Selective Early Request Termination for Busy Internet Services

Jingyu Zhou
Ask.com and UCSB
jzhou@cs.ucsb.edu

Tao Yang
Ask.com and UCSB
tyang@cs.ucsb.edu

ABSTRACT

Internet traffic is bursty and network servers are often overloaded with surprising events or abnormal client request patterns. This paper studies a load shedding mechanism called selective early request termination (SERT) for network services that use threads to handle multiple incoming requests continuously and concurrently. Our investigation with applications from Ask.com shows that during overloaded situations, a relatively small percentage of long requests that require excessive computing resource can dramatically affect other short requests and reduce the overall system throughput. By actively detecting and aborting overdue long requests, services can perform significantly better to achieve QoS objectives compared to a purely admission based approach. We have proposed a termination scheme that monitors running time of requests, accounts for their resource usage, adaptively adjusts the selection threshold, and performs a safe termination for a class of requests. This paper presents the design and implementation of this scheme and describes experimental results to validate the proposed approach.

Categories and Subject Descriptors

H.3.5 [**Information Systems**]: Information Storage and Retrival—*Online Information Services*

General Terms

Performance, Design, Experimentation

Keywords

Internet services, Load Shedding, Request Termination

1. INTRODUCTION

Busy Internet service providers often use a service-level agreement in terms of response time and throughput to guide performance optimization and guarantees. 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 flash crowds under an unexpected high request rate [7, 18]. Sometimes an abnormal change in the characteristics of request traffic may also cause service overloading. For example, an unexpected increase in the percentage of long requests or heavy requests that require a large amount of processing time can cause the slowness of the overall system performance. This is because long requests can take over most of the system resource even when the request rate is relatively low.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

Previous work has been using admission control [16, 9, 31, 34, 36, 6] and adaptive service degradation [3, 10, 36] 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 performance feedbacks [6, 36], bounding the incoming request queue length [16, 31], and policing TCP SYN packets [34].

In this paper, we explore a new load shedding mechanism that monitors request resource usage and terminates overdue long requests after they have been admitted to the system. Since a web site typically has an expectation to deliver results within a soft deadline, long requests passing the deadline often have minimal value for end users. In addition, an excessive accumulation of long requests in a server can significantly reduce the success chance of other short requests completed within a deadline. By terminating selected overdue long requests, the system can accommodate more short requests during load peaks and thus increase the throughput. Certainly not every request can be terminated safely. In this paper we are targeting a class of requests in which the termination of an individual request does not affect other requests. Examples of such applications include read-only index matching service in search engines. Our scheme called SERT monitors running times and resource usage of requests, adaptively adjusts the termination threshold, and aborts selected long requests safely.

We have implemented our scheme in the Neptune clustering middleware for network services [31, 29]. An application can link our library with little or no code modifications in deploying selective early termination of requests. The rest of the paper is organized as follows. Section 2 discusses the background on multi-threaded Internet services and motivates this work with a micro-benchmark. Section 3 gives the design considerations, architecture, and API of SERT. Section 4 describes our implementation. Section 5 evaluates SERT with application benchmarks from Ask.com. Section 6 summarizes related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

2.1 Cluster-based Internet Services and Concurrency Control

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

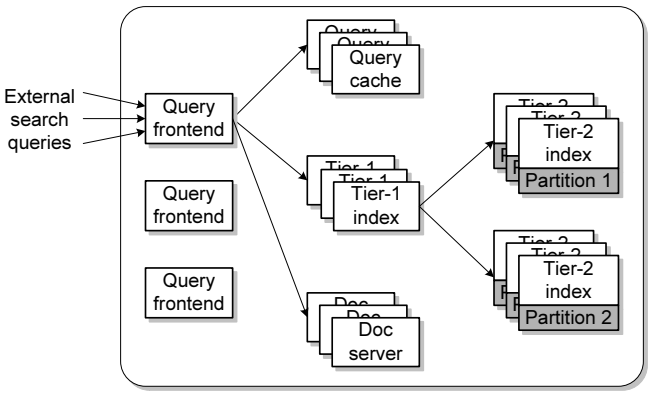

Figure 1: A three-tier keyword-based document search service.

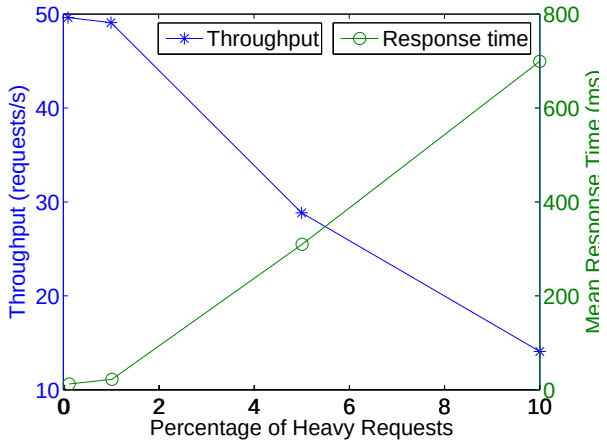

Figure 2: The impact of distribution increase in long requests on system throughput and mean response time for a CPU spinning micro-benchmark.

availability management [11, 31, 35, 24]. 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 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.

As discussed above, each machine in a service cluster handles requests concurrently sent from another machine in the cluster or from clients through Internet. In this paper, we focus on the execution of each request using a thread. Multi-threaded programming is widely used for concurrency control in Internet services such as Apache [1] and IIS [23] Web servers, BEA WebLogic application server [5], and Neptune clustering middleware [31, 29]. 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.

For an interactive Internet service, the design of service software often imposes an implicit or explicit soft deadline on its response time because a user normally expects a fast turnaround time. The performance of a service system is often measured by the number of requests that can be processed per second while each request is expected to meet a soft deadline.

