Abstract

On the Unix platform, it is a well-known problem that the polling I/O such as select() and poll() becomes a performance bottleneck especially when a server is heavily loaded by network I/O. To resolve the issue, we propose a unique technique that improves the efficiency of the polling I/O without modification to its semantics. The key idea for the high efficiency is a fine-grained interval control on the polling I/O, which reduces the invocation frequency and increases the number of I/O-ready sockets returned by each invocation. In addition, the simplicity of the implementation is the other advantage of our solution. The approaches proposed in other literature so far provide a special interface to notify the socket state changes to a server process, and therefore they require considerable modification in the server program and/or the operating system. Since our technique does not alter the current programming model of the polling I/O, it can be applied easily to any kind of network servers based on the polling I/O model. Benchmark tests on a web server accelerator confirmed that the interval control mechanism largely improves the I/O performance such as service throughput and response time.

1 Introduction

Unix is often utilized for providing a vast variety of internet services such as e-mail, NFS, and web. One of the reasons for the popularity of Unix as a network service platform is that the socket model for network I/O was established early in 1980’s. Based on this socket model, several advanced programming models have been developed. Among these models, I/O multiplexing by select() or poll(), which are called as polling I/O [1], plays an important role especially for high-performance servers. The polling I/O checks each condition of a set of sockets and waits until one of them changes. This mechanism makes it unnecessary for a server program to allocate a thread \(^1\) for each socket because it enables a single thread to handle more than one socket concurrently. As a consequence of this advantage, a large number of network server programs use this I/O multiplexing mechanism.

On the other hand, both a rapid growth of network bandwidth achieved in these years and a trend toward more sophisticated network services have boosted the request arrival rate at network servers. This phenomenon raises a problem that the polling I/O becomes a performance bottleneck [2]. To give a solution for this problem, several research groups have proposed new mechanisms that enhance the performance of I/O multiplexing. Unexpectedly, these solutions are based on a highly similar idea; they quit scanning the socket table and provide a special interface between the kernel and the server threads to notify the events of state changes in the sockets. This mechanism thus reduces the processing cost of the polling I/O and improves the network I/O performance. However, they raise another problem here that it is difficult to implement this mechanism because it requires significant modification in the server program and/or the operating system itself.

In this paper, we propose a unique technique that augments the performance scalability of the polling I/O model. The polling I/O has a characteristic that the smaller the invocation interval is, the smaller amount of useful information the caller gets. Consequently, the processing overhead of the polling I/O increases when it is requested too often. The technique we propose improves the efficiency of the polling I/O by controlling its execution intervals and enhances the server performance particularly from a viewpoint on scalability.

This paper is organized as follows. Section 2 gives a survey as a bird’s-eye view of the Unix network programming models for I/O multiplexing. In Section 3, we observe the previous work done by other research groups that tackled the performance issue of I/O multiplexing. From the techniques in the work, we especially pick up and describe a multi-accept server that made a hint for this research. In Section 4, we depict awkward aspects of the polling I/O and propose our solution: control of I/O polling intervals. The performance improvement achieved by our solution is confirmed with benchmark tests on a web server accelerator\(^2\) in Section 5. Herein, we also profile the behavior of the operating system during the benchmark tests and present how much processing cost is reduced by our technique. Finally in Section 6, we conclude this paper and state our future work.

---

\(^1\)In this paper, the term “thread” denotes a minimum execution unit scheduled by an operating system.

\(^2\)A web server accelerator is also called a web reverse proxy.
2 Programming Models for Network I/O Multiplexing on Unix

To achieve high performance on network I/O, a single thread should handle multiple sockets simultaneously to avoid various overheads such as frequent context switches and large memory consumption. In this section, we survey the network programming models for I/O multiplexing on Unix. From a view of I/O multiplexing, the current Unix network programming models are categorized into three groups: Non-blocking I/O, Polling I/O, and Signal-driven I/O [1].

2.1 Non-blocking I/O

Non-blocking I/O on a socket is enabled when an O_NONBLOCK flag is set on the socket with fcntl() \(^3\). Non-blocking I/O on a socket never blocks a thread that requests the I/O because the operating system executes only a part of the I/O task that can be completed without blocking. For example, when there is no data to be read on a socket, non-blocking read I/O thus returns to the calling thread immediately without data.

