and write lock operations that obtain all necessary locks for the queries in that servlet. Locks are held until the end of the execution of the servlet. When locking is done in the servlet, no explicit SQL locking operations are added; the Java synchronization mechanisms are used instead to serialize conflicting requests to the database.

4.3 The TPC-W Benchmark

When evaluating Web server performance, a *workload generator* is frequently used to drive the system in a hopefully representative manner. We use what is effectively the current standard workload generator for e-commerce sites, TPC-W [31, 43]. The TPC-W benchmark from the Transaction Processing Council (TPC) is a transactional Web benchmark specifically designed for evaluating e-commerce systems. It is meant to model a "typical" e-commerce site, in the form of an online bookstore. The TPC-W specification requires 14 different interactions, each of which must be invoked with a particular frequency. Of these interactions, eight initiate queries that result in data being written to the database, whereas the other six generate read-only queries. Each interaction may also involve requests for multiple embedded images, where each image corresponds to an item in the inventory. With one exception, all interactions query the database server. In our testbed, all persistent data is stored in the database, except for the static images used with each book. All images are kept in the file system of front-end Web server rather than stored in the database. The database contains multiple tables that are meant to represent the data needed to maintain a real site, including customers, addresses, orders, credit card information, individual items, authors, and countries. We scaled the TPC-W database to 10,000 items and 288,000 customers, which corresponds to 350 MB of data. The size of the static images, which represent the inventory (i.e., book covers), is 183 MB.

TPC provides a specification but not source code. We thus used the freely available TPC-W implementation developed by the Dynaserver project at Rice University [5, 36]. The Rice implementation captures all the functionality required by the TPC-W specification that affects performance, including transactional consistency and support for secure transactions. It does not implement some functionality specified that affects only price, such as the requirement to provide enough storage for 180 days of operation. More detail on the Rice TPC-W implementation may be found at the Dynaserver Web site and in their paper [5].

4.4 Client Workload Generator

The Rice TPC-W implementation includes a workload generator, which is a standard closed-loop session-oriented client emulator. Each emulated client represents a virtual user. The amount of load generated is determined by the number of emulated clients. Thus we use the *number of clients* to indicate the *load* on the system. Each client opens a session to the front-end Web server using a persistent HTTP connection, issues a series of requests for the duration of the session, and then closes the connection. Session duration is exponentially distributed with a mean of 15 minutes and a maximum of 60 minutes. Within each session, the client repeatedly makes a request, parses the server's response, waits a variable amount of time, and then follows a link embedded in the response. The server's response is a Web page consisting of the answer to the queries in the request, and contains links to the possible set of pages that the client can transition to from this response. A finite-state Markov model is used to determine which subsequent link from the response should be followed, using a transition matrix with probabilities attached to each transition from one state to another. Each state in the transition matrix corresponds to a particular interaction defined in the TPC-W specification.

The variable amount of time between requests is called the *think time*, and it is intended to emulate a real client who takes some period of time before clicking on the next request. Therefore, each client alternates between two states: either *thinking* to generate the next request or *waiting* until it receives the full response of the last request. TPC-W specifies that the think time should exponentially distributed with a mean between 7 to 8 seconds and is bounded at a maximum equal to ten times the average. In all our experiments where locking is done in the application server, think time has an average of 7 seconds and is bounded to 70 seconds. For locking in the database, think time has an average of 8 seconds and bounded to 80 seconds.

4.5 Metrics and Methodology

We evaluate throughput and response time, each as a function of the load. Throughput is the average number of successful requests that clients issue per unit time. Response time is the average amount of time it takes for a client to send a request and successfully receive the full reply. The measured response time includes the execution time on the Web, application and database servers as well as the queue waiting time (if any) inside the Gatekeeper proxy. However, the measured response time does not include the overhead of the Gatekeeper proxy (collecting and maintaining statistics, and sorting requests in ascending order of their expected service times in SJF scheduling). If a request fails or times out, it is not included in the measured throughput and response time, even though some components of the system may have executed parts of the request. Hence, throughput and response time are measured only for successful requests.

We use online measurements to estimate the load that each request imposes on the system in all experiments (except for the experiment that compares online measurements to offline measure-

ments). The database capacity is measured offline in all experiments.

During overload, there are more requests generated than the number requests successfully executed. When there is no admission control and when admission control is in effect with FIFO scheduling, requests that fail or time out during overload are distributed uniformly across all request types. However, when using admission control with SJF scheduling during overload, requests that fail or time out tend to be long requests because they stay much longer in the system than short requests.