2.2 Impact of Long Requests and Motivation for Early Request Termination

In a complicated web service, most requests are expected to complete by a soft deadline. However, a few requests may not be able to meet the soft deadline. For example in Ask.com search site, some uncached requests may take over one second but such a query will be answered quickly next time from a result cache. These long requests are often kept running because the number of such requests is small, and derived results can be cached for future use. In certain cases, current users may also still benefit despite the slow response.

However, when the percentage of such long requests exceeds a certain threshold, the system behavior can become extremely abnormal because long requests take an excessive amount of resource and affect short requests. The following experiment demonstrates the above phenomena. In this CPU spinning micro-benchmark, the average request arrival rate is at 50 per second, and there are two types of requests: one has the CPU cost of about 5 milliseconds while the other type costs 500 milliseconds. Initially the percentage of long requests (500 millisecond request) is around 0.1%. We gradually increase the percentage from 0.1% to 10%.

We run a server that handles these two types of requests with admission control. Each server maintains a waiting queue, and new requests will not be admitted if the queue is full (exceeding a threshold of 15 in this case). Figure 2 depicts the system throughput and response time for this server as the percentage of long requests increases. When the percentage is less than one, the system can handle incoming requests well with a throughput about 50 requests per second. With some increase in long request distribution, the system throughput falls rapidly. For example, the throughput drops to 29 from 50 when there are 5% of long requests and the response time increases from about 15 milliseconds to 309 milliseconds.

This experiment shows that although the majority of requests are short, a small portion of long requests can still dominate the overall system performance. Admission control without request differentiation on their resource usage is ineffective for load shedding in such cases.

It would be ideal that the admission controller can detect overdue long requests and reject them when load is heavy. However, it is hard to predict if a request takes a long time or not. For a web request that access static HTTP documents, previous job scheduling work [8, 15, 28] uses the file size as an estimator, but the execution time for accessing dynamic content is unpredictable in general. A viable way is to monitor the running time of each request and then terminate those detected overdue queries.

3. DESIGN OF SELECTIVE EARLY REQUEST TERMINATION (SERT)

Our strategy is to let a request run for a while so that long requests can be distinguished from short requests. Once a long request exceeds its deadline, the request can be dropped if it meets the selection criteria. There are several challenges to be addressed in this approach:

- **Execution timing and termination.** We need to monitor the elapsed execution time for each request and send a termination instruction. For the thread-based concurrent model using a thread to execute a sequence of requests, the SERT mechanism should be able to set a timer for each request. After passing a given deadline, the mechanism can perform termination of a request by rolling back program's control flow so that a new request can be processed.
- **Adaptive selection of timeout threshold.** Termination threshold selection should be based on system load situation. When the load is light, we should let each request be executed as long as an application permits. Otherwise if feasible, the timeout threshold should be adjusted adaptively following the load index.
- **Resource accounting for safe termination.** To ensure the correctness of applications, the mechanism needs to properly account for various resources (e.g., memories, locks, and sockets) used in serving a request so that these resources can be deallocated when terminating a request. Otherwise, termination without complete resource release could cause errors such as resource leaks and deadlocks.
- **Simplicity in programming**. The mechanism should require minimal or no changes to applications and should ensure the correctness of the application execution.

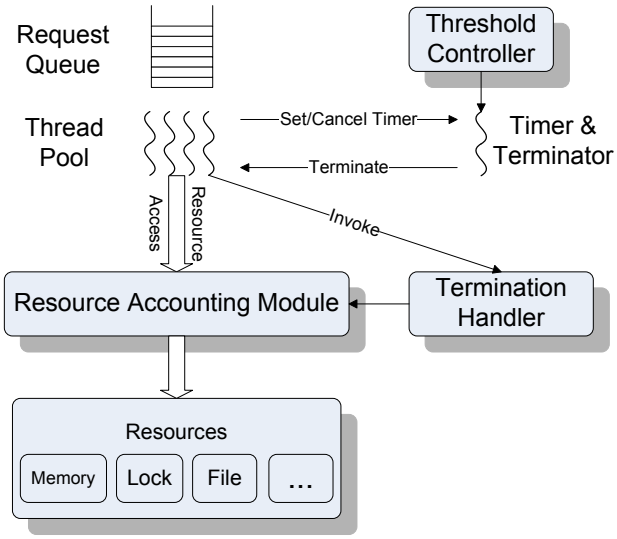

Figure 3: Architecture.

Not every request or every application can use SERT. We target classes of requests or applications where termination of one request does not affect the application semantics of another request. For example, when a request is stateless, performing read-only information retrieval, such a request can be terminated without affecting other requests. The stateless property ensures that there are no shared states among different requests, thus terminating one request will not affect the correctness of another request. The read-only property simplifies early termination because disk states remain intact.

The architecture for early termination contains the following components as illustrated in Figure 3.

- **Request Timer and Terminator.** A dedicated thread is responsible for monitoring the elapsed time of each request and sending a termination signal to the request execution thread once such a request passes its deadline.
- **Threshold controller.** This component monitors load condition and selects the timeout threshold periodically and adaptively.
- **Termination handler.** This component receives a termination signal and checks if it is safe to proceed to termination. It also releases the resource allocated for a request to be terminated.
- **Resource accounting.** This module records all resources acquired by each request. This information will be used by the termination handler to release all allocated resources before aborting a request safely.

The rest of this section describes our design in details.

3.1 Execution Flow of Requests with SERT

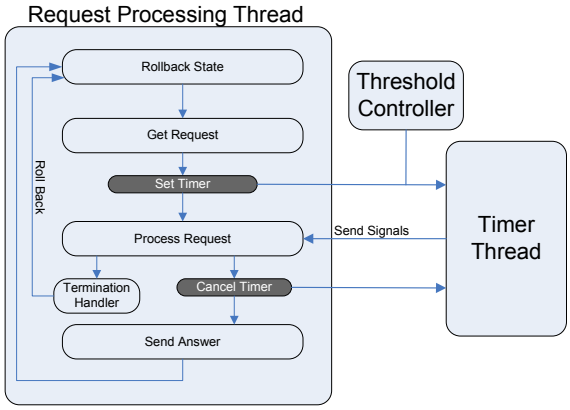

Figure 4: Request execution flow with SERT.

