Difference of Degradation Schemes among Operating Systems — Experimental analysis for web application servers —

Hideaki Hibino Kenichi Kourai Shigeru Chiba
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
{hibino,kourai,chiba}@csg.is.titech.ac.jp
2-12-1-W8-50 Ohkayama, Meguro-ku, Tokyo 152–8552, JAPAN
Phone: +81–3–5734–3041 Fax: +81–3–5734–2754

Abstract

Graceful degradation is critical ability of highly available middleware such as a web application server but naive graceful degradation is often unsatisfactory. If a web application server provides multiple services under heavy workload, it must gracefully degrade the execution performance of each service to a different degree depending on the quality required to that service. This paper reveals that the schemes of this performance degradation, which we call a degradation scheme, *are different among several major operating systems unless middleware-level control is performed. Some operating systems showed undesirable behavior with respect to the degradation. This paper reports this fact with the results of our experiments and it also mentions that a major factor in this difference is the waiting time for acquiring a lock.*

keywords: degradation scheme, graceful degradation, operating system, web application server, multiple services

1 Introduction

Web application servers have been often used to build complex enterprise applications on the Internet, such as online shopping sites. A key performance factor of such servers is the response time of dynamic pages generated by, for example, Java servlets. Thus gracefully degrading the execution performance of servlets under heavy workload is critical ability of highly available web application servers [8].

If a web application server provides multiple services, degrading the performance of these services by servlets to the same degree is not desirable. The performance of each service must be degraded to a different degree depending on the quality required to that service. The scheme of this performance degradation of each service, which we call a *degradation scheme*, is significant to build highly available web application servers and hence it must be controlled at middleware level. Without such control, some operating systems show undesirable behavior.

In this paper, we report that several major operating systems without the middleware control showed different behavior with respect to the degradation scheme. Those operating systems include Solaris 9, Linux 2.6.7, 2.6.5 and 2.4.18, FreeBSD 5.2.1, and Windows 2003 Server. We ran heavy-weight and light-weight services implemented as servlets at the same time on the Tomcat web application server and measured the throughput as we gradually increased the workload. The result revealed that the throughput of the light-weight service was gradually degraded on Solaris whereas it was steeply degraded on other operating systems. Also, it was shown that the small difference of versions in Linux causes a large difference of degradation schemes. Although which behavior is appropriate depends on the service contexts, such difference in the degradation scheme should be absorbed by the middleware layer. Otherwise, the application developers would have to adjust their applications for each operating system.

Furthermore, we investigated what is a major factor of the difference in the degradation schemes between Solaris and Linux. According to the results of our experiments, it is the waiting time for acquiring locks, which was longer in Linux than in Solaris. This is because (1) a Linux thread issues a larger number of system calls for acquiring locks and (2) the Linux kernel uses an inappropriate policy of thread scheduling in this particular contexts.

The rest of this paper is organized as follows. Section 2 describes what the degradation scheme is. Section 3 reveals the difference of the degradation schemes among operating systems. Section 4 investigates a major factor of the difference in the degradation schemes between Solaris and Linux.

Section 5 presents the future direction of this research. Section 6 discusses related work and Section 7 concludes the paper.

2 Degradation Scheme

Under heavy workload, a web application server must gracefully degrade its execution performance. However, degrading the performance of all the services to the same degree might be an inappropriate scheme.

Let us consider an online shopping site. Such a site provides various services ranging from light-weight services, such as simply listing product information, to heavy-weight services, such as transactions for purchasing products. Each service is small software implemented as a Java servlet, for example. If fair round-robin scheduling is executed, the performance of all the services will be degraded to the same degree under heavy work load. However, this naive degradation might reduce the customer satisfaction. They might want to keep short response time for the light-weight services since they frequently request those services for listing product information. On the other hand, they might not mind that the heavy-weight services like transactions for purchasing products slow down for processing more requests for light-weight services since they run such a heavy-weight service only at the end of their session.