In the graphs we present, each data point is the average of five runs, where each run is the average over a 600 second sampling period after a 100 second warm-up. Most graphs include 90 percent confidence intervals [26] calculated using the T distribution, which assumes that the distribution underlying the data is Gaussian.

5. EXPERIMENTAL RESULTS

In this section, we present our results in detail, showing the effectiveness of our techniques. In all experiments, the bottleneck is in the database machine, and the bottleneck resource is either the host CPU or lock contention. The front-end machine, which hosts both the Web server and the application server, has consistently low CPU utilization of about 30 percent. As will be seen, the single client machine is capable of driving the system both to saturation and beyond into overload.

5.1 Admission Control

Figure 5 shows the throughput of the system, when locking is done in the application server. The X-axis is the number of emulated clients, and the Y-axis is throughput in interactions per minute. Three curves are presented: the original system without admission control, marked "original"; the system using the Gatekeeper proxy with FIFO scheduling, marked "gatekeeper-FIFO", and the system using the Gatekeeper proxy with shorted-job-first scheduling, marked "gatekeeper-SJF." For small loads, the three systems behave similarly, up until roughly 50 clients where differences start becoming apparent. The original system without admission control reaches a maximum throughput of about 852 interactions per minute, but then starts degrading above 250 clients, falling to 589 interactions per minute at 300 clients. The system using the Gatekeeper proxy maintains a consistent throughput even at the higher loads, and exhibits a higher peak throughput than the original system, of 941 interactions per minute at 230 clients. We see that the Gatekeeper proxy is effective at preventing overload, and even improves peak performance by about 10 percent. Throughput is unaffected by changing the scheduling algorithm in Gatekeeper from FIFO to SJF.

Figure 6 illustrates the difference between the online and offline approaches to measuring servlet response time. The graph shows the same system as in Figure 5, but here the SJF curve is removed and a new curve is added showing throughput using offline measurements taken in isolation. While Gatekeeper using offline measurements does perform better than without any admission control, the curve shows degrading throughput after about 270 clients. Recall from Table 1 that service times change under high loads. Thus, the offline estimates are no longer accurate, leading to unstable behavior at high loads. These results emphasize the importance of taking into account the behavior of the system under heavy load.

While theory would predict no change in peak throughput using admission control, we see that this is not the case in practice. To determine why, we ran a number of profiling experiments. Table 2 shows various performance statistics for the system under 3 scenarios from Figure 5: the original system at peak throughput, the original system during overload, and the system using Gatekeeper at peak throughput. In all situations the database CPU is fully utilized, the database fits in memory, and the network bandwidth is not a limiting factor. Throughout the "plateau" in Figure 5, the statistics stay relatively consistent up until about 250 clients. Several factors contribute to the explanation why the system throughput degrades during overload, and why Gatekeeper provides better performance. First note that, while small, the I/O transfer rate goes up under overload, and shrinks when Gatekeeper is used. Second, there are a significantly lower number of database processes when Gatekeeper is used. Finally, memory pressure is clearly reduced, as the amount of memory used is much lower. Profiling measurements using the Athlon performance monitoring counters show that when the load is increased from 200 clients to 300, the rate of the L1 data cache misses increase 24.9 percent, L1 DTLB misses that causes a hit in the L2 DTLB increase 24.8 percent, and L1 DTLB misses that also cause a miss in L2 DTLB increase 22.5 percent. Clearly the data cache and DTLB misses contribute to the thrashing behavior of the system.

We now turn to the case where locking is done in the database, rather than in the application server; this case is illustrated in Figure 7. Note that the scales of this graph are different from the scales in Figure 5; the peak throughput here is much lower than in the previous graph. This shows how locking in the database is more expensive than locking in the application server. In this graph, the three curves show similar performance up to about 70 clients, after which the original system degrades quickly, but the experiments using Gatekeeper again demonstrate consistent performance even during overload. The original system reaches the peak throughput of 515 interactions per minute at 70 clients. At this point, the CPU utilization is 70 percent and all other system resources are far less utilized. As the load increases beyond this point, contention for the data locks causes thrashing, as two phase locking is used and many database queries become blocked. This is consistent with analytic models which show that the mean number of blocked transactions is a quadratic function of the total number of transactions [41]. This phenomenon explains the rapid decline in the throughput for the original system without Gatekeeper. Conversely, Gatekeeper queues any excess database requests when the estimated maximum capacity for the database is reached, and thus maintains peak performance in the overload region.

