Improving Scalability of Processor Utilization
on Heavily-Loaded Servers with Real-Time Scheduling

Eiji Kawai†‡ Youki Kadobayashi† Suguru Yamaguchi†
{eiji-ka,youki-k,suguru}@is.aist-nara.ac.jp
† Nara Institute of Science and Technology
‡ Japan Science and Technology Corporation

ABSTRACT
It is a well-known issue that the polling I/O, such as select() and poll(), has a poor scalability. As a solution of the problem, we previously proposed the interval control technique of the polling I/O, which eases excessively frequent invocations of the polling I/O [1]. Although the benchmark tests proved that the technique effectively reduced the service latency, it still has a problem of high processor usage when the number of concurrent sockets grows large. This problem is caused by unexpected context switches, which degrade the efficacy of the interval control. In this paper, we propose the interval control technique with POSIX real-time scheduling that prevents such unexpected interruptions. Because programs under real-time scheduling are difficult to develop and they often cause a system-wide freeze, simple application of real-time scheduling to the interval control mechanism is considered.

KEY WORDS
polling I/O, interval control, real-time scheduling, select(), poll(), server performance

1 Introduction
On the Unix platform, the polling I/O [2], which is usually implemented by select() and/or poll(), is widely used for network server programs to handle more than one socket concurrently. The polling I/O is an efficient programming model because the polling I/O makes it unnecessary to spawn a thread1 for each connection established at a server. However, the advancement of networking technologies pushes up the network bandwidth and the number of internet users drastically [3], and the polling I/O is now recognized as a performance bottleneck because the number of concurrent sockets that a today’s highly-loaded server has to handle also increases up to thousands or tens of thousands [4].

To solve this issue, several solutions have been proposed so far [4, 5, 6, 7, 8, 9]. Although all the mechanisms presented by them are slightly different from each other, their basic ideas are quite similar. They argued that the problem of the high overhead caused by the polling I/O was

1In this paper, we use the term “thread” to indicate an execution unit including one which is traditionally termed “process”.

in its large socket list scanning, and therefore they developed new event-notification mechanisms between the kernel and the server program, which replace the polling I/O. Their argument is right in a sense because the total processing cost of checking each socket state increases as the number of concurrent sockets grows. However, the real problem of the polling I/O is its excessively frequent invocations caused by its event-driven style to handle concurrent sockets. Such frequent invocations lead to severe starvation of processing power.

From that viewpoint, we previously proposed the interval control technique of the polling I/O to solve the problem of its frequent invocations [1]. This technique is based on the following two procedures, which are depicted in Figure 1 (b). The one is the measurement of the processing time in the thread that executes the polling I/O and the following read/write I/O. In this paper, we term this kind of thread an I/O thread. The other is the fine-grained self-blocking to make the interval of the polling I/O larger than a certain pre-configured threshold.

The major advantage of our mechanism is the high compatibility with the current server programs based on the polling I/O and the operating systems. All the other previous work mentioned before requires large modification in the server program and/or the operating system. Our solution can be easily applied to the traditional server programs using the polling I/O only with addition of a dozen of lines of code. Benchmark tests proved that our technique largely improved the service latency.
On the other hand, the interval control mechanism still has a drawback of high processor usage when the number of concurrent sockets is large. In such situation, the real processing time of the I/O thread grows because it has to scan a large socket list. This incurs the increase of the possibility that the I/O thread receives unexpected context switches, which degrade the accuracy of the measured processing time calculated from the time-stamps taken by the I/O thread itself. This inaccurately measured processing time disables the interval control because a large processing time means that the server is already heavily-loaded and it should avoid self-blocking in such situation.

To solve that issue, we propose an improvement of the interval control by utilizing POSIX real-time (RT) scheduling [10] to avoid such unexpected interruptions of the I/O thread. RT scheduling is easy to program but difficult to debug. Because it often causes a system-wide freeze [11], we have to consider the integration scheme carefully. Simple application of RT scheduling to our interval control mechanism is highly important.