Selecting an appropriate *degradation scheme* is crucial to maximize the customer satisfaction. The degradation scheme is a scheme of how much the performance of each service should be degraded under heavy workload to satisfy the quality required to that service. If a burst of requests are sent to multiple services at the same time, each service can process only a smaller number of requests per time than in the case where only that service is requested. We propose how much smaller number of requests is processed per time should differ among the services. Since all the services share limited system resources including CPU time, the performance of each service is degraded respectively when the workload is heavy and thus the services compete for system resources with each other. Therefore, the degradation scheme is a scheme of how much amount of system resources are allocated to each application-level service.

3 Difference among Operating Systems

The degradation scheme should be controlled by middleware to fit applications. Without such middleware-level control, operating systems would use an inappropriate scheme. In fact, according to our experiment, operating systems widely used today show different behavior with respect to the degradation scheme if any middleware-level control is not performed. The behavior of some operating systems is inappropriate for our scenario mentioned in Section 2. This behavioral difference among operating systems means that the behavior of a web application under heavy workload significantly depends on the underlying operating system.

This section shows the results of our experiment to illustrate differences in the degradation scheme among operating systems.

3.1 Experimental Setup

In this experiment, we used the Tomcat web application server [7], which ran on top of the Java VM (JVM). Tomcat is usually used in 3-tiered web sites, which consist of web servers, application servers, and database servers. Since many network I/O for clients are done in web servers and file I/O is done in the database, services provided by application servers are often CPU- and memory-intensive. For simplicity, we prepared three types of CPU- and memory-intensive services: heavy-weight, light-weight, and middle-weight services. The heavy-weight service creates a Document Object Model (DOM) tree from an XML file and repeats the search of all the nodes in the tree 100 times. This heavy-weight service uses a large amount of memory resource and CPU resource and causes garbage collection in JVM because it creates many short-lived objects in Java. On the other hand, the light-weight service performs the calculation of the 25th Fibonacci number and uses only a little amount of CPU resource. The middle-weight service performs the calculation of the 35th Fibonacci number and uses a relatively large amount of CPU resource.

To reveal the performance degradation scheme, we let the clients generate various workloads to Tomcat. For one workload set, we fixed the number of requests to the light-weight service to 30 and increased that to the heavy-weight service from 0 to 40 (workload set 1). For the other workload set, we fixed the number of requests to the light-weight and middle-weight services to 20 and increased that to the heavy-weight service from 0 to 40 (workload set 2). The clients sent requests to Tomcat until the number of concurrent requests to each service reached the specified number. When a client receives a response from Tomcat, the client sends a new request. The admission control provided by Tomcat did not work because the maximum number of concurrent requests described in the configuration file was 150.

Under such workload sets, we measured the performance of Tomcat on various operating systems: Solaris 9, Linux 2.6.7, 2.6.5, and 2.4.18, FreeBSD 5.2.1, and Windows 2003 Server Enterprise Edition. A server host was Sun Fire V60x, which had dual Intel Xeon 3.06GHz processors, 2GB memory, and 1Gbps Ethernet card. The processors enabled hyperthreading and the number of logical CPUs was four. The

