

Request-Aware Scheduling for Busy Internet Services

Jingyu Zhou*[§] Caijie Zhang*[§] Tao Yang^{§*} Lingkun Chu[§]

[§] Ask Jeeves Inc.

* University of California at Santa Barbara

Abstract—Internet traffic is bursty and network servers are often overloaded with surprising events or abnormal client request patterns. This paper studies scheduling algorithms for interactive network services that use multiple threads to handle incoming requests continuously and concurrently. Our investigation with applications from Ask Jeeves search shows that during overloaded situations, requests that require excessive computing resource can dramatically affect the overall system throughput and response time. The most effective method is to manage resource usage at a request level instead of a thread or process level. We propose a new size-adaptive request-aware scheduling algorithm called SRQ with dynamic feedbacks to control queue properties and have implemented SRQ in the Linux kernel level. Our experimental results with several application service benchmarks indicate that the proposed scheduler can significantly outperform the standard Linux scheduler.

I. INTRODUCTION

Busy Internet service providers often use a service level agreement in terms of response time and throughput to guide performance optimizations and guarantees. For instance, when Ask Jeeves [2] provides search results to other portals, the goal is to complete 99% of requests within a second and to achieve the average response time within a few hundreds of milliseconds.

It is challenging to satisfy performance requirements of requests at all times because Internet traffic is bursty, and resource requirement for dynamic content is often unknown in advance. Even with over-provisioning of system resources, a web site can still be overloaded in a short period of time due to slash-dot effects under an unexpected high request rate [10], [25]. Sometimes even though the amount of traffic does not increase sharply, the characteristics of traffic change dramatically due to some abnormal situations. For example, the percentage of long requests increases significantly and these long requests take over most of the system resources. As a result, the system is unable to handle other incoming requests and becomes overloaded.

Previous work has been using admission control [21], [37], [39], [40], [43] and adaptive service degradation [4], [12], [43] to curb response time spikes during overload. Admission control improves the response time of admitted client requests by rejecting a subset of clients. Admission control techniques include using 90th-percentile response time [43], bounding the incoming request queue length [21], [37], and policing TCP SYN packets [40].

Complementary to the above admission control and adaptive service degradation mechanisms, we propose a request-aware scheduling approach for improving quality of service. This comes from the observation that the scheduling principal for Internet services is an individual request. The traditional operating systems or previous network services use processes or threads to handle incoming requests and scheduling principals are these processes or threads. It would be much more effective if the system can perform fine-grained scheduling at the request level for admission controls and performance optimizations. In this way, we can differentiate long requests and short requests during load peaks and prioritize resource for short requests. The system throughput can be optimized and the impact of long requests is minimized. Our new SRQ algorithm combines multi-level feedbacks with soft deadlines, and has been implemented within a request-aware scheduling mechanism at the Linux kernel level, which also supports a variety of different scheduling policies. We have conducted experiments using a number of applications from Ask Jeeves to demonstrate the effectiveness of the proposed techniques.

The rest of the paper is organized as follows. Section II discusses the background on multi-threaded Internet services and scheduling policies. Section III discusses the design of the SRQ scheduling framework. Section IV describes our implementation in Linux. Section V evaluates request-aware scheduling with real applications. Section VI summarizes related work. Finally, Section VII concludes the paper.

II. BACKGROUND

In this section, we describe the context of Internet services and their implementation with the multi-threaded programming model. Then we summarize the related scheduling policies for Internet services.

A. Multi-threaded Programming Model for Internet Services

An Internet service cluster hosts applications handling concurrent client requests on a set of machines connected by a high speed network. A number of earlier studies have addressed providing middleware-level support for service clustering, load balancing, and replication management [13], [37], [41]. In these systems, a service component can invoke RPC-like service calls or obtain communication channels to other components in the cluster. A complex Internet service cluster

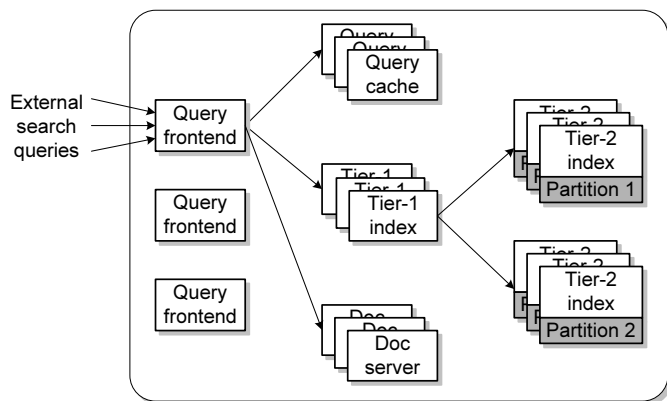


Fig. 1. A three-tier keyword-based document search service.

often has multiple tiers and service components depend on each other through service calls.

For example, Figure 1 shows the architecture of a three-tier search engine. This service contains five components: query handling front-ends, result cache servers, tier-1 index servers, tier-2 index servers, and document servers. When a request arrives, one of the query front-ends parses this request and then contacts the query caches to see if the query result is already cached. If not, index servers are accessed to search for matching documents. Note that index servers are divided into two tiers. Search is normally conducted in the first tier while the second tier database is searched only if a first tier index server does not contain sufficient matching answers. Finally, the front-end contacts the document servers to fetch a summary for relevant documents. There can be multiple partitions for cache servers, index servers, and document servers. A front-end server needs to aggregate results from multiple partitions to complete the search.

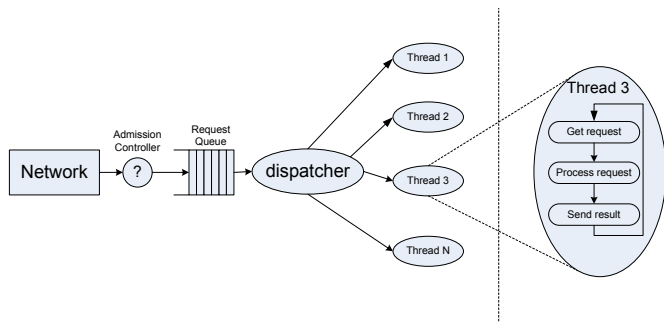


Fig. 2. Multi-threaded server design.

As discussed above, each machine in a service cluster handles requests concurrently sent from another machine in the cluster or from clients through Internet. Multi-threaded programming is widely used for concurrency control in Internet services such as Apache [1] and IIS [29] Web servers, BEA WebLogic application server [7], and Neptune clustering middleware [37]. As shown in Figure 2, each machine maintains a set of active worker threads and each accepted request

is dispatched to a worker thread for processing. Each worker thread performs a continuous loop to receive a new request, process it with an application logic, and then send results back. The thread pool size typically has a limit in order to control resource contentions in a heavy loaded situation. Additionally, admission control may be used to regulate incoming requests during a busy period.