One question that might occur is whether our results are dependent on artifacts of a particular implementation, in this case MySQL. To test this notion, we ran our experiments using DB2 instead of MySQL, and the results are shown in Figure 8. Note that this experiment uses a different system, unlike the other figures, and thus results should not be compared across graphs. Gatekeeper again shows consistent throughput during overload, whereas the original system degrades quickly after 400 clients. These results demonstrate the applicability of our approach across database implementations whenever overload causes the database to thrash.

5.2 Request Scheduling

In this section we evaluate the effect of request scheduling using the Gatekeeper proxy. Figure 9 shows the average response time in our system, corresponding to the experiments shown in Figure 5, where locking is done in the application server. In this graph, the X-axis is again the number of emulated clients, but the Y-axis is response time in milliseconds. As expected, response times are low for small numbers of clients, but increase as a function of the load, as jobs queue up in the system. The original system, as can be seen, performs the worst. The system using the Gatekeeper proxy with FIFO scheduling gives better response time, as it provides bet-

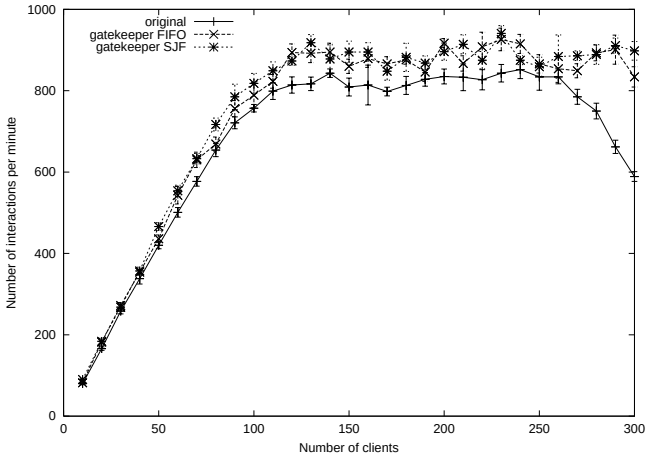

Figure 5: Throughput (MySQL, locking in application server).

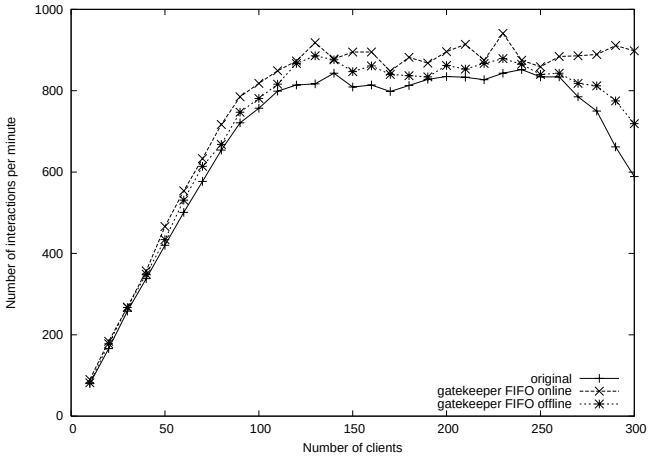

Figure 6: Online vs. offline (MySQL, locking in appl. server).

Statistic	Original	Overloaded	Gatekeeper
Throughput (interactions/min)	852	589	941
Number of Clients	200	300	230
Database CPU Utilization (user/system)	100 (76/24)	100 (74/26)	100 (75/25)
DB Memory Used (MB)	450	509	375
DB Memory Free (MB)	318	259	393
I/O Transfers/sec	36	43	33
Network Bandwidth (Mbits/sec)	1.1	0.9	1.2
Number of Processes (database/system)	266 (233/33)	378 (345/33)	82 (49/33)
Context Switches/sec	330	373	319
Interrupts/sec	219	220	219

Table 2: Performance statistics (MySQL, locking in application server.

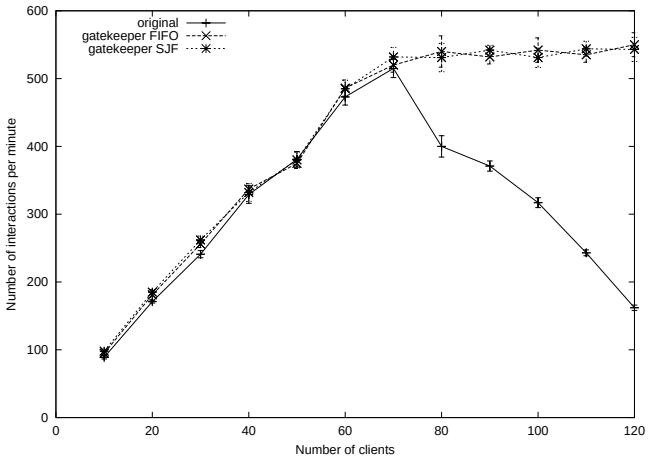

Figure 7: Throughput (MySQL, locking in database).