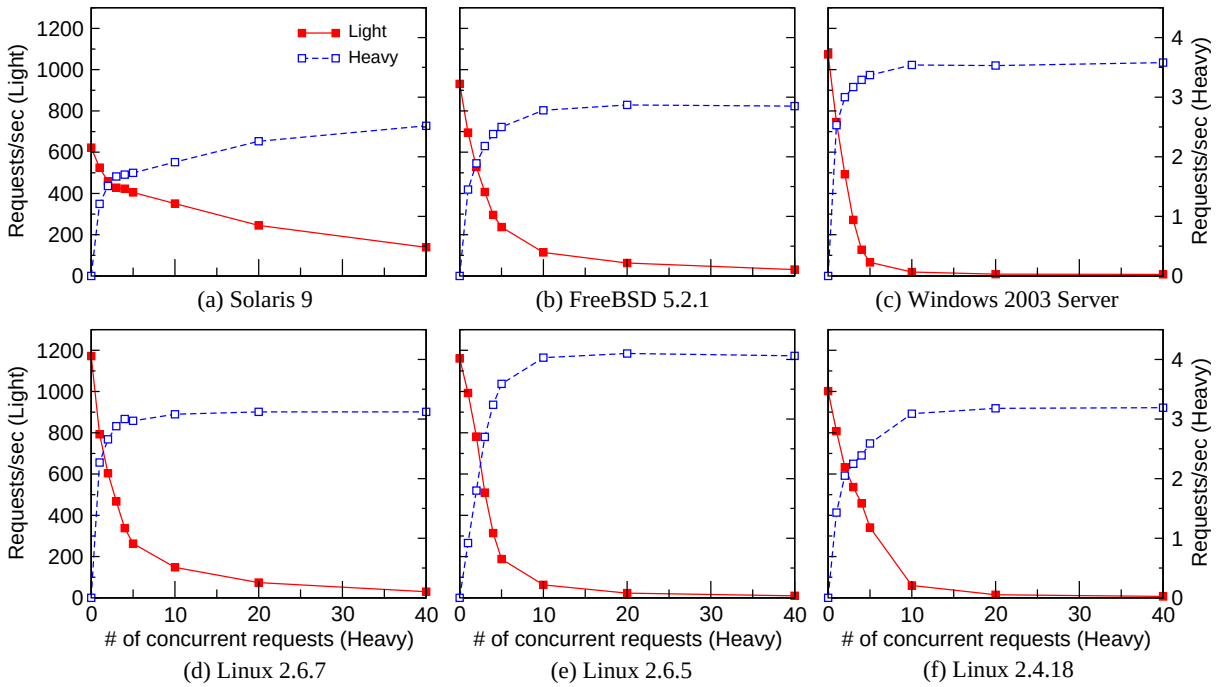


Figure 1. Degradation schemes of various operating systems. (workload 1)

version of Tomcat was 5.0.25 and that of JDK was 1.4.2. To generate workloads, we used eight client hosts, each of which had a Pentium 733MHz processor, 512MB memory, and a 100Mbps Ethernet card. The operating system was Linux 2.4.19. These hosts were interconnected by a 1Gbps Ethernet switch.

3.2 Results

Workload Set 1 Figure 1 (a)-(d) shows the throughputs of the services in different operating systems when we increased the number of concurrent requests to the heavy-weight service. In either operating system, the throughput of the light-weight service was degraded as the server load became higher. However, the degrees of the performance degradation were different as shown in Figure 2. The throughput in Solaris was degraded gracefully while those in the other operating systems were not. In Solaris, the throughput was decreased slowly by 78% when the number of requests to the heavy-weight service increased to 40. In the other operating systems, on the other hand, the throughput was decreased suddenly by 97% to 99%. Also, there were differences among Linux, FreeBSD, and Windows and the throughputs were degraded seriously in the order of Windows, Linux, and FreeBSD. On the contrary, the throughput of the heavy-weight service in Solaris was relatively lower than those in the others.

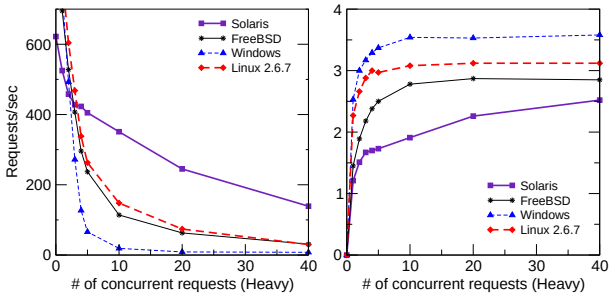
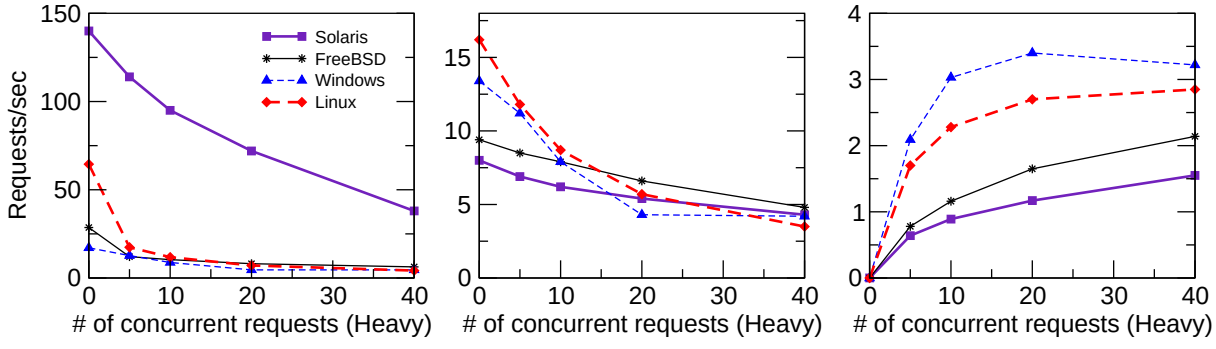