B. CPU Schedulers

In this section, we discuss existing CPU schedulers and their suitability for Internet services.

- Real-time systems:** These systems typically have hard deadlines and conventional schedulers such as Earliest Deadline First (EDF) and Rate Monotonic Algorithm (RMA) are suitable for these systems [3], [23], [28], [38]. These systems guarantee a maximum delay an application may experience with a number of restrictions. For example, in RTLinux [3], a hard real-time operating system, real-time tasks are written as special Linux modules and are loaded into kernel dynamically, which creates security problems because any bugs in real-time tasks can corrupt and crash the kernel. Additionally, RTLinux only provides a subset of POSIX APIs and real-time tasks cannot use virtual memory. All these limitations are preventing many applications from being ported to the RTLinux platform. Soft-deadline scheduling for aperiodic tasks have been studied in [26], [33]. The focus is to minimize response time of aperiodic tasks with soft deadlines and to avoid missing hard deadlines of periodic tasks. Generally, such a scheduler tries to “steal” all the possible slack time from the periodic tasks. These studies have assumed FIFO ordering for processing aperiodic tasks and exponential workload.

Internet services often have soft deadlines instead of hard deadlines and requests may be served out-of-order considering their deadlines or importance. Additionally, applications exhibit a wide variety of size distributions such as heavy-tailed distribution, which calls for more adaptive scheduling algorithms.

- General purpose operating systems:** The general purpose OS schedulers (e.g., Linux, Windows, and Solaris) employ multi-level feedback queues (MFQ) to use process ages as feedbacks for prioritization. Each process is initially placed in a high priority queue and is gradually moved from higher priority queues to lower ones, i.e., the priority decays with its age. A process is preempted if another process in a higher priority queue becomes ready. When there are several processes inside one queue, the scheduler can use a different policy such as round-robin. These systems provide best effort resource sharing among processes so that throughput is high while response time is relatively low. The drawback is that scheduling principals for all above schedulers are processes or threads. In addition, deadline is not considered, while for Internet services each request can have a soft deadline that can be exploited. When a thread is dealing with a sequence of

requests for Internet services, it is hard to prioritize such a thread without knowing the characteristics of requests being handled. A fine grained scheduler that works at request level can allow us to perform more advanced optimizations, which will be described in more details in next section.

III. DESIGN

This section begins with a motivating example of request-aware scheduling. Then two design choices are discussed and a new SRQ scheduler is presented.

A. Request-aware Scheduling

In a general purpose operating system such as Linux, there is a discrepancy for Internet services in terms of scheduling principals. The Internet services are request-driven and the perceived performance is measured by individual requests. It is desirable that the operating systems use requests as scheduling principals. However, the request is an application level concept unknown to the operating system kernels. In a multi-threaded programming model, as in Figure 2, the actual scheduling entities used by the operating systems are threads.

When the system is busy under a high request rate or size distribution shift, a thread may serve multiple requests and only relinquishes CPU after using up its time quantum. In this case, thread-level scheduling deviates from request-aware scheduling and delivers sub-optimal performance. The thread scheduling matches closely with request-aware scheduling if the system is only lightly loaded. Because in this case there exist free threads waiting for new requests, a new request will be served by one of the free threads. The chance of a thread continuously serving multiple requests is low. Our evaluation in Section V confirms this by showing similar performance of request-aware schedulers and the Linux kernel in lightly loaded scenarios.

We use Figure 3 to illustrate the performance difference of request-oblivious and request-aware scheduling. In the request-oblivious case, let scheduling quantum be 10 ms and let threads A and B have the same priorities at time 0. These two threads will alternate their executions every 10 ms. Note that requests 3 and 4 can only be served by thread B from time 100 ms to 290 ms, because A hasn't finished processing of request 1.

On the other hand, a request-aware scheduler can improve the average response time. As shown in Figure 3 also, threads A' and B' are managed by a request-aware scheduler, which decreases thread priority after servicing a request for 50 ms. At time 90 ms, thread A' has processed request 1 for 50 ms and is given a low priority. Thread B' gains a higher priority when starting servicing new requests (3 and 4), thus can execute requests 3 and 4 without interruption from thread A' . Here request 4 happens to be served before request 3. As a result, the response time of request 4 is reduced from 200 ms in request-oblivious scheduling to 50 ms in request-aware scheduling. The response time of request 1 has a minor increase from 290 ms to 300 ms.

To summarize, although a traditional scheduler performs well in an underloaded situation, it can be significantly outperformed by a request-aware scheduler with finer-grained control during overloaded cases.

B. Design Choices

There are two questions for a request-aware scheduler: 1) Is a user-level implementation or a kernel-level implementation more appropriate? 2) What scheduling algorithm works well for this scheduler? The following discusses these two design choices.

1) *User-level vs. Kernel-level Implementation:* A user-level implementation is more portable across different platforms and may seem easier to develop, but has a number of problems.

- Being at user-level, the implementation has no control over which thread is to run currently. Changing thread priority with `nice(1)` is tricky, because the dynamic priority also depends on recent CPU usage of the thread. Forcing some threads to sleep can decrease the CPU utilization, because the concurrency is reduced.
- Accurate timing and quick thread preemption is difficult. The implementation needs to be able to preempt a running thread, which could be implemented using signals sent by a dedicated thread. Though the granularity of timer on the default Linux 2.6 kernel is 1 ms, the time needed for notifying the dedicated thread when to send a signal and the delay of signal delivery are prolonged in heavy load. Thus, the thread preemption may not be performed at a fine granularity.

A kernel-level implementation, on the other hand, can overcome the above problems. Priority and preemption are readily supported in the kernel scheduler. The kernel naturally decides the running thread. Thread preemption can be performed at timer granularity by modifying timer interrupt handler. The drawbacks are platform-dependent.

Our initial implementation is a user-level thread package, which was abandoned after realizing the problems with it. The current implementation is at the kernel-level and the rest of the paper discusses this implementation.

2) *Scheduling Algorithms:* For a request-aware scheduler, there are three possible scheduling parameters: age of a request (elapsed time since the request is received), a deadline of the request, and a value or request yield associated with the request. EDF and VRD [20] take deadline and value as input. None of these schedulers consider all three scheduling parameters. In fact, our evaluation of these schedulers during size distribution shift in Section V-C shows that none is satisfactory, which motivates us to design a new scheduler, given below, with all three parameters as input.