Theoretically, the non-blocking I/O makes it unnecessary to check the state of each socket before its execution. However, there arises a problem that a huge number of wasteful read I/O requests on sockets are issued because the server program cannot know \textit{a priori} whether there is some data to read or not. Therefore, non-blocking I/O is not an efficient method for I/O multiplexing, and it should be used with the polling I/O described in the next subsection.

2.2 Polling I/O

As we stated in Section 1, the polling I/O is a programming model implemented by select() and orpoll(). It enables a server thread to request I/O selectively only on the I/O-ready sockets. The simple and strong feature of the polling I/O has led many developers to adopt this I/O multiplexing model for high performance network servers.

On the other hand, recent researches discovered that the polling I/O is an expensive mechanism especially when it handles a large number of concurrent sockets. The problem and solutions described by them are summarized in Section 3.

2.3 Signal-driven I/O

Signal-driven I/O is a programming model that uses a signal for event notification of a socket state change. It is enabled when a signal and a process id for the event notification are specified by the F_SETSIG command and the F_SETOWN command of fcntl() respectively.

In reality, we need POSIX real-time signals [3] to adopt the signal-driven I/O model because there are difficulties in using the signal-driven I/O with traditional Unix signals [4]. The real-time signals are sent to a process via a signal queue that allows the process to receive the signals one by one asynchronously. In addition, each real-time signal can contain some information, thus it can notify the descriptor of an I/O-ready socket to the process.

The signal-driven I/O with real-time signals still has a problem. If the socket is closed before the signal is removed from the queue, the information that will be delivered by the signal becomes stale. To avoid such false event notifications, server programs have to handle closed sockets carefully.

There is also a problem of queue overflow. When a signal queue for a server process overflows, some event information is lost and the server has to switch back to some other programming models such as the non-blocking I/O model or the polling I/O model. However, such a switch makes the situation worse because it increases the processing cost although the queue overflow means the server is already overloaded [5].

3 Related Work

This section observes the previous work on the performance issue of I/O multiplexing models. From the techniques presented in them, we pick up and describe a \textit{multi-accept server}, which showed an interesting behavior and made a hint for us to conduct this research.

3.1 Previous Work on I/O Multiplexing

Several research papers [2, 6, 5, 7, 8] have been published on the performance issue of I/O multiplexing, especially on the severe performance degradation of the polling I/O with an increase of the number of concurrent sockets. To address this issue, they proposed various solutions. Although the approaches they took are different from each other, their basic ideas are almost the same; they developed a mechanism that an operating system records a state change of each socket and notifies the change to a server process asynchronously through a special interface.

Banga et al. [2] developed a technique that a server process registers a set of sockets as an “interest set” in an operating system and the operating system keeps watch on the sockets and notify the state changes to the process through a special event queue between the kernel and the process.

Lemon [6] developed a general-purpose interface called “kqueue” on FreeBSD and enhanced the event notification mechanism to solve the false notification problem described in Section 2.3.

Provos et al. [5] developed a similar interface, /dev/poll\(^4\), on Linux and evaluated its performance. In addition, they compared the performance of the /dev/poll interface and that of the signal-driven I/O model with real-time signals. They also proposed an improved mechanism of the

\(^3\)Note that this kind of operations on a file descriptor can be performed also by ioctl().

\(^4\)The /dev/poll interface was first developed on Solaris by Sun Microsystems [9].
signal-driven I/O model [7]. It allows a server process to receive multiple real-time signals at once.

Chandra et al. [8] discussed the issue of signal queue overflow. They developed a new signal queue that holds only one entry for each socket to resolve that issue. The mechanisms proposed by these previous work remove the processing cost of scanning the sockets in the polling I/O. On the other hand, it is a tough work to implement such a mechanism in an existing network server system based on the polling I/O model. The proposed programming models are rather different from the original polling I/O model and therefore they require considerable modification in the server program and/or the operating system.