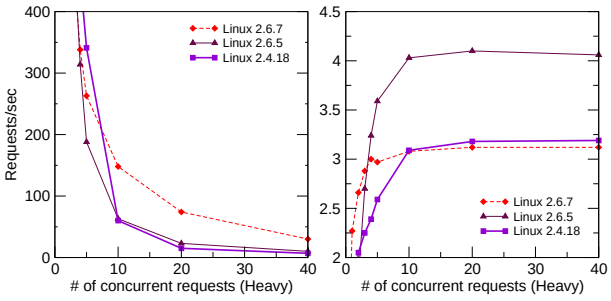
Figure 2. The difference of degradation schemes among different operating systems for each service. (workload set 1)

Figure 1 (d)-(f) shows the throughputs of the services in different versions of Linux. Based on this figure, degradation schemes for each service are plotted in Figure 4. The figure shows that the small difference of versions between 2.6.5 and 2.6.7 makes a large difference of degradation schemes. Under heavy load, version 2.6.5 and 2.4.18 showed similar throughput for the light-weight service while version 2.6.7 and 2.4.18 showed similar throughput for the heavy-weight service.



(a) Light-weight service (b) Middle-weight service (c) Heavy-weight service

Figure 3. The difference of degradation schemes among different operating systems for each service. (workload set 2)



(a) Light-weight service (b) Heavy-weight service

Figure 4. The difference of degradation schemes among different versions of Linux for each service. (workload set 1)

Workload Set 2 Figure 3 shows degradation schemes in different operating systems when we used three types of services. The overall throughput of the light-weight service was higher in Solaris than in the other operating systems. This is because the throughput of the light-weight service in the other operating systems was enough low due to resource conflicts with the middle-weight service, even if there was no request to the heavy-weight service. On the other hand, the throughput of the heavy-weight service in Solaris was worst. For the middle-weight service, there was no large difference.

4 Principal Factor of the Differences

In the previous section, we showed the differences of degradation schemes among operating systems. To control these degradation schemes, we must know the principal factor that causes the differences. To investigate the principal factor, we especially compared the behavior of the light-weight service in Solaris and Linux at the kernel level. We selected these two operating systems because the light-weight service in Solaris was degraded gracefully but that in Linux was degraded seriously. In this experiment, we used the same environment as described in Section 3. The number of requests concurrently posted was 30 for the light-weight service and 20 for the heavy-weight service.

4.1 Thread Processing Time for Each Request

To examine the breakdown of the thread processing time spent for each request to the light-weight service, we obtained the events on the CPU scheduling and the system calls about all the threads in Tomcat. We can distinguish the Java threads used by Tomcat at the kernel level because a Java thread is bound to a specific kernel thread in Solaris 9 and Linux. To trace such events in Solaris, we used the `prex (1)` command, which can record selected events occurred in the kernel. For Linux, we have developed a similar tool called *kev*. These tools use the performance counters of Pentium to measure time values and the resolution is microsecond at least.