The remainder of this paper is organized as follows. Section 2 describes our previous version of the interval control technique and its problem. Section 3 presents the design and implementation of our new solution with RT scheduling. For the performance evaluation, the benchmark tests were conducted and the results are described in Section 4. We also profiled the operating system during the benchmark tests similar to our previous work [1]. Last, we conclude this paper in Section 5.

## 2 Issue of Interval Control on Polling I/O

This section describes the interval control mechanism on the polling I/O and its issue of the high processor usage.

### 2.1 Interval Control on Polling I/O and Its Implementation

Before we explain the interval control mechanism, we describe its implementation target. Then, we depict the implementation with pseudo source code. We keep the description of the interval control mechanism minimum in this paper, so please refer to our previous work [1] for its details.

#### 2.1.1 Chamomile Web Accelerator

We designed the interval control mechanism of the polling I/O and implemented it in the Chamomile web accelerator\(^2\) [12] for its evaluation. Chamomile has a multi-threaded architecture that consists of listen threads, I/O threads, and retrieve threads as depicted in Figure 2. The listen thread invokes accept() on the listening port, and the newly accepted socket is passed to the I/O thread and managed by the polling I/O concurrently with other sockets. In case an

\(^2\)A web accelerator is also known as a web reverse proxy server.
Figure 3. Pseudo code implementation of the I/O thread with interval control (extracted from [1]). (A)–(D) in the comments correspond to those in Figure 1 (b).

![Diagram](image)

Figure 4. Process flow cycle of the interval control with RT scheduling.

has a problem of high processor usage when the I/O thread has to handle a large number of concurrent sockets.

When the number of concurrent sockets becomes large, also the real processing time of the I/O thread grows. This long processing time increases the possibility that the I/O thread receives unexpected context switches during its execution. Because the self-blocking time of the I/O thread is calculated from the processing time of a cycle in the main loop based on time-stamping, the unexpected interruptions of the I/O thread incurs less self-blocking time. If the measured processing time of the main loop grows larger than the pre-configured threshold, the interval control is disabled since such situation can be generated also when the server is really over-loaded.

To solve this issue, we have two possible ways. The one is to remove such unexpected interruptions and the other is to measure the real processing time of the I/O thread by some other means. However, the latter solution cannot be implemented in most Unix operating systems without a special fine-grained resource accounting mechanism on processor cycle usage, which often requires a special implementation and causes a high overhead. As a consequence, we decided to adopt the former one.

3 Interval Control with Real-Time Scheduling

We designed our new interval control mechanism with RT scheduling. The POSIX RT scheduling is quite simple itself. There are only two scheduling classes: FIFO (first-in first-out) and RR (round robin). Although it is easy to program, it is hard to debug because the threads under RT scheduling can block all the other threads with lower priority forever, which often causes a system-wide freeze. Consider a case that a server host settled in a distant place. If the server program operated on that host under RT scheduling gets troubled by some reason such as a bug in the program, the system can be response-less and we may not be able to even reboot the host remotely. As a consequence, we should keep the utilization of RT scheduling simple enough.

Figure 4 depicts the process flow of our new interval control mechanism. Our purpose to use RT schedul-
1 // set rt scheduling
2 realtime_priority();
3
4 // main loop
5 for (;;) {
6     // (A)
7     // prepare the connection list
8     prepare_conn_list(&list);
9
10     // (B, C, D)
11     // sleep for a while if the loop
12     // interval is too short
13     gettimeofday(&now, NULL);
14     ptime = time_diff(&now, &prev);
15     if (ptime < min_cycle) {
16         #ifdef STATIC_MIN_CYCLE
17             set_time(&stime, 
18                     min_cycle - ptime);
19         #else
20             set_time(&stime, ptime * ratio);
21         #endif
22         // set non-rt scheduling
23         normalize_priority();
24         // self-blocking
25         nanosleep(&stime, NULL);
26         // set rt scheduling
27         realtime_priority();
28     } else {
29         sched_yield();
30     }
31     // (E, F)
32     gettimeofday(&prev, NULL);
33     // process each connection
34     if (!empty(&list))
35         // poll(), read/write I/O
36         proc_conns(&list);
37 }