3.2 A Multi-accept Server

The multi-accept server is a server with a simple technique mentioned in [8]. It is based on the traditional I/O multiplexing model using \texttt{select}(). The only modification introduced to the multi-accept server is that it invokes \texttt{accept}() multiple times, not once, when the listening port turns out to be I/O-ready by \texttt{select}(). This technique was proved to achieve moderate performance scalability compared with a traditional server based on \texttt{select}(). Such scalability is the common goal of the other techniques proposed by the earlier work.

This interesting phenomenon made a hint for our research because it suggests that the polling I/O itself does not include a critical drawback in its processing framework. In other words, there is a chance that we can improve the performance scalability of the polling I/O.

4 Problems of Polling I/O and Our Solution

In this section, we first examine the performance issue of the polling I/O in detail. Then, we propose a fine-grained control mechanism on the invocation intervals of the polling I/O.

4.1 Problems of Polling I/O

The poor performance scalability of the polling I/O is caused by the following two attributes. One is the synchronization of \texttt{accept}() to the polling I/O, which degrades the connection processing rate. The other is the severe CPU starvation, which increases service delay.

1. \textit{Synchronization of \texttt{accept}() to polling I/O}

The improved performance of the multi-accept server implies that one of the problems with the traditional polling I/O model is that the invocation rate of \texttt{accept}(), which is equivalent to the connection processing rate, depends tightly on that of \texttt{select}(). The situation is described more precisely as follows. As the network speed increases, also the number of concurrent sockets and the invocation rate of \texttt{select}() grows (this phenomenon will be revisited later). However, this tendency changes at a certain point because the processing time of each \texttt{select}() call increases as well. This means that there is an upper limit to the invocation rate of \texttt{select}(), i.e., that of \texttt{accept}(), and it can be lower than required. When the invocation rate of \texttt{accept}() gets lower than the connection arrival rate, some new connections are refused by the server. This bottleneck can be removed easily by breaking the tight dependency between \texttt{accept}() and \texttt{select}() as the multi-accept server does.

2. \textit{CPU starvation by frequent and inefficient polling I/O}

The frequency of events that the state of a socket changes to I/O-ready is directly proportional to the network speed. This indicates that the growth of network speed raises the invocation rate of the polling I/O. The high invocation rate of the polling I/O consumes a substantial proportion of the CPU cycles. To make matters worse, each polling I/O processing is quite inefficient. The amount of useful information returned by a single invocation of the polling I/O is usually small. Here, we have defined the amount of the useful information as the number of I/O-ready sockets (NRS) returned by the polling I/O. This NRS is just one in the worst case although the size of a socket set given as an argument to the polling I/O grows when the network speed increases. Therefore, the efficiency of the polling I/O, which is defined as the ratio of NRS to the total number of concurrent sockets, declines when the socket set becomes large.

4.2 Fine-Grained Interval Control on Polling I/O

The goal of this study is to resolve the problem of CPU starvation caused by the polling I/O and enhance the total performance of network servers based on the polling I/O. All the previous work mentioned in Section 3 focused on the problem of high processing cost of the polling I/O. We focus on how to decrease its invocation frequency and how to increase the amount of returned information.

Figure 1 (a) shows the process flow cycle of the traditional I/O multiplexing model with the polling I/O. Our technique extends the processing cycle. As depicted in Figure 1 (b), the extension of the cycle can be implemented by self-blocking for a short time before the polling I/O if the cycle is too short. This technique decreases the invocation rate of the polling I/O and increases the NRS of each invocation. Therefore, it achieves an improvement in the efficiency of I/O polling especially on a busy network server.

Here, we have to consider the influence of the interval control on service delay. Increasing the interval of processing cycle means that also the service delay can be increased. The parameter value for the interval should be therefore chosen carefully. This viewpoint is quite important and we will discuss this topic later in Section 5.2.
4.3 Implementation

We implemented our solution in a Chamomile web server accelerator [10]. Chamomile has a multi-threaded architecture; it spawns several threads with different roles. One of them is a thread that invokes accept() on a listening port and passes the new socket to another thread that processes requests from the client. In this paper, we call the latter thread as an I/O thread. Chamomile thus has already solved the problem of the synchronization of accept() to the polling I/O.