The early termination requires the execution of a request to be timed. It is possible to use kernel timers to notify a request processing thread of passing deadline events with upcalls or scheduler activations [4]. However, this approach would require many kernel modifications as well as application changes. To achieve minimum changes to applications and better portability, we choose a user-level design using signals.

Figure 4 illustrates the flow of request execution incorporated with SERT. Each thread that handles an incoming request goes through the following steps. First, a rollback point is set so that when a request is terminated, the execution can return back this safe point. After a new request is received, the request processing thread notifies the *Timer* thread to start timing this request processing operation.

This dedicated Timer thread receives a startup message from the request processing threads to monitor the elapsed time spent for each request. The threshold controller dynamically selects the termination time for each request based on the system load. If a request exceeds the selected processing time limit, the Timer thread

will send a termination message to the corresponding request processing thread.

The Timer thread can also receive a cancellation instruction to stop timing certain requests. That is designed to avoid unsafe termination of some classes of requests. For example, when a request writes to a connection socket and then a termination is performed for the request, the calling tier that reads the socket only receives partial results and could generate faults. The solution is to monitor write events on connection sockets and to cancel the termination after the write event.

We also provide temporary termination masking capability. Namely a termination is temporarily delayed until a certain function call completes. For example, there are classes of system calls in which a signal can interrupt and cause the system call to fail. Thus we mask the termination signal during the system call. Some GLIBC functions internally use locks for serial execution, for instance `printf()` function. If a signal is delivered after locks are acquired and the program jumps back to the rollback point, deadlocks will occur for future calls. We wrap these special functions and mask the termination signal during their executions.

There is a potential race condition caused by a late delivered signal, i.e., a signal to abort a request is delivered after the request processing has finished. To avoid such races, we associate a per thread sequence number with each request. This sequence number is carried with the signal so that it can be checked by the signal handler. If the sequence number matches the current request, rollbacks are allowed. Otherwise, the signal is delivered late and rollbacks are denied.

3.2 Threshold Controller

Applications can specify a termination threshold or a range of timeout values for each class of requests. When a range is specified, the threshold controller can assign a timeout value adaptively within the given range that maximizes the system throughput. The upper bound value of this range indicates that the request result is not acceptable for the application if such a limit is reached. The lower bound value of this range indicates a minimal interval that should be allowed for a request to complete.

The basic idea behind our adaptive controlling strategy is that when the system load is excessively high, the controller should use a smaller threshold to minimize the adverse effect of long requests. Otherwise the controller sets a larger timeout threshold value so that long requests can complete without being terminated. For example, in web search applications, completion of such a long request may not be valuable for a current client, but can be useful for future requests with the same query since results can be cached.

To judge if the load of the system exceeds its capacity, we use the throughput loss as an indicator, which is defined as

$$p = \frac{\text{The number of requests dropped/terminated during last interval}}{\text{The total number of requests arrived during last interval.}}.$$

The controller monitors such a number with a fixed interval.

When the throughput loss exceeds a high watermark (HW), we consider the system is extremely overloaded, and the lower bound value for termination threshold should be used to terminate more requests and to create room to accommodate as many new requests as possible. When the throughout loss p is below a low watermark (LW), we consider the system load is still acceptable, and an upper bound may still be used. When the loss is between LW and HW, we compute the selection threshold as a nonlinear function of p with a scaling factor α as described below.

Let the range of timeout threshold be $[LB, UB]$, where LB and UB represent the lower and upper bounds respectively. Our formula for computing and readjusting a timeout value at each monitoring time interval is expressed as follows.

$$Threshold = LB + F(p) \times (UB - LB),$$

where

$$F(p) = \begin{cases} 1 & p <= LW \\ (\frac{HW-p}{HW-LW})^{\alpha} & LW < p < HW \\ 0 & p >= HW \end{cases}$$

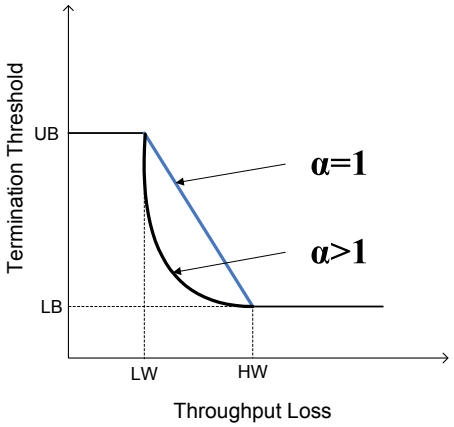

Figure 5: Illustration of threshold function.

As shown in Figure 5, the α value can control the tradeoff between the threshold lower bound and upper bound when the load index changes. When α is zero, the upper bound is used, suitable for a lightly loaded condition. When α is very large, a value close to the lower bound is used, suitable for a heavy loaded condition. In addition to α value, other parameters can be adjusted and different applications may use a different parameter setting. For our tested applications, we use $\alpha = 4$, $HW = 15\%$, and $LW = 5\%$ with a 10 seconds monitoring interval. In Section 5, we will present experiment results on the choices of these parameters.

3.3 Resource Accounting

After the deadline of a request is passed, the program changes its control flow to a pre-defined rollback point. To avoid resource leaks and other errors, all resources (e.g., memories, sockets, file descriptors, and locks) allocated while servicing the request need to be deallocated. This requires proper accounting of the resources and we discuss resources accounted in SERT as follows.

- **Memory.** The memory resource includes dynamic allocated memory from heaps using library functions such as `malloc()` and `new()`, memory mapped regions returned by `mmap()` calls, and local variables allocated on the stack. We track memory usage of heaps and memory mapped areas. Memories allocated on the stack are reclaimed after stack unwinding and not accounted.