C. SRQ Scheduler

We propose a new SRQ scheduler for Internet services, which combines dynamic feedbacks and deadline scheduling together. Specifically, dynamic feedback or the size of a request is the first scheduling discipline, while the deadline is used as the second discipline. Dynamic feedback is applied

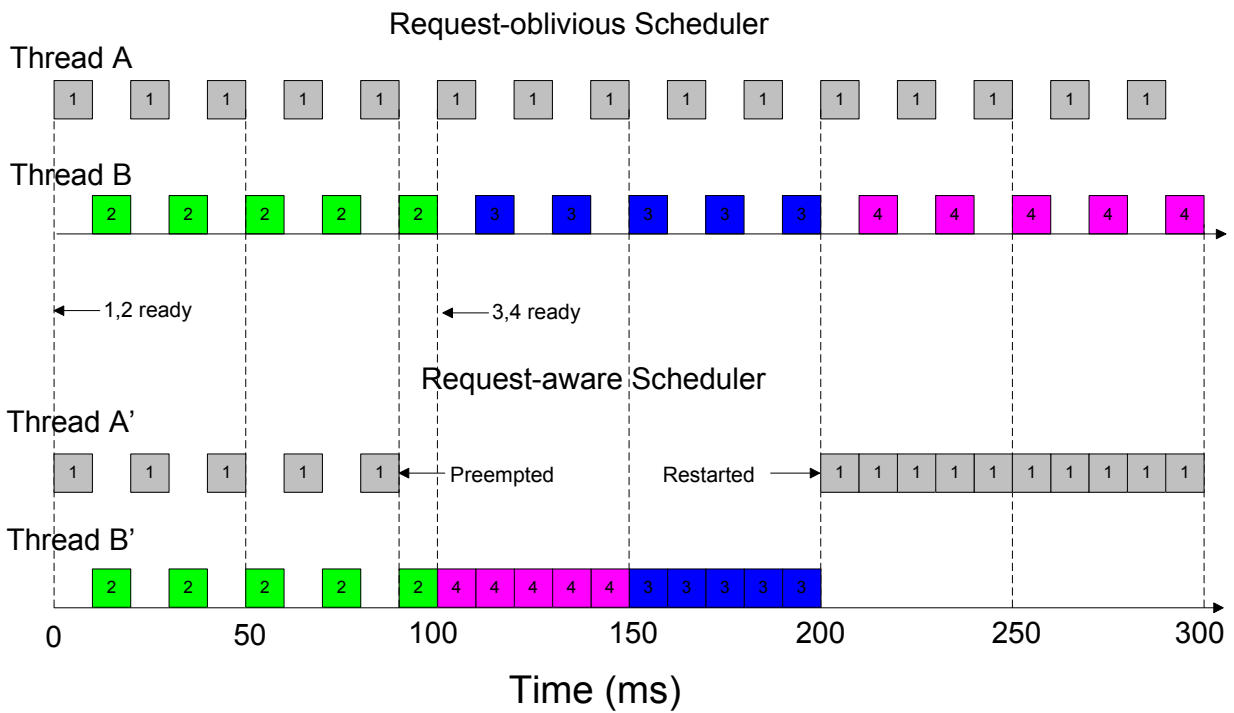


Fig. 3. Illustration of request-aware scheduling for response time improvement. Threads A and B are dispatched with an ordinary scheduler, while threads A' and B' are managed by a request-aware scheduler that decreases the priority a thread after servicing a request for 50 ms. The response time of request 4 is reduced from 200 ms to 50 ms, and only request 1 is delayed for 10 ms. We assume each scheduler gives 10 ms quantum for each thread in this example and the priorities at time 0 for all threads are the same.

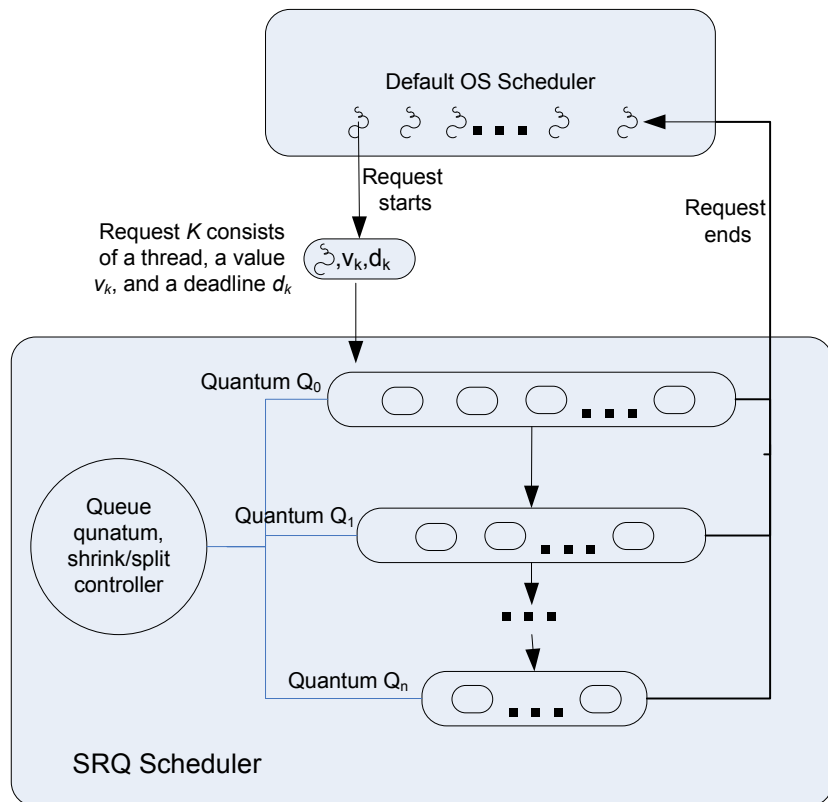


Fig. 4. The SRQ scheduler. Note each request is associated with a deadline and a value (or yield).

first so that short requests can preempt long requests and complete first, approximating the theoretical best Shortest Remaining Processing Time (SRPT) [6] scheduler.

Given a set of threads executing on different incoming requests, we will dynamically compute the priorities of these threads by tracking the status of these requests being processed by these threads. The request status tracking mechanism in SRQ is illustrated in Figure 4. The dynamic feedback is achieved with a number of request queues. A queue quantum is associated with all requests in the same queue, but different queues may have different quanta. A request is first put into the first queue and is gradually moved to the next queue if uses up current queue quantum. A request exits the scheduler after its completion. Note that the idea of using multiple queues is similar to the MFQ, but a significant difference is that the scheduling principals in SRQ are requests, not threads.

Adaptive queue management. The goal of the SRQ controller is to distinguish requests of different sizes and to group requests of similar size together so that they can finish in the same queue. The questions are how many queues should be actively maintained and what are the quanta for these queues? The controller monitors the previously executed requests and then uses the statistical information to determine the number of priority queues and scheduling quanta so that short requests are separated from long requests and can complete in the first few queues.