Figure 2 depicts the pseudo code implementation of the I/O thread with the interval control mechanism. Although there is no explicit expression of the polling I/O, the function proc_conns() calls poll() and executes actual I/O on each socket returned by poll(). The only essential modification into Chamomile is an addition of code between the line 7 to 18. Thus, the I/O thread checks the processing time of the main loop\(^5\). If the time is shorter than a threshold, which can be specified in the configuration, the I/O thread sets blocking time depending on the processing time. As a consequence, the interval of the processing cycle is kept larger than the specified time. If the processing time is found to be larger than the specified time, the I/O thread does not block itself and just yields to some other thread.

5 More precisely, the I/O thread checks the elapsed time in the main loop.

5.1 Web Polygraph

In the benchmark tests, we used Web Polygraph [11], a benchmark software for web cache systems. Contrastingly, most of the researches mentioned in Section 3 used httperf [12] to evaluate the server performance. Our main intention to use Web Polygraph is to evaluate the server performance on a more realistic environment. Web Polygraph is a benchmark system that evaluates the performance of a web cache system including a web server accelerator, and therefore we cannot compare our results directly with the results obtained from httperf.

Web Polygraph has a prepared workload called WebAxe-4 [13], which takes into account content size distribution, content modification, dynamic contents, and HTTP/1.1 persistent connections. We can thus estimate the influence of the interval control mechanism upon the overall server performance. This viewpoint is notably useful for server developers because they can weigh the implementation cost and the performance improvement.
5.1.3 Reliability of Test Results

To gain a reliable evidence for the target performance by a benchmark test, the target should not be given an excessive load. In general, the behavior of an overloaded target depends on the implementation of not only the target program but the benchmark system itself. It is therefore difficult to discuss the

socket. In that case, the thread writing the data chunk may be blocked by the kernel.

For a fair discussion, we have to give much attention to the number of concurrent sockets because the processing cost of each polling I/O is highly dependent on it. When the persistent connections are enabled, the number of concurrent connections can be considerably large, which is determined by a variety of factors such as connection/request arrival rate, users’ behavior, and connection timeout. On the other hand, the main purpose of the experiments is to analyze the performance of the polling I/O with a relatively large number of concurrent sockets. It is therefore a good idea to keep the number of concurrent connections constant.

To achieve that end, we implemented in Chamomile a mechanism that controls the connection timeout dynamically, considering the number of concurrent connections. The parameter we added to Chamomile was the upper threshold number of concurrent sockets, which was set at 7000\(^7\). To keep the number of concurrent sockets around the threshold, Chamomile checks it periodically and decreases the connection timeout whenever it detects the number of connections exceeding the threshold. The connection timeout is increased by one second when the number of connections stays below the threshold for consecutive 10 seconds.

To find the best interval of the processing cycle, we also parameterized the minimum invocation interval (MII) of the polling I/O during the experiments (Table 2). It ranged from 0 ms to 20 ms with a step of 5 ms when HTTP/1.1 persistent connections were disabled. Otherwise, it ranged from 0 ms to 40 ms with a step of 10 ms. The reason of the difference between these two configurations is the large difference in the number of concurrent sockets. As we mentioned before, the number of sockets was kept around 7000 when the persistent connections were enabled. This situation made the processing time of the I/O thread large and required some longer MII. In both cases, the interval of 0 ms means that the I/O thread never blocked itself by nanosleep().

5.1.1 Network Environment

Figure 3 illustrates the experimental environment for Web Polygraph tests with WebAxe-4 workload. All the hosts including a client cluster, a server cluster, and a web server accelerator are connected to a single switch. Each client sends an HTTP request to the accelerator and the accelerator returns a response directly to the client in case of a cache hit, or forwards the request to one of the servers and relays the response to the client in case of a cache miss. The specification of the hosts and the switch used for the tests is listed in Table 1.

5.1.2 Configuration of Chamomile

During the benchmark tests, Chamomile was configured as follows. The size of content cache area was set at 768MB, all prepared on main memory. Because WebAxe-4 puts an assumption that the size of content working set on the server cluster is 1GB, most of the requests to cachable contents hit the cache. We observed that Chamomile achieved 73 – 75 % hit rate against the request sequences generated by Web Polygraph, whose theoretical maximum hit rate is 80 %.