From these event logs, we extracted only the events on the light-weight service. Since Tomcat uses the thread pool to reuse threads for request handling, we needed to distinguish the section of processing the light-weight service and that of processing the heavy-weight service in every kernel thread. In this experiment, 51 threads were created in advance to process 50 requests concurrently and wait for a next request. First, we divided the sequential events into sections from the end of an `accept` system call to that of the next `accept` system call for each thread. Next, we determined which service was executed for each section by

Table 1. The breakdown of the thread processing time of each request (ms).

	Solaris	Linux
running time	3.71	3.91
waiting time	137	375
(accept)	1.19	2.44
(poll)	0.41	19.4
(lock)	136	348

a client IP address recorded with an event of the `accept` system call in kev. In prex, however, since we cannot obtain such information, we determined the service executed for each section by whether the thread issued the `open` system call or not for reading a XML file.

Table 1 shows the breakdown of the thread processing time for each request to the light-weight service in Solaris and Linux. Of the total time, the running time was almost the same and these times are consumed primarily for executing the Fibonacci calculation. On the other hand, the waiting time is quite longer than the running time. The waiting time in Linux was 2.6 times longer than that in Solaris. That longer waiting time in Linux causes the serious performance degradation.

The waiting time primarily consists of the waiting time for I/O and for lock acquisition. In Solaris, the longest waiting for I/O was the `accept` system call and the next was the `poll` system call, which was issued once to wait for a request from a client. In Linux, the longest waiting was the `poll` system call and the next was the `accept` system call. In either case, the waiting time for I/O was short. On the other hand, the total waiting time for lock acquisition was longest. In Linux, the `futex` system call with the `FUTEX_WAIT` operation was issued to wait for lock acquisition. In Solaris, the `mutex_lock` and `cond_wait` system calls were issued for lock acquisition. This difference of the waiting time is the largest differences of the performance degradation between Linux and Solaris.

4.2 Long Waiting Time for Lock Acquisition

We investigated the reason why the lock waiting time for the light-weight service is longer in Linux than in Solaris.

4.2.1 Dominant Factor in Request Processing

We measured the waiting time for system calls on lock acquisition during request processing, which is a section from accepting a connection from a client to closing the connection and entering the thread pool. Table 2 shows the waiting time during request processing in Solaris and Linux. The total waiting time was longer in Linux than in Solaris by a factor of 2.3. However, the waiting time per system call was almost the same between Linux and Solaris. The difference between Solaris and Linux was the number of times of system calls issued. The number in Linux was 2.2 times of that in Solaris.

Table 2. The waiting time of system calls for lock acquisition. (ms)

	Solaris	Linux
per-call waiting time	64.6	65.2
call frequency	1.3	2.9
total waiting time	82.0	189

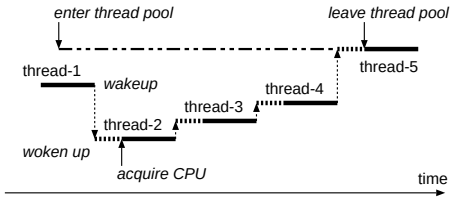


Figure 5. The thread scheduling in the thread pool .

One of the reasons is that the JVM 1.4.2 in Solaris is implemented so that the thread is not blocked as frequently as possible. The JVM first calls the `mutex_trylock` function to try to acquire a lock in the userland. If that try fails, the JVM calls the `mutex_lock` function, which may issue the `mutex_lock` system call and be blocked. Another reason is that the thread library in Solaris implements adaptive locks, which use both spin locks and the `mutex_lock` system call. If the thread can acquire a lock during spinning, it is not blocked. The other reason is that Linux does not have the `cond_wait` system call as provided in Solaris. Therefore, the `pthread_cond_wait` function includes four operations for lock acquisition, which may issue the `futex` system call and be blocked.