Specifically, the scheduler monitors *fail-percent*, *deviation*, and *average* elapsed times of each queue periodically. The fail-percent is the percentage of requests moving to the next queue during the time period. For all the requests completed within the period, the deviation and average elapsed times represent the standard deviation and the mean of the elapsed time of these requests since they are received in this machine. The system adjusts the number of queues and quantum of each queue periodically, guided by the following rules:

- **Queue shrinking** happens when no request is completed for the last queue since the last queue adjustment time. It causes the scheduler to merge the last two queues. Here queues are ordered based on the average elapsed time of requests within each queue. The last queue is the queue containing requests with the largest elapsed time.
- **Queue splitting** occurs if the ratio of the deviation and the average elapsed time in the last queue exceeds a pre-defined threshold T_{split} .
- **Queue quantum** is updated as follows: if the fail-percent exceeds a threshold T_{fail} , which means less requests are completed within the queue and more requests have to be moved to next queue, then the quantum of the current queue is increased by a constant factor adj_i (quantum increase). If the ratio of the deviation over the average exceeds a threshold T_{desc} , the quantum is decreased by the average times a constant factor adj_d (quantum decrease).

This controller has two advantages: short and long requests are separated, and the variation within the same queue is small

minimizing the adverse effect that long requests have on short requests.

Running thread selection. At each scheduling step, the SRQ scheduler needs to choose a queue and then selects a request from this queue. The thread that processes this request will be switched in and executed by the CPU.

In most of times we select requests from the first queue containing all requests with the shortest elapsed time. In this way, short size requests get a high priority for execution. To ensure a reasonable fairness, we impose a proportional selection policy such that the first queue will be selected $X\%$ of chance while the rest of queues will be considered proportionally with $(100-X)\%$ of chance. In our implementation, $X=90\%$.

Priority computation. When there are multiple requests in a queue, the SRQ scheduler uses relative deadlines and values of these requests for priority calculation:

$$P = \alpha \times Deadline + (1 - \alpha) \times (MAX_VAL - Value),$$

where $0 \leq \alpha \leq 1$ and lower P means high priority. When α is zero, this policy becomes highest value first; when α is one, this policy is earliest deadline first; in all other cases, the algorithm is between these two policies.

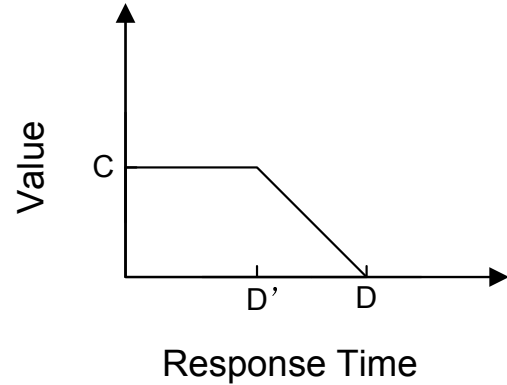


Fig. 5. An example value function.

For each request, its value is a function of response time. As shown in [35], different QoS metrics, such as throughput and response time, can be captured using this method, and the overall goal of provisioning resource is expressed in terms of maximizing the aggregate values. Figure 5 gives an example of value function: if a request is completed before a soft deadline D' , full value C is realized; if the request is completed after a hard deadline D , there is no service yield; between D' and D , a partial value is realized.

Triggering of SRQ and summary of scheduling parameters. The parameters used in the implementation are summarized in Table I. These parameters are determined using trace-driven experiments with real applications.

The SRQ scheduler is triggered during busy periods, which can be identified by querying the admission controller. For instance, our implementation uses the Neptune threshold of queue length as the indicator of overload.

TABLE I
PARAMETERS USED IN THE SRQ SCHEDULER.

Parameter	Description	Default Value
α	Deadline weight constant	0.2
$interval$	Time before adapt SRQ	60 s
T_{fail}	Fail percentage threshold	60%
T_{split}	Queue split threshold	1.0
T_{desc}	Quantum decrease threshold	1.0
adj_i	Quantum increase factor	2.0
adj_d	Quantum decrease factor	0.2
D, D'	Request soft deadline	2.0 s
C	Request yield value	1.0

The combination of different techniques allows the SRQ scheduler to be effective for different workload. Firstly, the request-aware scheduling is more fine-grained than the process scheduling of the operating systems, thus the SRQ scheduler can make better scheduling decisions. Secondly, the adaptive queue management enables the scheduler to use dynamic feedback to favor short requests, which is beneficial to heavy-tailed distributions. Finally, the variance of jobs within the same priority queue is low, which reduces the adverse effect that long requests have on short requests.

IV. IMPLEMENTATION

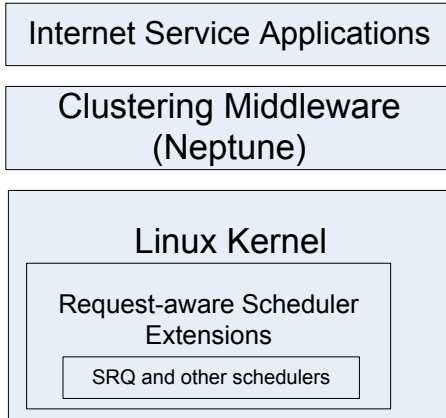


Fig. 6. System architecture.

We have implemented SRQ in the Linux kernel and have integrated it with the Neptune clustering middleware [35], [37]. Note that Neptune has been used in Ask Jeeves websites and internal sites supporting many different Internet and data mining services running on several thousands of machines at each data center. Figure 6 illustrates the system architecture of our implementation. Changes are made to the Linux 2.6.11 kernel and to the Neptune middleware in order to realize request-aware scheduling. One big advantage of this approach is that service applications can still run unmodified binaries under this new SRQ scheme.

A. Changes to the Linux Kernel

The Linux kernel is modified to create a scheduler extension interface and a new system call. The scheduler extension interface consists of a set of callback functions for implementations of different scheduling algorithms, round-robin for example. The new system call allows a user process to notify the kernel the start and the end of a request, which is further discussed in the next part of this section.

We have implemented Request-aware EDF (R-EDF), Request-aware VRD (R-VRD), and SRQ scheduler as separate loadable kernel modules by realizing the extension interface. These callbacks are inserted in the kernel at points related to making scheduling decisions, i.e., the timer interrupt handler, the `schedule()` function, process migration code (i.e. SMP load balancing code), and blocking or unblocking of a process. Data structures related to each scheduler are carefully designed on a per CPU basis to avoid global locks. We do not change the SMP load balancing logic and only make necessary modifications to keep data consistency, though it is possible to exploit multiple CPUs of SMP machines [18].