We also decided to use non-blocking I/O for general I/O, such as read() and write()\(^6\). We do not have to set the sockets non-blocking in most cases because each socket is checked whether it is I/O-ready or not by the polling I/O. However, we cannot ignore the possibility that enough buffer is unavailable especially when a large data chunk is written to a

\(\text{HTTP/1.1 PC}^1\) | disabled / enabled
---|---
\(\text{MII}^2\) of polling I/O | 0 ms – 20 ms (PC disabled) 0 ms – 40 ms (PC enabled)

\(^1\)PC: Persistent Connection
\(^2\)MII: Minimum Invocation Interval

---

\(\text{Indeed, Chamomile uses one of }\text{send()}\), \(\text{recv()}\), \(\text{sendmsg()}\), and \(\text{recvmsg()}\) \(\text{according to the type of I/O buffer.}

\(^7\)The absolute upper limit of the number of concurrent sockets was also set statically at 10000.
performance of the target using benchmark results that include some error states of the target or the benchmark system itself.

The previous studies mentioned in Section 3 adopted experiment results with errors to derive some suggestion on the behavior of an overloaded server. On the other hand, we determined not to use such results for discussion. Each client of Web Polygraph enqueues connection requests and HTTP requests into its internal request queues at the configured rate. If the request rate is higher than the server capacity, some of those queues overflow in a short term, and the result is therefore not reliable.

5.1.4 Resolution of Kernel Timer

Our solution requires blocking the I/O thread for a short time in milliseconds by `nanosleep()`. Although `nanosleep()` receives an argument of sleep time in nanoseconds, most of the implementation of Unix operating systems do not have such a fine-grained timer mechanism. The source code of Linux, which we used as a development and experiment platform, has a macro definition of `HZ` and the resolution of the kernel timer mechanism is defined as 1/`HZ` second. In our experiments, we defined the value of `HZ` as 1000, which means that the resolution of the kernel timer is 1 ms. Also Web Polygraph, which works on customized FreeBSD [14], uses such a fine-grained kernel timer.

5.2 Throughput and Response Time

We present the results of the benchmark tests and discuss the influence of the interval control mechanism on the server throughput and the response time. The experiment results are categorized into two cases, with and without HTTP/1.1 persistent connections. Each result is taken from the top2 phase [13] of the WebAxe-4 workload. During the top2 phase, the request rate from the clients is held at the maximum.

5.2.1 Case without HTTP/1.1 Persistent Connections

When HTTP/1.1 persistent connections are disabled, a client establishes a TCP connection for each HTTP request. This means that the connection arrival rate is almost the same as the request rate.

Figure 4 depicts the relationship between the request rate and the mean response time. The left, center, and right graphs are for all responses, cache hits, and cache misses respectively.

From these graphs, we found that the interval control on the polling I/O changes the characteristics of performance scalability drastically. When the control was disabled, i.e., the MII of the polling I/O was 0 ms, the response time increased almost linearly as the request rate grew. With the interval control on the polling I/O, the response time stayed at an almost constant value regardless of the request rate.

We also have to pay attention to the growth of the response time according as the interval of the polling I/O increases. When the request rate was relatively low, the response time in case with the over 10 ms interval control exceeded that of the original Chamomile.

Lastly, the graphs show that the interval control on the polling I/O enhances the server performance from a viewpoint of throughput. When the control was disabled, the maximum throughput was 1800 req/s. With the control, the maximum
throughput was 2000 req/s, except the case when the MII was set at 5 ms.

5.2.2 Case with HTTP/1.1 Persistent Connections

When HTTP/1.1 persistent connections are enabled, a client can send more than one HTTP request on a connection. Thus, the connection arrival rate is much lower than the request rate. As we mentioned before, we implemented a mechanism that dynamically controls the connection timeout value in Chamomile and configured the upper threshold of the number of concurrent sockets at 7000. Therefore, the number of concurrent sockets was kept around 7000 during the top2 phase of the tests.

Figure 5 depicts the relationship between the request rate and the mean response time. The order of the graphs is the same as Figure 4, and the graphs show both similar and different performance behavior compared with the case without persistent connections.

The similar behavior is the improvement in the maximum throughput. When the interval control was disabled, the maximum throughput was 1800 req/s. When the MII of the polling I/O was set at either 30 ms or 40 ms, the maximum throughput was increased up to 2200 req/s.