Figure 5. Pseudo code implementation of the I/O thread with RT scheduling. (A)–(F) in the comments correspond to those in Figure 4.

ing is only to avoid unnecessary interruptions of the I/O thread. In other words, we only make the I/O thread non-preemptive in a sense. Therefore, we set the I/O thread under RT scheduling only during its processing time. In addition, when the processing cycle really grows larger than the threshold, the I/O thread should be released from RT scheduling and yield to some other threads to prevent its continuous execution. The pseudo code implementation is presented in Figure 5.

In this implementation, we designed two kinds of interval control mechanisms: static interval control and dynamic interval control. The static interval control is selected if the macro STATIC_MIN_CYCLE in the pseudo code presented of Figure 5 is defined. Otherwise, the dynamic interval control is selected. The difference between the two mechanisms is just their methods to calculate the self-blocking time. The static interval control sets the threshold of the cycle interval at a static value, which is given by the server configuration, and it calculates the remaining time as the self-blocking time. In this case, if the processing time increases, the self-blocking time decreases inversely. We expect that this achieves constant service delay and linear increase in processor usage. On the other hand, the dynamic interval control sets the self-blocking time proportional to the processing time at configured ratio. Therefore, when the processing time increases, also the self-blocking time increases. We expect that this achieves linear increase in service delay and constant processor usage.

4 Evaluation

For the evaluation of our solution, we conducted benchmark tests with Web Polygraph [14]. Web Polygraph has a pre-defined workload WebAxe-4 [15] for web accelerator. For the details about Web Polygraph and its WebAxe-4 workload, please refer to the web site of Web Polygraph of which URL can be obtained in the reference list of this paper.

The environment configured for the tests was completely the same as the ones which had been prepared for our previous work [1] for fair discussion. As depicted in Figure 6, the client cluster send HTTP requests to the accelerator, and the accelerator processes the HTTP requests as a proxy cache server of the origin server cluster. Table 1 is the list of devices we used in the benchmark tests.

In following subsections, we first discuss the service throughput and the response time. Then, we present the kernel profile during the tests to see how RT scheduling af-
fects the processor utilization of the kernel and user-land threads. Among them, we especially pick up the behavior of `poll()` in the kernel, which is used in Chamomile as the polling I/O function.

### 4.1 Throughput and Response Time

Figure 7 and Figure 8 show the mean response time of the Chamomile web accelerator under Web Polygraph WebAxe-4 benchmark tests. The former is for the static interval control, the latter for the dynamic interval control. The left, center, and right graphs are the distribution of the mean response time of all responses, cache hit responses, and cache miss responses respectively. Herein, we have a point to note. With the static interval control, the `poll()` interval of zero millisecond, which is shown in the graphs of Figure 7, means that the interval control itself is disabled. In that case, Chamomile is operated in a traditional polling I/O model without the interval control, the processing flow of which is depicted in Figure 1 (a).

#### 4.1.1 Case with Static Interval Control

From Figure 7, we first can observe that almost flat distribution of the response time was achieved by our interval control mechanism. From a viewpoint of the scalability, the increase rate in the response time was considerably small. This indicates that the interval control worked effectively and unexpected interruptions of the I/O thread were removed. On the other hand, we have to recall that our expectation here was that the static interval control would achieve constant response time. The source of this slight increase of the response time is the increased processor usage in the operating system by the TCP/IP protocol stack. We will see this phenomenon later in Section 4.2.