For us, this loadable module approach proves to be better than directly modifying the kernel for three reasons. First, the change made to the kernel is minimal and easy to verify its correctness. Second, the loadable kernel module only needs to implement the scheduling interface independent of other components of the kernel. The loadable module also enables testing a new scheduler simply by reloading the module without reboots. Finally, the scheduling interface can help for developing other more sophisticated schedulers, for example, a hierarchical scheduler with proportional guarantees [14].

B. Changes to the Neptune Clustering Middleware

```

#include <sys/syscall.h>
#define __NR_svr 289 /* system call # */
#define SVR_START 1
#define SVR_END 2

void worker()
{
    struct scheduler_ext_t arg;

    while (1) {
        Request *request = get_request();

        setup_arg(&arg);
        syscall(__NR_svr, SVR_START, sizeof(arg), &arg);
        process_request(request);
        send_result(request);
        syscall(__NR_svr, SVR_END, 0, NULL);
    }
}
  
```

Fig. 7. A pseudo-code example of invoking request-aware schedulers.

A user process communicates with request-aware schedulers via a new system call interface. Figure 7 gives a pseudo-code example, where the start and end of a request are made explicit to the kernel using different system call parameters. In our implementation, for instance, deadlines are passed to the kernel schedulers as part of the arguments. The Neptune

middleware [37] is easily modified by adding about thirty lines of code.

V. EVALUATION

In this section, we mainly seek to evaluate the performance of request-aware schedulers in busy workload situations. The request-aware schedulers are R-EDF, R-VRD, and the proposed SRQ. Our main objectives are:

- 1) Demonstrate request-aware schedulers have better response time while maintaining comparable throughput in overloaded situations, compared to request-oblivious thread-level scheduling in Linux.
- 2) We compare SRQ with simple request-aware scheduling algorithms such as R-EDF and R-VRD built on our request queue management scheme and illustrate the proposed SRQ scheduler is able to deliver better performance and has negligible overhead.

A. Settings and Applications

Experiments were conducted on a cluster of nine dual Pentium III 1.4 GHz machines connected with fast Ethernet. Each machine has 4GB memory and two 36GB, 10,000 RPM Seagate ST336607LC SCSI disks. These disks have 8MB buffer, the average seek time is 4.7ms, and the average latency is 2.99ms.

The applications used for the evaluation are two services from Ask Jeeves [2] with different size distribution characteristics: a database *retrieval* service and a page *description* service. The retrieval service that finds all web pages containing certain keywords is heavy-tailed. The description service that finds a paragraph in a web page containing the search keywords is exponential.

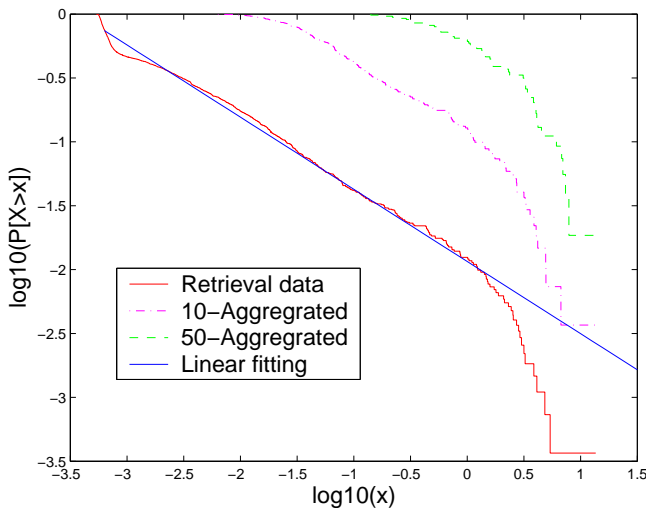


Fig. 8. LLCD plot and CLT test for the retrieval service data.

Figure 8 gives the *log-log complementary distribution* (LLCD) plot of the retrieval service data, which is the complementary cumulative distribution $\bar{F}(x) = 1 - F(x) = P[X > x]$ on log-log axes. Plotted in this way, an estimate for the

heavy-tailed distribution is obtained as the slope of the linear fitting. To verify that the dataset exhibits the infinite variance of the heavy tails, the Central Limit Theorem (CLT) test is also performed: the m -aggregated dataset from the original dataset is also drawn in the LLCD plot. As we can see from Figure 8, increasing aggregation level doesn't cause the slope to decline, reflecting the distribution has infinite variance property of heavy tail. We also employed *Hill* estimator to estimate tail weight in the data. The result shows an apparent straight line behavior for large x -values consistent with the hyperbolic tail assumption. In our experiments, the index database size is 2.1 GB and can be completely cached in memory. All experiments on the retrieval service were conducted after a warm-up process that ensures service data is completely in memory.

For the description service, trace data exhibits the exponential processing time distribution. The mean and standard deviations of the trace are close and the distribution is right-skewed (median is less than mean). Though the data does not fully satisfy the statistical patterns of exponential distributions, we can consider request time distribution of this service is close to exponential. In the experiments, the description service uses a dataset of 6.3 GB data, which cannot completely fit into memory.

Table II shows the average, 90-percentile, and maximum response time of these two services.

TABLE II
RESPONSE TIME OF THE RETRIEVAL AND THE DESCRIPTION SERVICE.

Application	Ave. (ms)	90% (ms)	Max (ms)
Retrieval	23.6	46	2,732
Description	24.6	40	596

We will mainly evaluate two types of service overloads. The first type of overload is traffic peak, for example caused by unexpected events. In this case, the system becomes overloaded because of high request rate. The second type is *size distribution shift* within the incoming traffic. This effect has been observed in live traffic at Ask Jeeves: while the client request rate remains in an expected range, the percentage of long requests increases significantly. As a result, the system becomes overloaded, which leads to service response time spikes and throughput loss.

For all applications, we use real traffic traces collected at Ask Jeeves search. We avoid measuring performance in a multi-tiered setup, because the admission control and timeout of middle tiers can shadow the real performance results. Instead, we directly replay traces collected at each server application machine. To determine the capacity of each application, we increase the request rate until there are five percent throughput losses with the vanilla Linux kernel. This probed request rate is then used as the full service capacity. During overloaded periods, all services take advantage of the admission control mechanism of the Neptune [37] middleware which uses request queue length to shed load.