- **Locks.** Locks are another type of resource that must be accounted to avoid deadlocks that can happen if the execution rolls back right after acquiring a shared lock. Since the lock is not freed, all subsequent attempts to acquire the lock would block the calling thread, causing deadlocks. We solve this problem by delaying the termination until it is safe to do so. Since a thread may hold multiple locks, we use a integer counter to track the number of locks held by a thread and only

allow the termination when the counter is zero. The counter is increased before a lock is acquired and is decreased after releasing the lock. If a signal is delivered inside the critical section, the signal handler simply marks a flag associated with the thread. After releasing the lock, this flag is checked to see if a termination is needed.

- **Sockets and file descriptors.** If not accounted, socket or file descriptor leaks can cause further allocation error, because each process has an upper limit for them. Proper accounting of these resources is required since our service applications should be able to run for months without interruptions.

The SERT library tracks all resources allocated by a thread for a request, thus the resources can be deallocated before the control flow returns to the rollback point.

For some applications that involve resource sharing among request processing threads, some minor change in applications can be made to take advantages of SERT. For example, if there is an application module that handles memory objects among requests, and this module guarantees the memory is allocated and deallocated properly independent of the success of request execution. Then we do not need to account for resources allocated in such a module and SERT can be used safely for such applications.

3.4 SERT API

We provide SERT library functions to be used for request execution control. If a middleware such as Neptune [31] controls the high level execution flow of a request, then the code change will happen in the middleware level as we will illustrate later in Figure 7. No code change is necessary in the application level. If an application directly writes the thread control code for executing a request, then this application needs to be modified slightly to insert the proper SERT control functions.

Figure 6 illustrates the API functions of the SERT library that need to be inserted to applications. `SERT_start()` and `SERT_end()` define the beginning and the end of a request processing respectively. A termination threshold value or range is made explicit to the SERT library with `SERT_set_args()` call, which specifies the deadline range and other control parameters of a request. Note that applications may want to set different criteria for different types of requests for differentiated quality control. `SERT_init_timer()` starts the timer thread and registers the signal type that is used for notifying timeout event. `SERT_register_rollbackpoint()` registers the return point for execution rollback.

```
/* start timer thread and set signal type */
extern int SERT_init_timer(int signum);

/* start & end of a request */
extern void SERT_start();
extern void SERT_end();

/* set timeout value and controller parameters */
extern void SERT_set_args(struct sert_arg *);

/* set the rollback point */
extern void SERT_register_rollbackpoint(void *);
```

Figure 6: API functions of the SERT library.

To allow targeted applications to run without code change, SERT intercepts GLIBC/Pthread functions to add needed control for resource accounting. The library defines a set of GLIBC functions related to memory management (e.g., `malloc()` and `free()`), Pthread functions (e.g., `pthread_mutex_lock()`), and file operations (e.g., `open()` and `close()`). When linked with applications, these functions will first perform internal logging of resource usage and then invoke corresponding GLIBC or Pthread routines.

4. IMPLEMENTATION

We have implemented the SERT library and have linked it with the Neptune clustering middleware [31, 29], which has been used in Ask.com and internal servers supporting many different Internet and data mining applications running on giant-scale clusters. The dynamic memory management code is a modified version of TC-Malloc [12].

To implement the termination through execution rollback, we need to let control flow jump to a saved stack context (rollback point), we use C library `setjmp(3)` and `longjmp(3)` pair instead of C++ exceptions. Though the exception handling feature of the C++ language can simplify some of our work (deallocate resources by destruction of local objects), it is not suitable for us to define a special exception to achieve stack unwinding. This is because the programmer can specify a `catch(...)` block to capture all exceptions, thus preventing further stack unwindings. Because our use of jumps is inside a signal handler, the actual implementation uses `sigsetjmp(3)` and `siglongjmp(3)` variants to ensure that correct signal masks are saved and restored.

Figure 7 gives a pseudo-code example of using the SERT library. The Neptune middleware [31] is easily modified by adding about ten lines of code using the SERT API. Note that applications running on top of Neptune require no changes.

```
void worker()
{
    while (1) {
        Request *request = get_request();
        jump_buf env;

        if (sigsetjmp(&env, 1) == 0) {
            SERT_register_rollbackpoint(&env);
        } else {
            /* longjmp back, resources has already
               been deallocated */
            continue;
        }

        SERT_start();
        process_request(request);
        SERT_end();

        send_result(request);
    }
}
```

Figure 7: A pseudo-code example of using SERT.

5. EVALUATION

In this section, we will compare the SERT approach with an admission control based load shedding method. This admission control method is called AC in this section which uses the queue length of incoming requests to shed the load. Note that SERT uses both selective request termination and AC method. We will also assess the effectiveness of adaptive termination threshold selection. Our experiments show that SERT has negligible performance impact on applications and are not reported here.

We will use the following two types of traces:

- **Size distribution shift.** In this type of trace, the total number of queries arrived per second remains constant. The distribution of long queries changes during some period. This trace

evaluates the algorithm's capability in dealing with the shifting of query size patterns.

- **Traffic peaks**. In this type of trace, the total number of queries increases gradually while the distribution of long queries remains the same. This trace evaluates the algorithm's capability in dealing with traffic with different load conditions.

The performance metrics we use are response time and throughput. In measuring the sustained throughput compared to the arrival rate, we use the loss percentage which is defined as

$$LossPercent = \frac{TotalRequests - SuccessfulRequests}{TotalRequests} \times 100.$$

A loss percentage of zero means that all arrived requests are handled successfully. In terms of response time, we measure the average response time for successfully executed requests within the latest deadline required.

5.1 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.

The applications used for the evaluation are two services from Ask.com [2] with different size distribution characteristics: a database *index matching* service and a page *ranking* service. The index matching service that finds all web pages containing certain keywords is heavy-tailed. The ranking service, that ranks web documents based on relevancy of such documents for a given query, is exponential. Our experiment in Section 5.2 shows that SERT is especially effective for heavy-tailed workload.