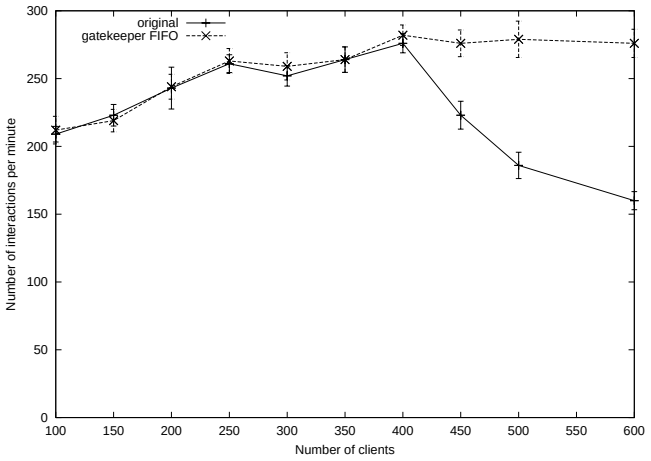

Figure 8: Throughput (DB2, locking in application server).

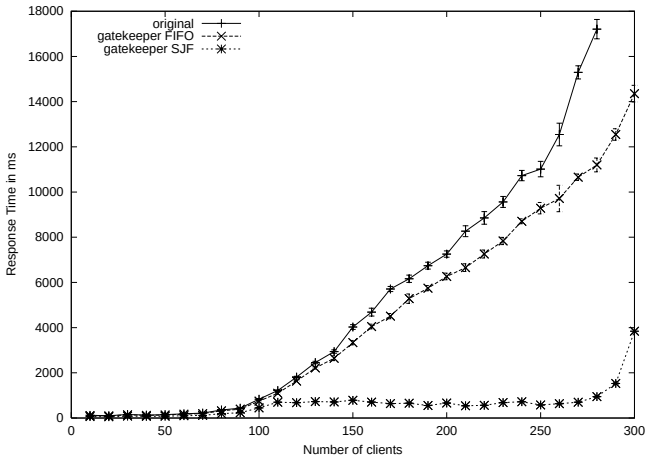

Figure 9: Response time (MySQL, locking in appl. server).

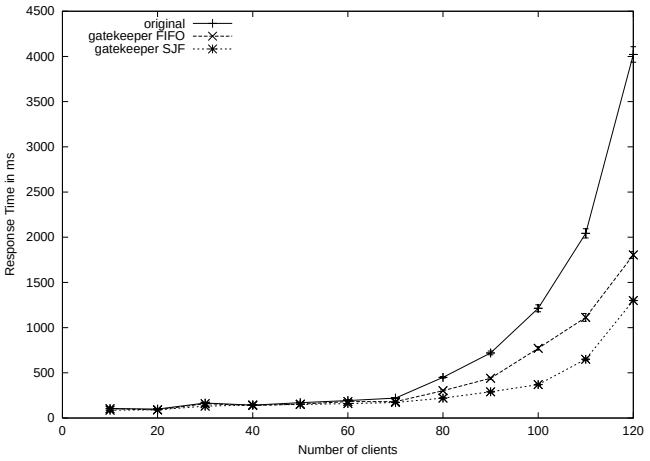

Figure 10: Response time (MySQL, locking in database).

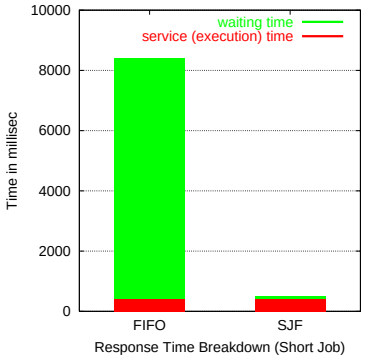

Figure 11: "Exec Search" request.

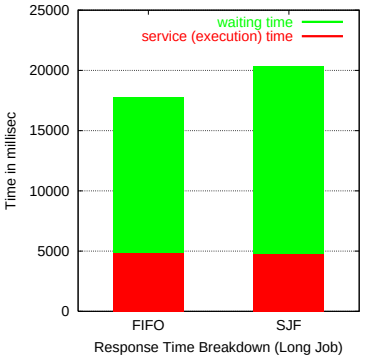

Figure 12: "Admin Response" request.