4.2.2 Dominant Factor in the Thread Pool

After a thread finishes processing a request, it enters the thread pool and waits until it is woken up by a thread leaving the thread pool, as illustrated in Figure 5. From Table 3, the waiting time in the thread pool was 2.9 times longer in Linux than in Solaris. This waiting time is the sum of the time from when a thread in the thread pool is woken up until it wakes up the following thread. The number of waiting threads in the thread pool and/or the execution time spent by each thread can be the cause of the large difference in the total waiting time. The number of waiting threads was

Table 3. The waiting time of threads in the thread pool. (ms)

	Solaris	Linux
per-thread execution time	4.34	11.7
# of waiting threads	12.7	13.0
total waiting time	52.6	154

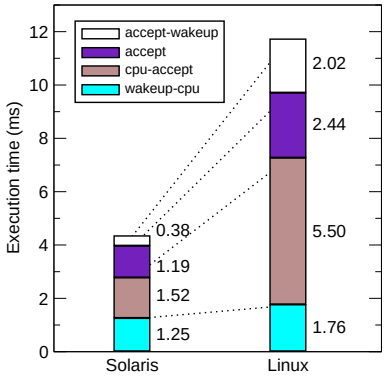
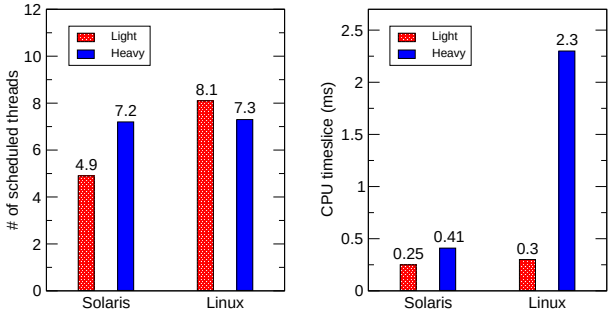


Figure 6. The breakdown of the execution time of each thread in the thread pool (ms).



(a) Scheduled threads (b) Timeslice for each thread

Figure 7. The total number of threads scheduled during execution of each thread in the thread pool and the time slice (ms).

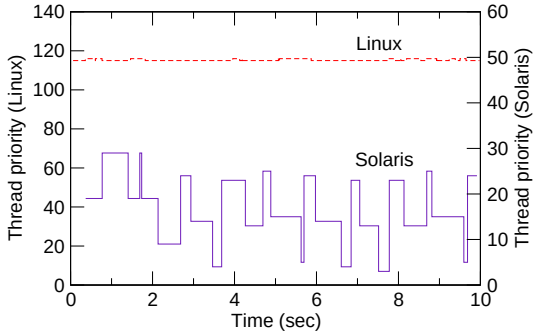


Figure 8. The change of the thread priority.

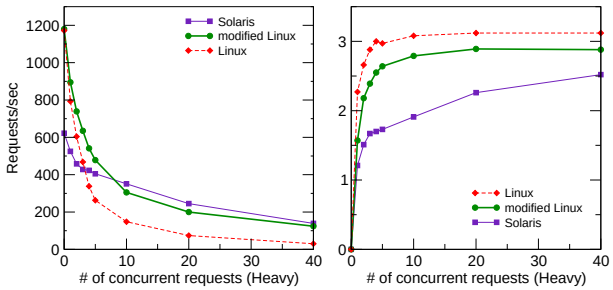
almost the same between Solaris and Linux. On the other hand, the execution time of each thread was 2.7 longer in Linux than in Solaris and it was the dominant factor.

Figure 6 shows the breakdown of the time spent by each thread in Solaris and Linux. After a thread is woken up, it enters one of the CPU run queues to wait for acquiring a CPU (wakeup–cpu). When it acquires a CPU, it issues the accept system call (cpu–accept). In the `accept` system call, the thread waits for a new connection from a client (accept). After it accepts a connection, it wakes up the following thread waiting in the thread pool (accept–wakeup). The time from when a thread is woken up until it acquires a CPU depends on the CPU scheduling. Until the CPU scheduler allocates a time slice for the thread, the thread cannot run even if it acquires a lock. In Linux, the time is 40% longer than in Solaris although this factor affects the total waiting time a little.