B. Overhead of SRQ

We demonstrate that the SRQ scheduler has negligible performance impact on applications. To determine SRQ's overhead, we use the CPU timestamp counter on Intel chips to measure the time for system calls introduced by the SRQ scheduler, i.e., the starting and ending of a request illustrated in Figure 7. We also measure the overhead for SRQ scheduler's operations, i.e., the scheduling time, and queue splitting and shrinking time. The results are shown in Table III. As we can observe from the table, each of these operations only takes a few microseconds. Thus, the SRQ scheduler has negligible impact on application performance as long as the processing time of requests is in the order of milliseconds or more.

TABLE III
OVERHEAD OF SYSTEM CALLS INTRODUCED BY THE SRQ AND SRQ'S SCHEDULING AND QUEUE MANAGEMENT OPERATIONS.

Ops.	Start	End	Schedule	Split	shrink
Cycles	4132	7230	1433	1872	1401
Time (μ s)	2.95	5.16	1.02	1.34	1.00

C. Performance during Size Distribution Shift

TABLE IV
RESPONSE TIME OF THE RETRIEVAL SERVICE DURING SIZE DISTRIBUTION SHIFT. THE PERCENTAGE DIFFERENCE ROW ASSUMES THE PERFORMANCE OF LINUX AS 100%.

Scheduler	Linux	Linux 100	R-EDF	R-VRD	SRQ
Ave. (s)	1.413	3.106	1.706	1.121	0.529
Diff. (%)	0	+119.8	+20.7	-20.7	-62.6

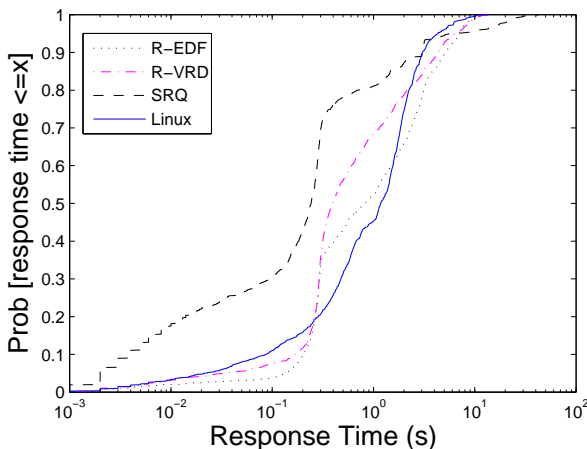


Fig. 10. Cumulative distribution of response time during size distribution shift.

In this experiment, we replayed a traffic trace from Ask Jeeves with a constant rate of 50 requests per second. We compare five different schemes: Linux with a default thread

pool size of 20; Linux using a thread pool size of 100; and three request-aware scheduler R-EDF, R-VRD, and SRQ all using the default thread pool size of 20. The throughput and response time of the retrieval service are illustrated in Figure 9. The size distribution shift happens roughly from time 30 to 155. During this time interval, there is a significant drop of system throughput due to admission control triggered by overload. The throughput of different schedulers (R-EDF, R-VRD, Linux, and SRQ) are similar to each other, because of the same admission controller used.

The second graph in Figure 9 illustrates response time changes during the experiment. SRQ performs the best, followed by R-VRD, Linux, and R-EDF. The average response times are given in Table IV. Comparing to the default Linux, SRQ and R-VRD reduce the average time by 62.6% and 20.7%, respectively. The R-EDF is 20.7% longer because the system is overloaded and the EDF algorithm degrades rapidly in such situations. The R-VRD is better than the Linux because it does request-aware scheduling and has more requests completed earlier.

In addition to request-aware scheduling, the SRQ also uses dynamic feedbacks to prioritize short requests, which is more evident from Figure 10. The figure shows that the SRQ scheduler completes short request more quickly than other schedulers. Also shown in the figure, the SRQ has longer tails, signifying that long requests are delayed. The response time improvement is mainly because a majority of requests are completed more quickly for the SRQ.

With the thread pool size of 100, the response time of Linux-100 is 119.8% higher than the default setting of 20 threads. This is because during the size distribution shift all the 100 threads are quickly occupied for processing requests, resulting in more CPU contentions. This result shows that simply allocating more threads for the applications doesn't help for size distribution shift.

D. Performance during Traffic Peaks

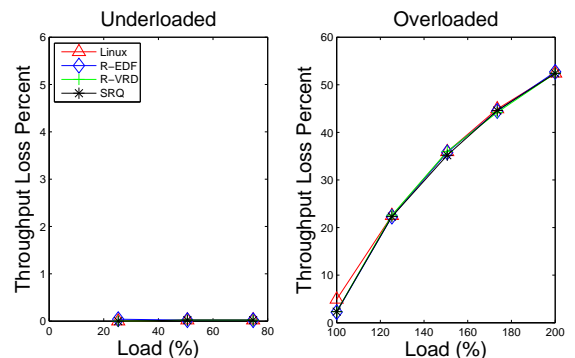


Fig. 11. Description service throughput loss.

1) *Description Service Evaluation:* For each request, the description server issues a variable number of random IO requests to disk. The Linux kernel has four different block IO schedulers: Noop, Anticipatory [22], Deadline, and CFQ.

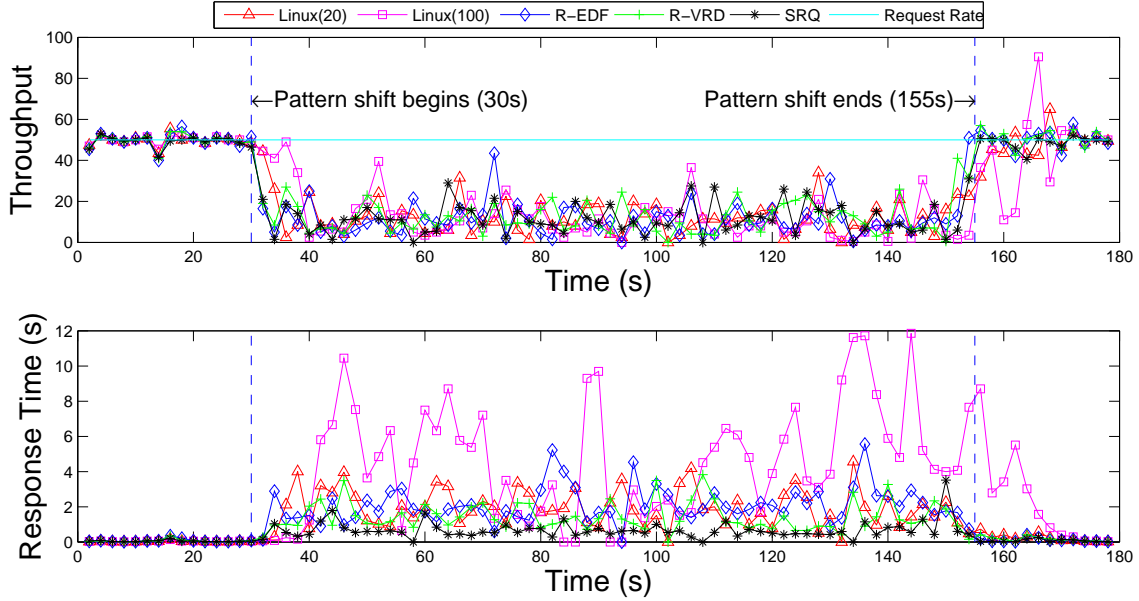


Fig. 9. Retrieval service response time and throughput during size distribution shift for five different schemes: Linux using the default 20 threads, Linux using 100 threads, R-EDF, R-VRD, and SRQ.