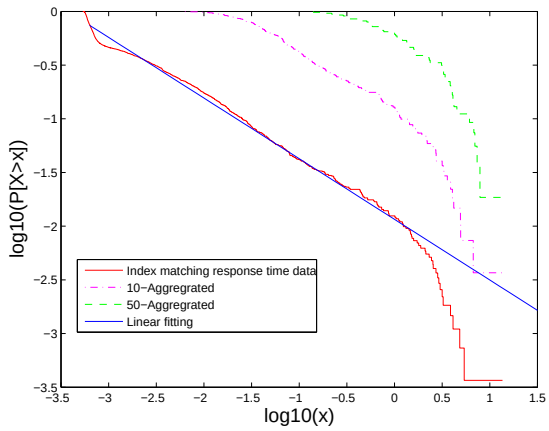

Figure 8: LLCD plot and CLT test for the index matching service response time data.

Figure 8 gives the *log-log complementary distribution* (LLCD) plot of response time data for the index matching service, 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 the 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 completed cached in memory. All experiments on the index matching service were conducted after a warm-up process that ensures service data is completely in memory.

For the ranking 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 (the median is less than the 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.

Table 1 shows the average, 90-percentile, and maximum response time of these two services.

Application	Ave. (ms)	90% (ms)	Max (ms)
Index Matching	23.6	46	2,732
Ranking	93	212	14,035

Table 1: Characteristics of response time in index matching and ranking services.

Table 2 lists the parameter values used in the threshold controller of SERT for these two services.

Param.	**Description**	**Rank**	**Index**
$interval$	Controller monitoring period	10 s	10 s
LB	Lower bound for timeout value	0.5 s	1.5 s
UB	Upper bound for timeout value	15 s	8 s
α	Scaling factor	4	4
HW	High watermark for loss	15%	15%
LW	Low watermark for loss	5%	5%

Table 2: Parameters used in the threshold controller for ranking and index matching services.

For all applications, we use real traffic traces collected at Ask.com. To determine the capacity of an application server, we increase the request arrival rate until there are five percent throughput losses when the standard Neptune server with admission control is used. 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 [31] middleware which uses request queue length to shed load.

5.2 Performance during Size Distribution Shift

In this experiment, we replayed a traffic trace from Ask.com which has an average arrival rate of 50 requests per second. There is a size distribution shift in this trace such that the distribution of heavy requests increases at a certain period. While the traffic pattern is normally heavy-tailed with less than 0.5% of heavy requests (those require more than 500 ms processing time), this trace contains about 10% of heavy requests during the shift period from time 30 to 155.

The throughput and response time results of the index matching service are illustrated in Figure 9 for responding to this trace. There is a significant throughput drop using the AC method during the distribution shift. The SERT's throughput is 209.1% higher than AC, because heavy requests were selectively dropped and more short requests were completed.

The bottom graph in Figure 9 illustrates response time changes during the experiment. SERT performs significantly better than AC. The average response time is 0.640 second for the SERT. Compared to the 1.413 second for AC, this is a 54.7% reduction.

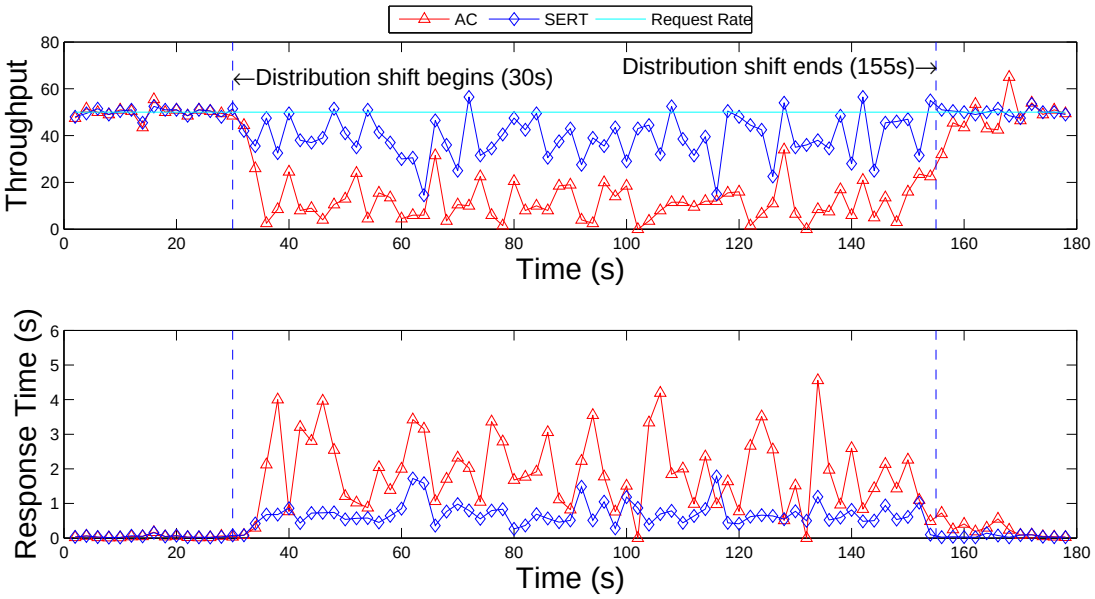

Figure 9: Average response time and throughput of index matching service during size distribution shift.

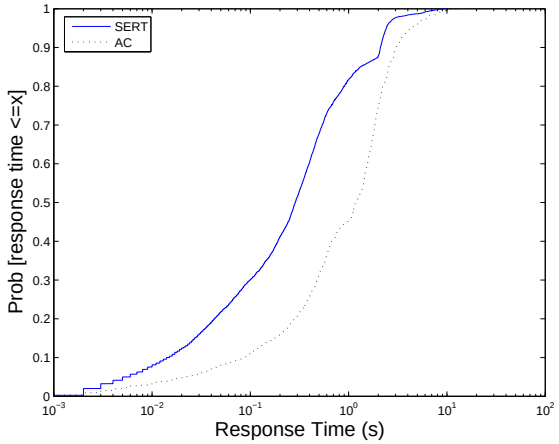

Figure 10: Cumulative distribution of index matching response time during size distribution shift.