The dominant factor waiting in the thread pool is the time from when a thread acquires a CPU until it wakes up the following thread (cpu–accept, accept, and accept–wakeup). This time is 3.2 times longer in Linux than in Solaris. First, the time taken in the `accept` system call was longer in Linux. This waiting time depends on the request rate from the clients and becomes longer as the rate is lower. In our experiment, the rate was proportional to the server throughput because the number of concurrent requests was constant. Therefore, it is considered that this longer waiting time is not a primary factor but is secondarily caused by performance degradation of the server.

Next, we examined the detail of how a thread runs from when it acquires a CPU until it wakes up the following thread, except the `accept` system call (cpu–accept and accept–wakeup). The number of threads scheduled in this interval is shown in Figure 7 (a). The number of threads executing the light-weight service was larger in Linux as we expected. On the other hand, that of the heavy-weight service was almost the same between Solaris and Linux although the time spent for this interval was 4.0 times longer in Linux. The key to explain this phenomenon is the time slice of the CPU scheduling. As shown in Figure 7 (b), the time slice of threads executing the heavy-weight service is 5.6 times longer in Linux than in Solaris. The longer time slice is a dominant factor that makes the waiting time in the thread pool longer in Linux.

The cause of the longer time slice is the management of thread priority in the CPU scheduler in Linux. As shown in Figure 8, the thread priority in Solaris was variable and the



(a) Light-weight service (b) Heavy-weight service

Figure 9. The degradation scheme of Linux with short time slices. (workload set 1)

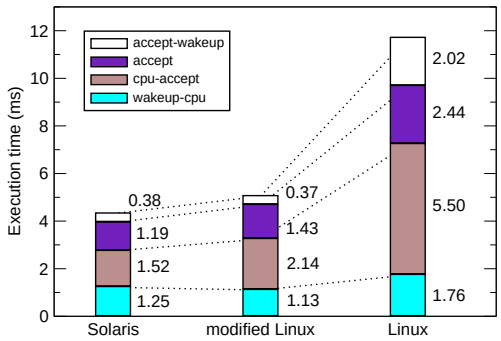


Figure 10. The breakdown of the execution time of each thread in the thread pool in Linux with short time slices.

thread execution was preempted frequently for CPU- and memory-intensive threads such as our heavy-weight service. On the other hand, a thread priority in Linux was almost constant, so that the CPU preemption occurred less frequently.

4.3 Linux with Shorter Time Slices

From the above experimental analysis, we found out that the time slice is one of the dominant factors of the difference between Solaris and Linux. To verify this analysis, we performed the same experiment with Section 3 using workload set 1 for Linux with shorter time slices. We changed the maximum time slice from 200ms to 2ms and the minimum time slice from 10 ms to 1ms by modifying the Linux kernel source. Figure 9 shows the result. When the server load became high, the performance degradation in the modified Linux was similar to that in Solaris in Figure 1. When the number of concurrent requests to the heavy-weight service was 20, the throughput in Linux was only 9% lower than that in Solaris. Comparing it with the result in the original Linux, the throughput of the light-weight service increased in a factor of 2.4.

We analyzed the details of thread processing time for the modified Linux. The total waiting time per request in the modified Linux decreased from 375ms to 161ms. Of that time, the waiting time in the thread pool decreased from 154ms to 66.9ms. The breakdown of the waiting time in the thread pool is shown in Figure 10 and the waiting time in every interval decreased. On the other hand, the waiting time in request processing also decreased from 189ms to 122ms. This is because the waiting time of a lock system call decreased from 65.2ms to 45.2ms.

From this result, it was shown that the performance degradation in Linux approaches to that in Solaris by changing the time slice, but we don't mention that the degradation scheme in Solaris is always the best. We only mention that the degradation scheme in Solaris is better in the case of this particular context. The important point is that it is possible to customize the degradation scheme according to the multiple services on some operating systems.