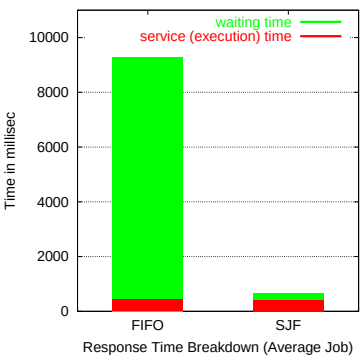

Figure 13: Average across all requests.

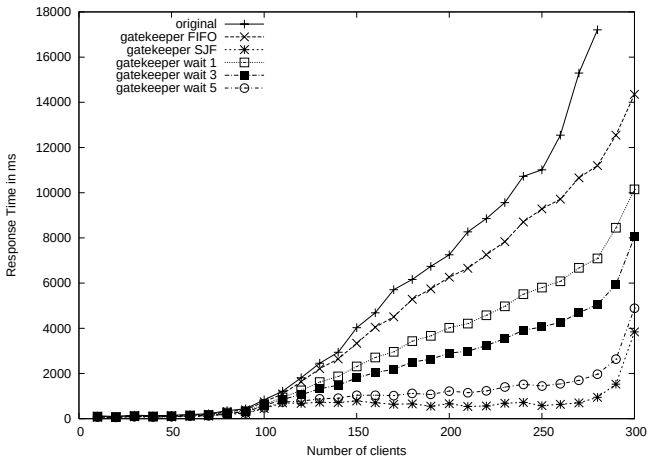

Figure 14: Response time (MySQL, Locking in appl. server).

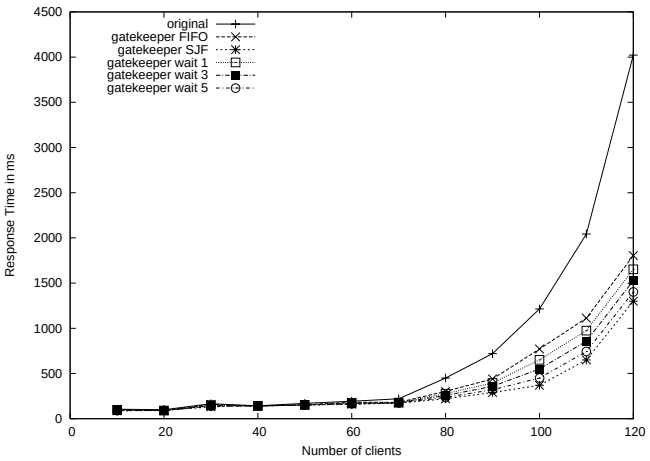

Figure 15: Response time (MySQL, locking in database).

ter throughput and thus lower service and waiting times. The curve showing the Gatekeeper proxy using SJF scheduling shows substantially better response times. For example, with 250 clients, the original system has an average response time of roughly 11 seconds, whereas with Gatekeeper, using FIFO scheduling provides a response time of 9 seconds, and using SJF scheduling gives a response time of about half a second. In this case, SJF improves response time by a factor of 22. Note that the knee of the SJF curve happens at about 290 clients in this experiment. While SJF scheduling can reduce response time substantially, it cannot prevent the queuing effects that occur with overload. If load exceeds capacity, and no load shedding is performed, response times will approach infinity. Scheduling can help stave off degrading response time under overload, but cannot prevent it.

Figure 10 shows the average response time in our system when locking is done in the database, corresponding to the experiments shown in Figure 7. While the absolute numbers are different, the trends are the same as in Figure 9. The original system has the worst response time, the system using Gatekeeper with FIFO scheduling is better, and the system using Gatekeeper with SJF scheduling performs the best. Here the magnitude of the improvement is not as great as in Figure 9, but the differences are still substantial, in this experiment by up to a factor of 4.

One concern about request scheduling using algorithms such as SJF is that large jobs will be penalized severely. Since we are providing preferential treatment to short requests, it is important to see what slowdowns may be incurred by large jobs. Figures 11, 12 and 13 show response times under both the FIFO and SJF policies, distinguishing execution time from waiting time. Three sample queries are shown: the "Exec Search" request, which is relatively inexpensive with an execution time of about 400 milliseconds, shown in Figure 11; the "Admin Response" request, which is complex and weighs in at roughly 4.8 seconds, shown in Figure 12, and the "Average" request, shown in Figure 13, the average across all queries in TPC-W , which is 425 milliseconds. Note that the scales of the y-axis are different across all Figures. As expected, execution times do not change in response to scheduling. Instead, improvements in response time are due to reductions in the waiting time of the requests. The "Exec Search" request waiting time shrinks from over 8 seconds using FIFO to 99 milliseconds using SJF, and the average waiting time falls from 8.8 seconds to 225 milliseconds. The large job, however, is penalized, with waiting time increasing from about 12.9 seconds to 15.6 seconds. Waiting time increases by about 21 percent, and overall response time, which includes the service time, increases by about 13 percent. While large jobs are penalized slightly, the slowdown is not significant, and the substantial benefit in overall response time makes this trade-off worthwhile.