Figure 10 gives the cumulative distribution of response time for this experiment. Value (x,y) means that the probability of computing within response time x is y. SERT has better response times in all cases. For example, the percentage of requests completed within one second is 81.7% for SERT and 45.3% for AC.

5.3 Performance during Traffi c Peaks

5.3.1 *A comparison of SERT and AC in ranking service*

This experiment examines and compares the performance of SERT and AC when the traffic load gradually increases. Figure 11 illustrates our evaluation results of throughput loss percentage for the ranking service in both underloaded and overloaded scenarios. Both schemes perform mostly similar to each other in underloaded situations. During the overloaded cases, SERT performs better and reduces the loss percentage by 7.5% to 27.9% compared to the AC scheme.

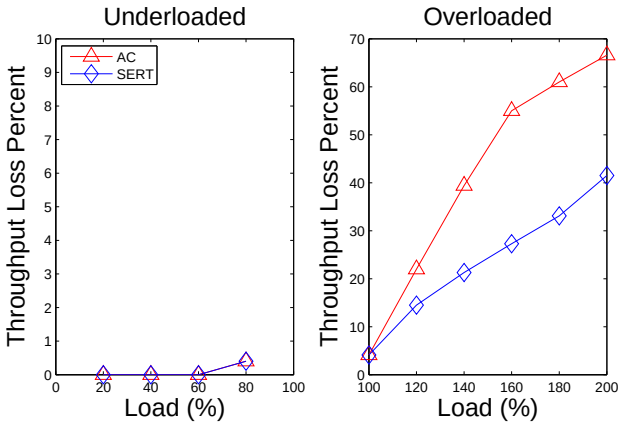

Figure 11: Throughput loss in ranking service with AC and with SERT under different load conditions.

Figure 12 illustrates the response time for the ranking service. Again, both approaches have comparable results in underloaded scenarios. Compared to AC, the overloaded response time of SERT is from 38.0% to 75.4% lower. In SERT, the response time drops to a certain degree from 120% to 140% load. This is because the threshold controller gradually reduces the termination threshold, which leads to more dropping of heavy requests.

In summary, both approaches have comparable performance in underloaded situations, while SERT outperforms AC during traffic peaks. The main reason is that SERT drops heavy requests and allows the system to serve more short requests, augmenting both throughput and response time.

5.3.2 *A comparison of SERT and AC in index matching service*

Figure 13 gives the results of throughput loss of the index matching service in both underloaded and overloaded situations. Both schemes have very little throughput loss when the system is underloaded and has sufficient resources. During the overloaded cases, the loss percentage of SERT is 2.7% to 13.8% lower than AC.

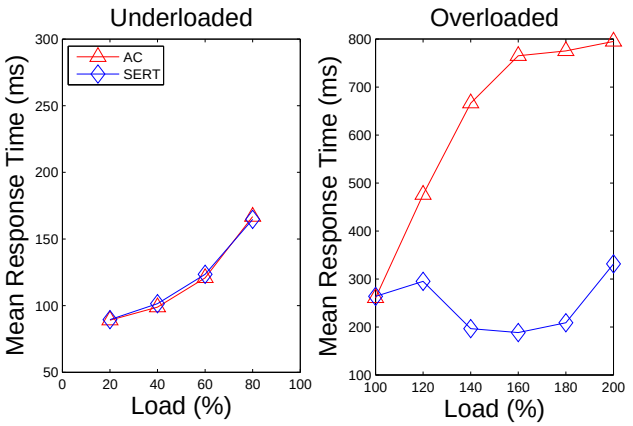

Figure 12: Response time of ranking service with AC and with SERT.

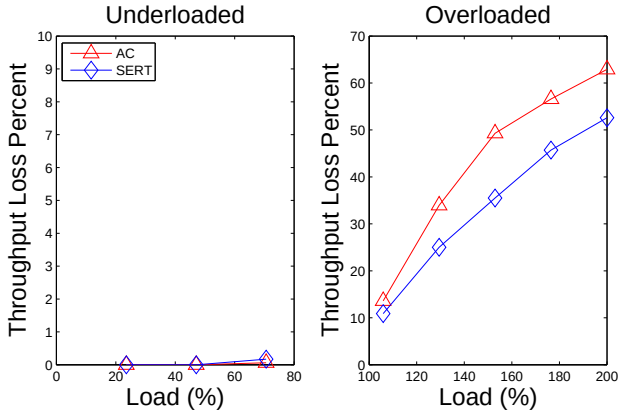

Figure 13: Throughput loss in index matching service with AC and with SERT under different load conditions.

Figure 14 shows response time for the index matching service. For underloaded cases, two schemes have comparable performance. For overloaded cases, AC has much higher response time than SERT because heavy requests adversely affect short requests in AC.

5.4 An Evaluation of Threshold Controller

5.4.1 *Effectiveness of adaptive threshold selection*

We evaluate the effectiveness of adaptive termination threshold selection by comparing with a fixed threshold policy. Figure 15 illustrates a comparison of the adaptive selection policy with three fixed threshold polices in terms of throughput and response time in the ranking service. The fixed termination thresholds used are: 0.5 seconds, 3 seconds, and 15 seconds. From the figure, we can see that a fixed policy with a higher threshold value performs better when the load is 100% loaded or less. For example, when system is 100% loaded, the 0.5 second threshold has 11.2% throughput loss, while the 15 second policy and the adaptive selection have 4% loss. When system load increases beyond 140%, a lower threshold has better throughput by rejecting heavy requests as earlier as possible.