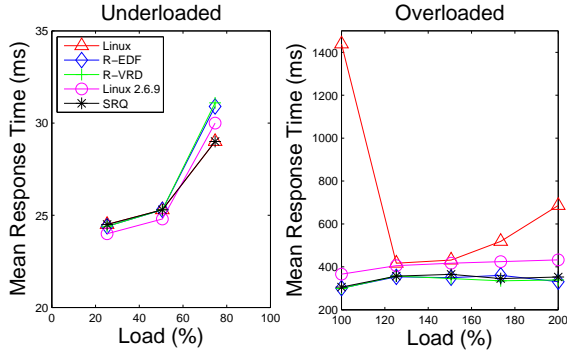


Fig. 12. Description service response time.

Using different underlying IO schedulers yields slightly different performance results. For a fair comparison, the Linux default anticipatory scheduler is used in this experiment. The results of the other three IO schedulers have similar trends and are not shown here.

Figure 11 illustrates our evaluation results of throughput loss percent in both underloaded and overloaded scenarios, which is defined as

$$LossPercent = \frac{TotalRequest - SuccessRequest}{TotalRequest} \times 100.$$

All schedulers perform mostly similar to each other because of the same admission control used. The only significant difference is when load is 100%, where R-EDF, R-VRD, and SRQ are about half the loss percent of Linux.

Figure 12 illustrates the response time for description service. The R-EDF, R-VRD, and SRQ all outperform the Linux with similar amount. For instance, the SRQ scheduler reduces

the response time by 14.5% to 78.8% compared to the Linux scheduler in overloaded situations. The similar performance is because of the workload characteristics of the description, where each request requires similar processing time. Since all three schedulers realize request-aware scheduling, the scheduling decisions are similar. Thus three schedulers have comparable performance.

In Figure 12, there is an unusual high response time of the Linux 2.6.11 scheduler when the load is 100%. Another possible factor for this performance anomaly is changes in the Linux kernel. For comparison, we also performed experiments using 2.6.9 kernel and plotted the data in Figure 12. From the figure, it's clear that 2.6.9 kernel doesn't have this anomaly and is better than the default 2.6.11 kernel we used. The performance degradation in 2.6 kernel series is also reported by other people elsewhere [9].

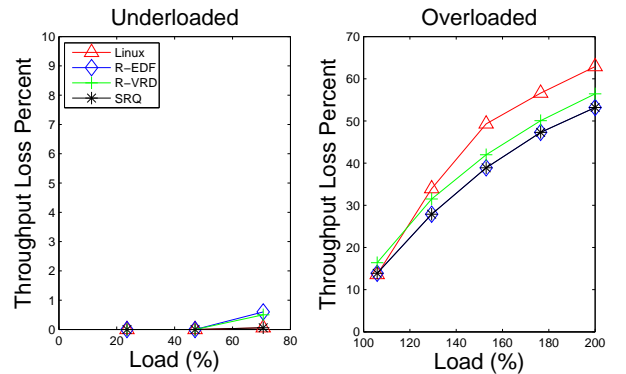


Fig. 13. Retrieval service throughput loss.

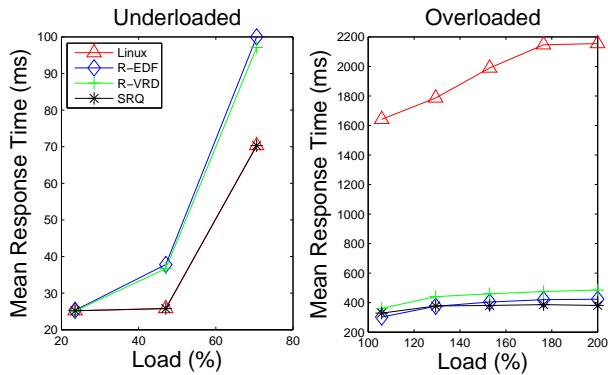


Fig. 14. Retrieval service response time.

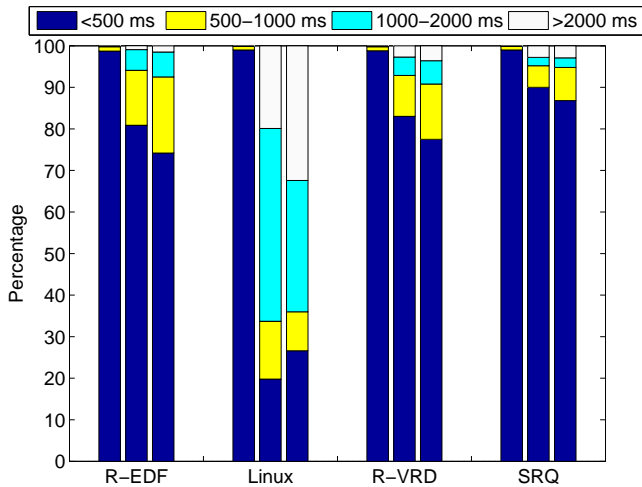


Fig. 15. Retrieval service response time breakdown when system load is 50%, 100%, and 200% (from left to right).

2) *Retrieval Service Evaluation*: Figure 13 gives the results of throughput loss of the retrieval service in both underloaded and overloaded situations. All schedulers have similar throughput loss because of the same admission control. When the system is underloaded and has enough resource, all schedulers have very little throughput loss. Linux performs the worst for overloaded cases.

Figure 14 illustrates the average response time for the retrieval service. As shown in the Figure, the Linux scheduler has much higher response time than the other three approaches in high load scenarios. For example, the SRQ reduces the mean response time by 78.9% to 82.3% compared to the default Linux during overload.

Figure 15 shows that Linux scheduler has much lower percentage of short response time requests, which explains the high response time when system load is high. The main reason for the poor performance of the Linux scheduler is that it dispatches CPU at thread level. During an overloaded period, a long request can delay the execution of multiple short requests. On the other hand, because of the request-aware scheduling mechanism, all other schedulers are able to make

new scheduling decisions after a request is serviced.