Finally, we evaluate our mechanism to prevent starvation. Figure 14 shows response times using Gatekeeper with FIFO, SJF with no aging, and SJF with aging, all when locking is done in the application server. Figure 15 shows the corresponding response times when locking is done in the database. Figures 14 and 15 show the response time when three aging policies are used, where the label "gatekeeper wait X" corresponds to imposing an upper delay bound of X multiples of the expected service time. For example, "gatekeeper wait 1" permits requests to be reordered such that each request can be delayed for at most as long as its service time. This restrictive policy produces a graph that is closest to the FIFO policy. The response time is lower (better) than FIFO because the scheduler still has a degree of freedom to reorder the requests in such a way to reduce average response time. The curve marked "gatekeeper wait 5" presents a response time curve similar to the SJF policy. However, SJF yields better average response time because it does not have any restrictions on reordering requests. As can be seen, a continuum of behaviors is available between FIFO and SJF. Web site operators can choose a value of X to achieve a specific behavior based on their desired policy.

6. RELATED WORK

Much related work has been done in the areas of overload control, admission control, service differentiation, quality of service (QoS), and request scheduling for Web servers. Due to space limitations, we provide a very brief overview here.

Early works focused on delivering priority to one class of requests over another [3, 23]. Other research investigates admission control, sometimes referred to as overload control. Mogul and Ramakrishnan [32] showed how to prevent overload caused by interrupts generated by packet arrivals in software-based routers. Druschel and Banga [22] demonstrate a similar concept in the context of Web servers, showing how a network subsystem architecture can provide improved stability and throughput under high loads. Cherkasova and Phaal [17] show how considering session characteristics rather than individual requests in admission control can be used to reject fewer sessions. Chen et al. [15] use *computation quantums*, which are similar to the pool of *units* used to estimate the capacity of the database in our study. They develop an analytical model to perform admission control using a double queue structure and verify their results using a simulation study driven by a Web trace.

More recent approaches seek to combine differentiated service with admission control. Bhatti and Friedrich [11] propose an architecture for Web servers to provide QoS to differentiated clients, incorporating request classification, admission control, and request scheduling. They do not examine SJF or SRPT scheduling to improve response time, and most importantly, do not experimentally demonstrate sustained throughput in the presence of overload. Li and Jamin [28] provide an algorithm for allocating differentiated bandwidth to clients in an admission-controlled Web server based on Apache. Bhoj et al. [12] present the Web2K mechanism, which prioritizes requests into two classes: premium and basic. Connection requests are sorted into two different request queues, and admission control is performed using two metrics: the accept queue length and measurement-based predictions of arrival and service rates from that class. The authors evaluate their system using Apache, and show how high priority requests maintain stable response times even in the presence of severe overload. Voigt et al. [44] study different kernel and user-space mechanisms for admission control and service differentiation in overloaded Web servers. They evaluate their proposed mechanisms in AIX and find that the kernel-based mechanisms provide better performance. Pradhan et al. [35] present an observation-based framework for "self-managing" Web servers that adapt to changing workloads while maintaining QoS requirements of different classes. Kanodia and Knightly [27] propose a mechanism that integrates latency targets with admission control. Using both request and service statistical envelopes, the mechanism improves the percentage of requests that meet their QoS delay requirements. The authors evaluate their scheme via trace-driven simulation.

Several researchers have examined how control theory can be applied in the context of Web servers. Lu et al. [29] present a control-theoretic approach to provide guaranteed relative delays between different service classes. Abdelzaher et al. [2] propose using classical control theory for Web servers to provide performance isolation, service differentiation, and QoS adaptation. They provide an implementation using the Apache Web server. Diao

et al. [21] advocate a similar approach, using control theory to maintain Apache's KeepAlive and MaxClient parameters, showing quick convergence and stability. However, these parameters do not directly address metrics of interest to the Web site, such as response time or throughput.

Heiss and Wagner [24] perform a simulation study of thrashing in transaction systems. They present two admission control algorithms that prevent overload by limiting the amount of concurrency in the system.