The threshold controller adapts to load conditions and performs favorably under different load conditions. For 120% load or less, the controller uses a high threshold and performs better than the fixed 0.5 second threshold in terms of throughput. The average response time with 0.5 second threshold is lower, because it rejects too many requests. As the load becomes excessively high, the controller uses a lower termination threshold with a competitive performance as using fixed 0.5 second threshold, and outperforms other fixed policies.

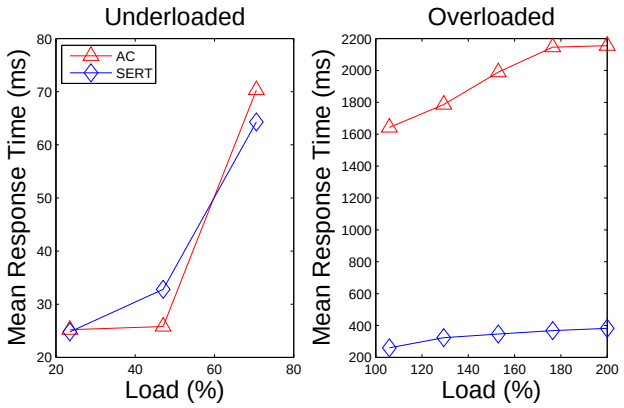

Figure 14: Response time for index matching with AC and with SERT.

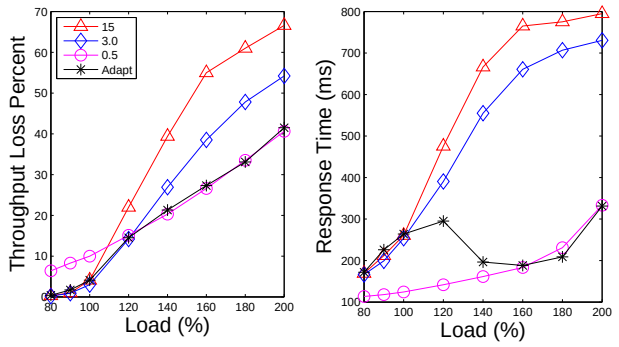

Figure 15: A comparison of ranking service using fixed termination thresholds (0.5s, 3s, 15s) with adaptive threshold selection.

Figure 16 shows a similar comparison for the index matching service. For fixed thresholds, using the 1.5s value always performs better and the adaptive selection has a comparable performance.

5.4.2 *Impact of varying control parameters*

We vary the value of throughput loss watermarks for the threshold formula described in Section 3.2. Since low watermark is effective when the load has not reached or is near the system capacity, the evaluation is focused on the throughput loss in such situations. For high watermark, evaluation is focused on overloaded scenarios. Figure 17 and Figure 18 illustrate the comparison results for the ranking and index matching services respectively when using 1%, 3%, 5%, and 7% for low watermarks, and 10%, 15%, 20%, and 25% for high watermarks. For both services, there is actually no significant performance difference for using these tested watermark values. Thus we choose 5% for low watermark and 15% for high watermark for both services.

For α value, the experiment results in the previous subsection actually reflect the cases of $\alpha = 0$, $\alpha = 4$, and $\alpha = \infty$ while $\alpha = 4$ has a significant advantage. We have done experiments to vary α from 4 to 8 and did not find significant performance difference. When α drops from 4 to 1, α value 4 outperforms α value 1 significantly. Due to paper length limit, we do not include the experimental details on these results.

For the monitoring interval, we find that the 10 seconds interval gives a better performance than a small interval such as 2 seconds, especially for the ranking service. This is because traffic is bursty,

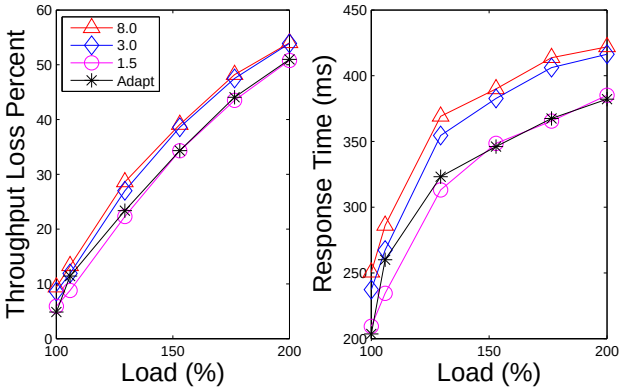

Figure 16: A comparison of index matching service using fixed termination thresholds (1.5s, 3s, 15s) with adaptive threshold selection.

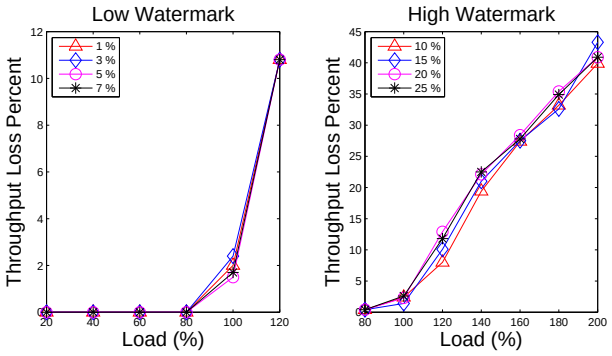

Figure 17: Varying throughput loss watermarks in ranking service.

and a small interval can yield an unstable prediction of traffic behavior. When increasing the interval from 10 seconds, there is no significant gain.

6. RELATED WORK

Admission control is an effective way to protect Internet services from being overloaded. Some of the work focuses on reducing the amount of work [6, 9, 16, 36], and others differentiate classes of requests so that response time of preferred clients do not suffer much during load peaks [29, 34, 39]. While admission control rejects a request before entering the system, our approach is complementary and allows the request to enter the system and to be rejected later based on its resource usage.