On the other hand, we observed some different behavior from a viewpoint of mean response time. First, the response time was improved by the interval control for all tested request rates. When the persistent connections were disabled, the interval control could make the response time larger than that of the original Chamomile. In case with the persistent connections, the interval control decreased the response time even when the request rate was relatively low. Next, the response time, however, increased as the request rate grew compared with the flat response time distribution we saw in Figure 4. We will revisit this phenomenon later.

5.3 Kernel Profile Analysis

To profile the behavior of the operating system during the tests, we used kernprof [15] version 1.5, which is a kernel profiler for Linux. The kernel profiling was done for five minutes during the top2 phase for each test to collect program counter (PC) samples. We discuss the results in case with and without persistent connections similarly to the throughput and response time analysis in the previous subsection.

5.3.1 Case without HTTP/1.1 Persistent Connections

The graphs in Figure 6 show the behavior of the operating system when the persistent connections were disabled. The left and the center graphs express the kernel CPU usage and the total CPU usage. The right one depicts the CPU consumption ratio of the poll() system call including its subcontract routines to all the kernel functions.

Examining the left and the center graphs, we see the CPU usage was considerably saved by the interval control of the polling I/O. When the control was disabled, all the CPU cycles were consumed, and the kernel accounted for a large portion...
(70 – 80 %) of them. The steady rise in the kernel CPU con-
sumption means that the amount of the CPU resource available
to Chamomile decreased although the request rate increased.
That is the major reason of almost linear increase in the re-
response time in case when the interval control was disabled (Figure 4).

The right graph shows clearly that the interval control of
the polling I/O gave much influence on its CPU consumption.
When the interval control was enabled, the CPU consumption
ratio of the polling I/O in the kernel stayed low, under 10%. On
the other hand, without the interval control, the polling I/O con-
sumed a substantial portion of CPU resource especially when
the request rate was relatively low. Herein, we can mention
an interesting behavior of the polling I/O. Contrary to our in-
tuition, we see that the CPU consumption by the polling I/O
decreased linearly as the request rate grew. This phenomenon
means that also the invocation rate of the polling I/O decreased.
Thus, the critical problem is not in the polling I/O itself but in
the severe CPU starvation caused by its programming model.

5.3.2 Case with HTTP/1.1 Persistent Connections

When the persistent connections were enabled, the system be-
behavior changed considerably compared with the case when they
were disabled. Figure 7 shows the results, and the order of the
graphs is the same as Figure 6.

First, we see that the interval control did not decrease the
kernel CPU consumption so much as the case without persist-
ent connections. In addition, the total CPU consumption was
on a high level as shown in the center graph. Even when we
enabled the interval control, the CPU cycles were completely
consumed when the request rate was higher than 1400 req/s.
This phenomenon of the CPU starvation explains the increase
in the response time (Figure 5).

The right graph in Figure 7 shows that the interval control
decreased the CPU consumption ratio of the polling I/O to some
degree. However, it was much higher than that in case without
persistent connections (Figure 6) and both the results with and
without the interval control show the similar behavior.

6 Concluding Remarks and Future
Work

This paper proposed the mechanism of interval control on I/O
polling. It decreases the high load caused by the polling I/O
and improves the overall performance of network servers. Be-
cause this mechanism does not alter the current I/O multiplex-
ing model of the polling I/O, it is quite easy to implement in
network servers.

With the benchmark tests, our mechanism was proved to
have a great advantage from a viewpoint of performance scal-
ability particularly when the number of concurrent sockets was
small. In that case, the distribution of the mean response time
against the request rate was completely flat. On the other hand,
the service delay of the system was very sensitive to the delay
added by the interval control, and we therefore have to keep the
MII small enough to avoid the increase in the service delay.

Also when the number of the sockets was large, the perfor-
mance was improved largely. However, the response time in-
creased slowly as the request rate grew even with the interval
control mechanism. One of the reasons of this steady increase
in the response time is the severe CPU starvation on the server.
Especially in case of a multi-threaded server program, the CPU
starvation can degrade severely the service delay.