5 Future direction

A future direction of our research is to develop a middleware-level mechanism for controlling the degradation scheme. As we showed above, the behavior of a web application server with respect to the degradation scheme significantly depends on the underlying operating system. We believe that this difference should be absorbed at the middleware level so that application developers do not have to care about the behavior of an underlying operating system. The developers should be able to specify a degradation scheme appropriate for their service applications. The specified scheme should be automatically applied to a web application server no matter what the underlying operating system is. Thus, controlling the degradation scheme makes a web application server dependable because the server can sustain the intended quality of service even if the operating system is changed.

Such a middleware-level mechanism would help developers. For example, the operating system of the machine used for development might be different from that of the target machine, which would be an expensive big server machine. If the behaviors of the two operating systems are totally different with respect to the degradation scheme, the developers must use the target machine for performance tuning but using the target machine for development is sometime difficult in the real world. The middleware-level mechanism for the degradation scheme would solve this problem since the developer can specify an appropriate degradation scheme independently of underlying operating

systems.

We mentioned that kernel-level modification of the CPU scheduler could change the degradation scheme in Section 4.3. To control the degradation scheme at the middleware level, we plan to yield a part of CPU time of heavy-weight services to light-weight services by dynamically injecting sleep code into heavy-weight services. We believe that the injected sleep time can be determined based on the progress of light-weight services.

6 Related Work

As for workload of requesting static web pages, several researchers have already reported the behavior of a web application server under heavy workload. Almeida et al. examined the behavior of the Apache web server under heavy workload consisting of HTML files, images, sounds, and videos [1]. They reported that the bottleneck was I/O processing by the kernel, which spent 90% of the time for handling a request. Pradhan et al. pointed out that different workload causes a different bottleneck as for requests for static web pages [5]. When persistent HTTP connections were used, the bottleneck was the accept queue. When SSL encryption was used, on the other hand, it was the CPU run queue.

McWherter et al. reported that the performance bottleneck of a servlet accessing a database differs if the database is changed. With one database, the bottleneck was to acquire a lock for logical database objects while, with another database, it was I/O synchronization for processing online transactions [4]. They also reported that they observed different results when they changed the servlet.

Several priority-based scheduling mechanisms have been proposed for degrading the performance of each service to a different degree. To improve the response time of short connections, Crovella et al. proposed the shortest-connection-first scheduling [2]. Elnikety et al. applied a priority-based scheduling to web applications [3]. Neptune allows web administrators to define a function for computing a service yield for each service. It schedules web applications to maximize the sum of service yields [6].

Our contribution against the work above is that we investigated difference in degradation behavior among several major operating systems. The result of our investigation showed the significance of middleware-level mechanism that absorbs the difference among operating systems with respect to the degradation behavior.

7 Conclusion

In this paper, we reported that the degradation behavior of a web application server under heavy workload significantly depends on the underlying operating system. We investigated this fact with Solaris, Linux, FreeBSD, and Windows Server. We measured the throughput of the Tomcat web application server providing light-weight and heavy-weight services. The result was that the throughput of the light-weight service in Solaris was gradually degraded whereas ones in the others were steeply degraded. As a result of our further experiments on Solaris and Linux, it was revealed that the major factor of the difference is the waiting time for acquiring locks. One reason of the longer waiting time in Linux was that the larger number of system calls are issued for lock acquisition due to the implementation of the JVM and the thread library. The other reason was that the Linux scheduler gives precedence to the threads executing the heavy-weight service over the ones executing the light-weight service.

References

[1] J. Almeida, V. Almeida, and D. Yates. Measuring the behavior of a world-wide web server. Technical Report TR-96-025, Boston University, 1996.

[2] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, oct 1999.

[3] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the 13th International World Wide Web Conference*, pages 276–286, 2004.

[4] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proceedings of the 20th International Conference on Data Engineering*, pages 535–546, 2004.

[5] P. Pradhan, R. Tewari, S. Sahu, C. Chandra, and P. Shenoy. An observation-based approach towards self-managing web servers. In *Proceedings of the International Workshop on Quality of Service*, 2002.

[6] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 225–238, 2002.

[7] The Apache Software Foundation. Apache Jakarta Tomcat. `http://jakarta.apache.org/tomcat/`.

[8] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 230–243, 2001.