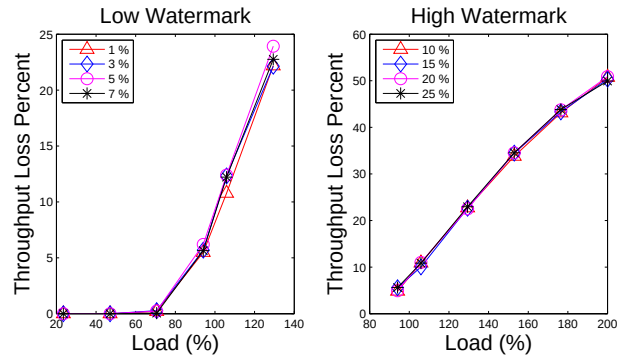

Figure 18: Varying throughput loss watermarks in index matching.

Our work is also motivated by the previous work on size-based scheduling for accessing static web content [8, 15, 28]. These studies prioritize short requests so that they are serviced first, while our approach actively detects and drops long requests. Based on the finding that different servlets of TPC-W benchmark have relatively consistent execution time, Elnikety *et al.* [9] schedule requests for known servlet types where each type represents a different resource need. This optimization is performed based on the static application knowledge.

Previous studies on Internet service infrastructure have addressed load balancing issues [11, 13, 14, 26, 30, 31, 37, 38]. Our work in this paper complements these studies by focusing on load shedding to improve quality of service.

Database concurrency control has employed the abort technique [17, 20, 32]. Particularly in real-time databases, the execution of higher priority transactions may cause lower priority transactions to be aborted. These studies have focused on the restoration of data dependence and studied REDO and UNDO logs for recovery of database transactions. In SERT, while the procedure for a general Internet request can be arbitrary and more complicated than ones using a database transaction language, we focus on a simplified dependence scenario where requests are independent, e.g., read-only and stateless requests. Recently, McWherter *et al.* [22] show that selectively aborting certain lower priority transactions can significantly improve the response time of higher priority transactions for TPC-C type workloads, where database locking dominates the response time. They have proposed lock-related abort techniques.

Recoverable virtual memory [27] and Rio Vista [21] are user-level recoverable memory libraries designed for database transactions. Both approaches require application modifications to access memory regions for safe transaction control. Our approach intercepts applications' memory allocation calls so that user code does not need to be changed for classes of applications we are targeting.

Process checkpointing and rollback [19, 33, 25] has been explored for fault tolerance, deterministic program replay and debugging. Instead of checkpointing, SERT logs resource usage after the execution of a request begins and rollbacks a program state by deallocating these resources. Additionally, such a rollback is performed on threads, not processes, so that other threads can continue serving new requests without interruptions.

7. CONCLUDING REMARKS

The main contribution of this work is the design and implementation of an early termination mechanism for busy dynamic Internet services that use the multi-threaded programming model. Our design dynamically selects termination threshold, adaptive to load condition and performs early termination safely. Our experiments with two applications from Ask.com indicate the proposed techniques can effectively reduce response time and improve throughput in overloaded situations. It should be noted that for different classes of requests, an application may deploy different termination ranges and control parameters and our API design can support such differentiation.

Techniques proposed in this paper have been applied in each single tier within a multi-tier application. Each node makes an independent decision in dealing with concurrent requests from clients of a website or from other service modules within a large cluster for other tiers. One future work is to study the impact of cooperative request-aware early termination among different nodes and tiers. Currently the threshold controller uses a number of parameters that are determined through offline profiling. A further improvement is to use online monitoring as feedbacks for adjusting these parameters.

Acknowledgments

We thank Lingkun Chu, Kai Shen, Giovanni Vigna, Rich Wolski, and the anonymous referees for their helpful comments on earlier drafts of this paper. This work was supported by IAC Search & Media (formally Ask Jeeves) and NSF grant CCF-0234346.

8. REFERENCES

[1] Apache web server. `http://www.apache.org/`.

[2] IAC Search & Media. `http://www.ask.com/`.

[3] 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.

[4] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[5] BEA Systems. BEA WebLogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/web%logic`.

[6] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *Proc. of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston MA, May 2005.

[7] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*, pages 160–169, 1996.

[8] 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.

[9] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *WWW*, pages 276–286, 2004.

[10] 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.

[11] 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.

[12] S. Ghemawat and P. Menage. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T.-W. Kuo, M.-C. Liang, and L. Shu. Abort-oriented Concurrency Control for Real-time Databases. *IEEE Transactions on Computers*, 50(7):660–673, Jul. 2001.

[18] W. LeFebvre. CNN.com: Facing a World Crisis. Invited talk at USENIX LISA'01, Dec. 2001.

[19] K. Li, J. F. Naughton, and J. S. Plank. Real-time, Concurrent Checkpoint for Parallel Programs. In *Proc. of 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, WA, 1990.

[20] Y. Lin and S. H. Son. Concurrency Control in Real-Time Database by Dynamic Adjustment of Serialization Order. In *IEEE 11th Real-Time System Symposium*, Dec. 1990.

[21] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, 1997.

[22] D. McWherter, B. Schroeder, N. Ailamaki, and M. Harchol-Balter. Improving Preemptive Prioritization via Statistical Characterization of OLTP Locking. In *Proc. of the 21st International Conference on Data Engineering (ICDE)*, San Francisco, CA, Apr. 2005.

[23] Microsoft Corporation. IIS 5.0 Overview. `http://www.microsoft.com/windows2000/techinfo/howitworks/iis/iis5techov%erview.asp`.

[24] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proc. of Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[25] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies– A Safe Method to Survive Software Failures. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Oct. 2005.

[26] 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.

[27] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[28] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technologies*, 6(1), Feb. 2006.

[29] 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.

[30] 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.

[31] 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.

[32] L. Shu and M. Young. A Mixed Locking/Abort Protocol for Hard Real-Time Systems. In *IEEE 11th Workshop Real-Time Operating System and Software*, pages 102–106, May 1994.

[33] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Annual Technical Conference*, 2004.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] H. Zhu, B. Smith, and T. Yang. Scheduling Optimization for Resource-Intensive Web Requests on Server Clusters. In *Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 13–22, 1999.

[39] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation for Cluster-based Network Servers. In *IEEE INFOCOM*, pages 679–688, 2001.