Instead of performing admission control and refusing clients as a response to overload, several researchers have investigated using *service degradation*. In this context, the service offered to clients is reduced, in the form of providing smaller content, e.g., lower resolution images. Different approaches include using HTTP's content negotiation feature [38], content substitution [1], or transcoding [14] to improve Web client response times and server throughput in the presence of server overload or network congestion.

Recently a number of researchers have observed the value of integrating resource management with service differentiation, admission control, and quality of service. Banga et al. [9] propose resource containers as an operating system abstraction that embodies a resource, improving robustness and control over priorities. Aron et al. [8] build on this notion by recognizing the fundamental connection between resource allocation and providing predictable quality of service. Their evaluation includes dynamic content based on a trace from Google that includes an average service time, but do not include database-driven workloads. Resource management approaches have also been used in clusters of servers [7, 39].

Welsh and Culler [45] describe an adaptive approach to overload control in the context of the SEDA [46] Web server. SEDA decomposes Internet services into multiple stages, each one of which can perform admission control. By monitoring the response time through a stage, each stage can enforce a targeted 90th-percentile response time. Their evaluation includes dynamic content in the form of a web-based email service.

Gatekeeper differs from the above works in many respects. Most importantly, most of the above work has only addressed static content, a much simpler workload, whereas Gatekeeper is fundamentally concerned with dynamic content and database-driven Web sites. Of the few that do consider dynamic content, two use a simple linear approximation of the service cost in the form of a dummy CGI script [12, 35]. We use a full implementation of the dynamic functionality and incur actual execution costs, which can vary orders of magnitude.

Relatively few works are closely related to ours in terms of being implemented and addressing dynamic content: Neptune [39], Aron's resource management framework [8], McWherter's priority mechanisms for transactional Web applications [30], and SEDA [45, 46]. Neptune and Aron's framework both use search as a dynamic workload, whereas we use a transaction-oriented e-commerce workload. McWherter implemented preemptive and non-preemptive prioritization algorithms by modifying the database server in order to provide differentiated performance for some requests. SEDA's evaluation includes dynamic content in the form of a custom email service driven by a home-grown workload generator. Our evaluation is performed using standard software components and driven using an industry-standard e-commerce workload. While SEDA's approach is perhaps more general, our approach has the advantage that it is completely transparent to the database and is thus more easily deployable.

Finally, many previous works have attempted to identify overload through indirect measurements such as queue length or bandwidth utilization. Others have taken more direct indicators such as response time, but measured them for the entire system or for class-based categories. We address the concept of load directly by measuring how per-script resources are used. We believe this mechanism provides more accurate and finer-grained approach to admission control, and is in fact complementary to other approaches. For example, SEDA allows different scripts to be handled by different stages. By including both an overall notion of system capacity with an estimate of per-script execution costs, the number of rejected requests at a given load could be lowered.

7. SUMMARY AND CONCLUSIONS

This paper presents a method for providing admission control and request scheduling for multiply-tiered e-commerce Web sites. By externally measuring service costs online and distinguishing between different types of requests, our approach can achieve both stable behavior during overload and dramatically improved response times. Other proposals require extensive modifications to the operating system or a complete re-write of the server. Our approach requires no changes to the source code, server software, application programs, or to the database. This allows ease of deployment, database independence, and use of standard software components.

Our method is embodied in a transparent proxy called Gatekeeper. Gatekeeper intercepts requests from the application server to the database, allowing interoperability with standard software components. We evaluated Gatekeeper experimentally using the Apache Web server, Tomcat servlet engine, and both the MySQL and DB2 databases, using the industry-standard TPC-W workload generator. Despite being external to the system, our proxy provides many of the benefits of admission control and request scheduling:

- We show *consistent performance* even in the presence of persistent overload, across database implementations and locking approaches.
- We demonstrate *improved peak throughput* of 10 percent, due to reduced thrashing and better memory reference behavior.
- We demonstrate *drastically reduced response times* via preferential scheduling using the shortest-job (SJF) scheduling policy. Average response time is reduced by a factor of 14, whereas large queries are penalized by only 15 percent.
- We present an aging mechanism that *prevents starvation* of jobs under SJF. The mechanism allows decisions of where in the continuum between FIFO and SJF a Web site should operate based on operator policy.

These results show that extensive modifications to operating systems, and servers are not always necessary in the context of e-commerce Web sites, where the database is the bottleneck. A natural next step for future work is to evaluate our approach using other dynamic workloads. In cases where the bottleneck resource is the application server, rather than the database, it would make sense to place the Gatekeeper proxy between the front-end Web server and the application server.