VI. RELATED WORK

Previous studies on Internet service infrastructure have addressed load balancing issues [13], [15], [16], [27], [34], [36], [37], [44]. Our work in this paper complements these studies by focusing on scheduling issues in a greater detail to improve quality of services. Multi-threaded Internet service infrastructure software, such as Neptune [37], can easily be modified to use the proposed scheduler for request-aware scheduling. However, event-driven [32], [43] servers cannot directly take advantages of the request-aware scheduler, because requests are serviced by several different threads in its life cycle, difficult for request-aware scheduling by the kernel.

Real-time systems have used deadlines to guide scheduling such as EDF [3], [23], [28], [38]. Soft-deadline scheduling is also proposed to execute aperiodic tasks among periodic tasks. Our approach adopts the concept of deadlines in our design to improve the response times. Our work is also motivated by the previous work on size-based scheduling for static web content [11], [19] and we share the same spirit of SRPT scheduling [6] but with a focus on dynamic content. Multi-level feedback queues with exponential job size have been addressed in [24]. For heavy-tailed workload, Guo and Matta [17] studied on the case of two queues. In our work, we propose dynamic queue management of multiple queues with queue splitting and shrinking. Theoretical analysis for such a queuing system is interesting for future work.

Proportional share of CPU resource [8], [23], [31], [42] targets traditional throughput-oriented computing with fair sharing. The Scout operating system unifies different software layers into the “Paths” abstraction [30] for resource allocation, scheduling, and admission control. Resource container [5] is another abstraction for resource management separation from protection domains. The main difference of our work is that this paper focuses on request-aware scheduling algorithms for Internet services optimizing quality of service in terms of response time and throughput.

VII. CONCLUDING REMARKS

The main contribution of this work is the design and implementation of a request-aware scheduling mechanism for busy dynamic Internet services that use the multi-threaded programming model. Our design integrates size adaptiveness and deadline driven prioritization in a multiple queue scheduling framework with dynamic feedback-guided queue management. Our experiments with several applications from Ask Jeeves indicate the proposed techniques can effectively reduce response time and sustain good throughput in overloaded situations, and can outperform other scheduling methods. Currently each server makes an independent scheduling decision for a large multi-tier network service application and our future work is to study the impact of cooperative request-aware scheduling among different servers.

ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful comments on earlier drafts of this paper. This work was supported by Ask Jeeves and NSF grant CCF-0234346.

REFERENCES

- [1] Apache web server. <http://www.apache.org/>.
- [2] Ask Jeeves, Inc. <http://www.ask.com/>.
- [3] RTLinux. <http://www.fsmlabs.com/>.
- [4] T. F. Abdelzaher and N. Bhatti. Web Content Adaptation to Improve Server Overload Behavior. In *Proc. of the 8th International Conference on World Wide Web*, pages 1563–1577, 1999.
- [5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 45–58, 1999.
- [6] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness. In *SIGMETRICS*, pages 279–290, 2001.
- [7] BEA Systems. BEA WebLogic. <http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/web%logic>.
- [8] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 45–58, San Diego, CA, Oct 2000.
- [9] K. Chen. Benchmarking 2.6. <http://kerneltrap.org/node/4940/>, Mar. 2005.
- [10] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*, pages 160–169, 1996.
- [11] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *2nd USENIX Symposium on Internet Technologies and Systems*, Oct. 1999.
- [12] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *ASPLOS*, 1996.
- [13] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, Oct. 1997.
- [14] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 107–121, 1996.
- [15] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.
- [16] S. D. Gribble, M. Welsh, E. Brewer, and D. Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proc. of USENIX Annual Technical Conference*, June 1999.
- [17] L. Guo and I. Matta. Scheduling Flows with Unknown Sizes: Approximate Analysis. In *ACM SIGMETRICS, Poster Session*, pages 276–277, Marina Del Rey, CA, June 2002.
- [18] M. Harchol-Balter. Task Assignment with Unknown Duration. *Journal of the ACM*, 49(2):260–288, March 2002.
- [19] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-Based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems (TOCS)*, 21:207–233, May 2003.
- [20] J. R. Haritsa, M. J. Carey, and M. Livny. Value-Based Scheduling in Real-Time Database Systems. *VLDB Journal*, 2:117–152, 1993.
- [21] R. Iyer, V. Tewari, and K. Kant. Overload Control Mechanisms for Web Servers. In *Workshop on Performance and QoS of Next Generation Network*, Nagoya, Japan, Nov. 2000.
- [22] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, pages 117–130, Oct. 2001.
- [23] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Symposium on Operating Systems Principles*, pages 198–211, St-Malo, France, 1997.
- [24] L. Kleinrock and R. Muntz. Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared Systems. *Journal of ACM*, 19(3):464–482, July 1972.
- [25] W. LeFebvre. CNN.com: Facing a World Crisis. Invited talk at USENIX LISA'01, Dec. 2001.
- [26] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 110–123, Dec. 1992.
- [27] R. Levy, J. Nagarajao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef. Performance Management for Cluster Based Web Services. In *IFIP/IEEE International Symposium on Integrated Network Management*, Colorado Springs, CO, Mar. 2003.
- [28] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [29] Microsoft Corporation. IIS 5.0 Overview. <http://www.microsoft.com/windows2000/techinfo/howitworks/iis/iis5techoverview.asp>.
- [30] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 153–167, 1996.
- [31] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In *Proc. of USENIX Annual Technical Conference*, pages 245–259, Boston, MA, June 2001.
- [32] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.
- [33] I. Ripoll, A. Crespo, and A. Garcia-Fornes. An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic-Priority Preemptive Systems. *IEEE Transactions on Software Engineering*, 23(6), Jun. 1996.
- [34] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service. In *Proc. of the 17th SOSP*, pages 1–15, 1999.
- [35] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 225–238, Boston, MA, 2002.
- [36] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, FL, 2002.
- [37] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replica Management and Programming Support for Cluster-based Network Services. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01)*, pages 197–208, San Francisco, CA, 2001.
- [38] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Systems. *IEEE Software*, 8(3):62–72, May 1991.
- [39] B. Urgaonkar and P. Shenoy. Cataclysm: Policing Extreme Overloads in Internet Applications. In *Proc. of the International World Wide Web Conference (WWW'05)*, Chiba, Japan, Jun. 2005.
- [40] T. Voigt, R. Tewari, and D. Freimuth. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *USENIX Annual Technical Conference*, pages 189–202, 2001.
- [41] J. R. von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A Framework for Network Services. In *USENIX Annual Technical Conf.*, 2002.
- [42] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, 1994.
- [43] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th USENIX Conference on Internet Technologies and Systems (USITS'03)*, pages 26–28, Seattle, WA, March 2003.
- [44] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.