For further improvement, we need a thorough study on the
behavior of multi-thread scheduling mechanisms. When a large
number of concurrent connections are given to the polling I/O,
the interval should be set almost equal to or larger than the
CPU time slice allocated to each thread by the operating sys-
tem. In that case, the number of concurrent threads, the MII of
the polling I/O, and the length of the time slice give somewhat
chaotic influence on the total performance. We are currently
working on a performance improvement by employing some
real-time scheduling mechanisms.

References

[1] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels,
and John S. Quarterman. The Design and Implementation
of the 4.4 BSD Operating System. Addison-Wesley, 1996.

scalable and explicit event delivery mechanism for UNIX. In
USENIX Annual Technical Conference, pages 253–
265, June 1999.


Linux. In 2000 USENIX Annual Technical Conference,
June 2000.

notification facility. In 2001 USENIX Annual Technical

zizing the Overhead Behavior of a Simple Web Server. In

[8] Abhishek Chandra and David Mosberger. Scalability of
Linux Event-Dispatch Mechanisms. In 2001 USENIX An-

[9] Shridhar Acharya. Using the devpoll (/dev/poll)
http://access1.sun.com/techarticles/
devpoll.html
Table 3: Requests and responses defined in WebAxe-4.

<table>
<thead>
<tr>
<th>Method</th>
<th>GET: 98.4%, POST: 1.5%, HEAD: 0.1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Control</td>
<td>200 to IMS Requests(^\d): 5%, 304 to IMS Requests: 10%, No Cache (reload): 5%, No Control: 80%</td>
</tr>
</tbody>
</table>

\(^\d\)An IMS request is a request including an “If-Modified-Since” line in its header.

Table 4: Content types defined in WebAxe-4.

<table>
<thead>
<tr>
<th>Type</th>
<th>Response Size</th>
<th>Cachability</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>exp. dist. (mean: 4.5KB)</td>
<td>80.0%</td>
<td>65.0%</td>
</tr>
<tr>
<td>HTML</td>
<td>exp. dist. (mean: 8.5KB)</td>
<td>90.0%</td>
<td>15.0%</td>
</tr>
<tr>
<td>download</td>
<td>log-normal dist. (mean: 300KB, std. dev.: 300KB)</td>
<td>95.0%</td>
<td>0.5%</td>
</tr>
<tr>
<td>other</td>
<td>log-normal dist. (mean: 25KB, std. dev.: 10KB)</td>
<td>72.0%</td>
<td>19.5%</td>
</tr>
</tbody>
</table>


Appendix

A Web Polygraph and WebAxe-4 Workload

Web Polygraph [11] is a benchmark tool suite for performance evaluation of web cache servers with precise and realistic workload. In the benchmark tests, Web Polygraph uses a client cluster and an origin server cluster. For benchmarking a web accelerator, the WebAxe-4 workload [13] can be used, which is enclosed in the workload set of Web Polygraph. In this appendix, we give a brief overview of the WebAxe-4 workload.

A.1 HTTP transactions on WebAxe-4

Table 3 and Table 4 are the summary of the characteristics of HTTP transactions defined in WebAxe-4. A huge number of requests with those characteristics are sent to the web accelerator, and the accelerator returns cached content in case of cache hit or relays the requests to one of the origin servers in case of cache miss. The total size of the content working set, which is a content set frequently accessed by the client cluster, is set at 1GB.

A.2 Network Environment on WebAxe-4

To emulate a realistic environment, delay and packet loss rate is inserted in the network that connects the client cluster and the web accelerator. These delay and packet loss are emulated by Dummynet [16] of FreeBSD, which is a recommended platform for Web Polygraph tests. The inserted network delay is 40 milliseconds and the packet loss rate is 0.005% in both direction on the network. In addition, as latency of the origin server service, delay of 300 milliseconds is inserted also in the origin server cluster.

Last, we have to mention the amount of network traffic generated in the test. There are two major factors that give a large influence on the traffic volume: the request rate and the cache hit ratio. The cache hit ratio is dominantly dependent on the size of the cache implemented in the accelerator, which we fixed at 768MB, all prepared on main memory. The reason we prepared all the cache area on the main memory is that the goal of the benchmark tests for this study is to evaluate the network I/O performance, not the I/O performance of disk drives. As a consequence, the traffic volume generated in the test was almost directly proportional to the request rate, about 45 Mbps against 1,000 request/s.