8. REFERENCES

[1] T. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. *Computer Networks*, 31(11–16):1563–1577, 1999.

[2] T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1), January 2002.

[3] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in Web content hosting. In *Workshop on Internet Server Performance*, Madison, WI, June 1998.

[4] AMD Corporation. AMD Athlon processor x86 code optimization guide. `www.amd.com`.

[5] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic Web site benchmarks. In *Proceedings of the 5th Workshop on Workload Characterization*, Austin, Texas, November 2002.

[6] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large Web-based shopping system. Technical Report HPL-2001-XX, HP Labs, April 2001.

[7] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *ACM SigMetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[8] M. Aron, S. Iyer, and P. Druschel. A resource managment framework for predictable quality of service in Web servers. Preprint vailable at `http:/www.cs.rice.edu/~ssiyer/r/mbqos/mbqos-full.pdf`, 2002.

[9] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[10] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into Web server design. Technical Report HPL-2000-3, HP Labs, January 2000.

[11] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, September 1999.

[12] P. Bhoj, S. Rmanathan, and S. Singhal. Web2K: Bringing QoS to Web servers. Technical Report HPL-2000-61, HP Labs, May 2000.

[13] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic Web content. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, Rio de Janeiro, Brazil, June 2003.

[14] S. Chandra, C. Ellis, and A. Vahdat. Differentiated multimedia Web services using quality aware transcoding. In *IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[15] X. Chen, P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time for Web accesses. In *Proceedings of the 10th World Wide Web Conference*, Hong Kong, May 2001.

[16] L. Cherkasova. Scheduling strategy to improve response time for Web applications. In *Proceedings High Performance Computing and Networking (HPCN)*, Amsterdam, April 1998.

[17] L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial Web sites. *IEEE Transactions on Computers*, 51(6), June 2002.

[18] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Nov 1997.

[19] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in Web servers. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, Boulder, Colorado, Oct 1999.

[20] P. J. Denning. A short theory of multiprogramming. In *3rd International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems (MASCOTS)*, Durham, North Carolina, January 1995.

[21] 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. In *Proceedings of the Network Operations and Management Symposium*, Florence, Italy, April 2002.

[22] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, pages 261–275, Seattle, WA, 1996.

[23] L. Eggert and J. Heidemann. Application-level differentiated services for Web servers. *World-Wide Web Journal*, 2(3):133–142, August 1999.

[24] H.-U. Heiss and R. Wagner. Adaptive load control in transaction processing systems. In *17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[25] D. Heller. Rabbit performance counters library. `http://www.scl.ameslab.gov/Projects/Rabbit/`.

[26] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.

[27] V. Kanodia and E. W. Knightly. Ensuring latency targets in multiclass Web servers. *IEEE Transactions on Parallel and Distributed Systems*, 13(10), October 2002.

[28] K. Li and S. Jamin. A measurement-based admission-controlled Web server. In *IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[29] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in Web servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, June 2001.

[30] D. T. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *20th International Conference on Data Engineering (ICDE 2004)*, Boston, MA, March 2004.

[31] D. A. Menasce. TPC-W: A benchmark for e-commerce. *IEEE Internet Computing*, May/June 2002.

[32] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[33] News.Com. E-commerce strong in third quarter. `http://news.com.com/2100-1017-971123.html`, November 2002.

[34] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[35] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An observation-based approach towards self-managing Web servers. In *International Workshop on Quality of Service*, Miami Beach, FL, May 2002.

[36] Rice University Computer Science Department. The Dynaserver project. `http://www.cs.rice.edu/CS/Systems/DynaServer`.

[37] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. Technical Report CMU-CS-02-143, Carnegie-Mellon University C.S. Department, Pittsburgh, PA, July 2002.

[38] S. Seshan, M. Stemm, and R. H. Katz. Benefits of transparent content negotiation in HTTP. In *Proceedings of the IEEE Globcom 98 Internet Mini-Conference*, Sydney, Australia, November 1998.

[39] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based Internet services. In *Operating Systems Design and Implementation*, Boston, MA, December 2002.

[40] Sysstat Project. Systat home page. `http://freshmeat.net/projects/sysstat`.

[41] Y. C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, December 1985.

[42] The Apache Project. The Apache WWW server. `http://httpd.apache.org`.

[43] The Transaction Processing Council (TPC). TPC-W. `http://www.tpc.org/tpcw`.

[44] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[45] M. Welsh and D. Cluller. Adaptive overload control for busy Internet servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2003.

[46] M. Welsh, D. Cluller, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.