ABSTRACT
On the Unix platform, it is a well-known problem that the polling I/O such as `select()` and `poll()` causes a performance bottleneck especially when a server is heavily loaded by network I/O. To resolve this issue, we propose a unique technique that improves the efficiency of the polling I/O without major modification to its semantics. The key idea for the high efficiency is a fine-grained interval control of the polling I/O, which avoids excessively frequent invocations of the polling I/O. In addition, the simple 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 confirmed that the interval control mechanism largely improved the I/O performance such as service throughput and response time.

KEY WORDS
polling I/O, interval control, scalability, `select()`, `poll()`, socket programming

1 Introduction
Several advanced socket programming models have been developed to support sophisticated network servers. Among these models, I/O multiplexing by `select()` or `poll()`, which are called as polling I/O [1], plays an important roll especially for high-performance servers. The polling I/O receives a socket list as an argument, checks the condition of each socket, and waits until one of them gets I/O-ready. 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, it is a well-known problem that the polling I/O causes 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 similar viewpoint; they provide a special interface between the kernel and the server threads to notify the events of state changes in the sockets. Although such solutions improve the network I/O performance, they raise another problem of high implementation cost. Because it requires significant modification in the server program and/or the operating system, it is not easy to integrate such mechanisms especially in a system already developed with the polling I/O model.

In this paper, we propose a unique technique that augments the performance scalability of a network server without major modification in the semantics of the polling I/O. The real problem of the polling I/O is in its event-driven mechanism, i.e., its invocation rate grows too high as the network speed increases. This causes the starvation of CPU cycles and degrades the server performance. The mechanism we propose controls the invocation interval of the polling I/O and avoids such CPU cycle starvation.

This paper is organized as follows. Section 2 gives a brief survey on the previous work that tackled the performance issue of I/O multiplexing. In Section 3, we revisit the awkward aspects of the polling I/O and propose the interval control mechanism. The performance evaluation is discussed in Section 4. The performance improvement achieved by our solution was confirmed with benchmark tests on a web server accelerator\(^2\). We also profiled the behavior of the operating system during the benchmark tests and present how much processing cost was reduced by our technique. Finally in Section 5, we conclude this paper and state our future work.

2 Related Work
This section observes previous work on the performance issue of the I/O multiplexing models. Other basic techniques such as non-blocking I/O, polling I/O, and signal-driven I/O, are described in several textbooks [1, 3].

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

\(^1\)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.
3.2 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. 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 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. This technique decreases the invocation rate of the polling I/O and increases the NRS of each invocation. Therefore, it improves the efficiency of I/O polling especially on a busy network server.

Here, we have to consider the influence of the interval control on the 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 4.2.

3.3 Implementation

To evaluate the interval control mechanism, we implemented it 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.

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 the addition of code from the line 7 to 21. Thus, the I/O thread checks the processing time of the main loop. If the time is shorter than a threshold, which can be specified in the extended configuration, the I/O thread sets blocking time depending on the processing time. As a consequence, the interval of the processing cycle is kept longer 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.

4 Performance Evaluation and Kernel Profiling

In order to verify the influence of the interval control on the system performance, we carried out benchmark tests. We also profiled the operating system kernel during the tests.

4.1 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 2 used httperf [12] to evaluate the server performance. Because Web Polygraph is a benchmark system that evaluates the performance of a web cache system including a web server accelerator, we cannot compare our results directly with the results obtained from httperf. Our main intention to use Web Polygraph is to evaluate the server performance on a more realistic network service environment.

Web Polygraph has a prepared workload called WebAxe-4 [13], which allows for content size distribution, content modification, dynamic contents, HTTP/1.1 persistent connections, and network environment. 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.

4.1.1 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. On the other hand, to emulate a realistic network environment, network delay, packet loss rate, and service delay are configured. 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.

4.1.2 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 the main memory. Because WebAxe-4 puts...
Table 1. Devices used for benchmark tests.

<table>
<thead>
<tr>
<th>Server</th>
<th>Pentium III 866MHz, 512MB, Intel Pro/100, FreeBSD 4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Pentium III 1.4GHz, 512MB, Intel Pro/100, FreeBSD 4.3</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Pentium III 800MHz, 2GB, NetGear GA620T (1000baseT)</td>
</tr>
<tr>
<td>Switch</td>
<td>NetGear FS518T (1000baseT×2, 100baseT×16)</td>
</tr>
</tbody>
</table>

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 used non-blocking I/O for general I/O, such as read() and write()\(^5\). 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 space is unavailable especially when a large data chunk is written to a 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 HTTP/1.1 persistent connections are enabled, the number of concurrent connections can be considerably large, which is determined by a variety of factors, e.g., 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, depending on the number of concurrent connections. The parameter we added to Chamomile was the upper threshold number of concurrent sockets, which was set at 7000\(^6\). 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\(^7\)) 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().

4.1.3 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 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 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.

4.2 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: the one is the case without HTTP/1.1 persistent connections and the other is that without the 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 sustained at the configured maximum value for four hours.

4.2.1 Case without HTTP/1.1 Persistent Connections

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.

Table 2. Parameters for benchmark tests.

<table>
<thead>
<tr>
<th>HTTP/1.1 PC(^\dagger)</th>
<th>disabled / enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>MII(^\ddagger) of polling I/O</td>
<td>0 ms – 20 ms (PC disabled)</td>
</tr>
<tr>
<td></td>
<td>0 ms – 40 ms (PC enabled)</td>
</tr>
</tbody>
</table>

\(^\dagger\)PC: Persistent Connection  
\(^\ddagger\)MII: Minimum Invocation Interval

\(^5\)Indeed, Chamomile uses one of send(), recv(), sendmsg(), and recvmsg() according to the type of I/O buffer.  
\(^6\)The absolute upper limit of the number of concurrent sockets was also set statically at 10000.
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 as the request rate grew. With the interval control on the polling I/O, the response time stayed constant 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. With relatively low request rate, the response time could be larger than that without the interval control.

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

### 4.2.2 Case with HTTP/1.1 Persistent Connections

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 graphs in Figure 4.

The similar behavior is the improvement in the maximum throughput. When the interval control was disabled, the maximum throughput was 1800 req/s. With the interval control, the maximum throughput was increased up to 2000 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.

### 4.3 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.

4.3.1 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 consumption means that the amount of the CPU resource available to Chamomile decreased although the request rate increased. That is the major reason of the increase in the 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 the CPU consumption of the polling I/O. 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 consumed a substantial portion of CPU resource especially when the request rate was relatively low. Herein, we observe an interesting behavior of the polling I/O. Contrary to our intuition, we see that the CPU consumption because of 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 because of the severe CPU starvation especially on the Chamomile I/O thread. Thus, the critical problem is not in the polling I/O itself but in the severe CPU starvation caused by its programming model.

4.3.2 Case with HTTP/1.1 Persistent Connections

When the persistent connections were enabled, the system 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 persistent 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, the CPU consumption was much higher than that in case without persistent connections (Figure 6).

5 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. Because this mechanism does not alter the current I/O multiplexing 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 scalability 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 (10 ms is almost optimal for our benchmark environment) to avoid the increase in the service delay. Also when the number of the sockets was large, the performance was improved largely. However, the response time increased slowly as the request rate grew even with the interval control mechanism.

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 may be set larger than the CPU time slice allocated to each thread by the operating system. To make matters worse, there is no guarantee that a thread can continue its execution during the time slice. In that case, various factors such as 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 the performance improvement by employing some real-time scheduling mechanisms.

References