Next, we can see that the configured static interval gives large influence on the response time. When we increased the threshold by 10 milliseconds, the response time was increased by about 100 milliseconds. Especially when we set the threshold at 30 or 40 milliseconds, the response time grew too large and such configuration cannot be accepted. This large increase in the response time is caused mainly by the interleaved `poll()` invocation technique implemented in Chamomile. In the benchmark tests, the upper threshold of the total number of the concurrent sockets was fixed at 7,000, which indicates that Chamomile requires at least seven cycles to scan all the sockets.\(^4\)

\(^4\)This count does not include the impact of the sockets closed after HTTP transactions completed and the sockets newly accepted. Therefore, the number of the cycles required to scan all the concurrent sockets can be larger than seven.
4.1.2 Case with Dynamic Interval Control

Similar to the case with the static interval control, also the dynamic control improved the scalability of the service response time as shown in Figure 8. With the dynamic interval control, our expectation that the response time is increased linearly against the request rate was met correctly.

4.2 Kernel Profile Analysis

In this subsection, we argue the kernel behavior during the benchmark tests. For kernel profiling, we used kernprof [16], a kernel profiler released by Silicon Graphics. Similar to the previous subsection, Figure 9 and Figure 10 show the processor usage during the benchmark tests. The left, center, and right graphs depict the kernel processor usage, the total processor usage including user programs, and the ratio of the processor usage of \texttt{poll()} in the kernel to that of the kernel, respectively.

4.2.1 Case with Static Interval Control

From Figure 9 (a) and Figure 9 (b), we can observe the linear increase in the processor usage against the request rate, which means that the scalability in the processor usage, which is the major objective of this research, was achieved.

Compared precisely with the result obtained from our previous work [1], the reduction in total processor usage is largely achieved in the user-land\(^5\), i.e., Chamomile web accelerator itself. Although we can see in Figure 9 (c) that the processing cost of \texttt{poll()} in the kernel was reduced, the reduction in the processing cost of the main loop of Chamomile was still larger. This means that our interval control mechanism worked effectively.

Another point we can observe is that the configuration of the interval threshold of the processing cycle gave influence on the processor usage. Although our previous version of the interval control reduced the response time, it did not produce much effect on the processor usage. On the other hand, these graphs show that our new solution worked correctly.

Last, we have to refer to the decrease of the ratio of the processor usage of \texttt{poll()} to that of the kernel. Comparing the graphs in Figure 9 (a) and Figure 9 (c), we can conclude that the processing cost of \texttt{poll()} is almost constant, and the major source of the increase in the processor utilization by the kernel is the increase in the processing cost of TCP/IP networking.

\(^5\)The processor usage in the user-land can be obtained by subtracting the CPU usage in kernel depicted in Figure 9 (a) from the total CPU usage depicted in Figure 9 (b).
4.2.2 Case with Dynamic Interval Control

The behavior of processor usage of the dynamic interval control depicted in Figure 10 is highly similar to that of the static interval control. Therefore, we can repeat almost the same argument as that in the case of the static interval control.

On the other hand, we have a question here why the processor usage is not constant. We should recall our expectation of constant processor usage by the dynamic interval control. The linear increase in the processor usage is explained by the increase in the processor usage by TCP/IP networking function in the kernel as we mentioned in the previous subsection.

5 Conclusion

To prevent the processor starvation especially when a network server is heavily-loaded with a large number of concurrent sockets, we proposed the interval control mechanism of the polling I/O with RT scheduling. From the benchmark tests, our solution was proved to be effective in such a severe situation.

The major advantage of our solution is the improvement of the scalability, especially from a viewpoint of processor utilization. The electronic power required by a today’s microprocessor is increased day by day for faster processing. To make matters worse, the rapid advancement of network technology makes the scale of server clusters larger and larger. Therefore, the server hosting service providers such as ISPs and data centers are now required to design their power management system with larger output and higher reliability. Our interval control mechanism can give a solution to such kind of power consumption issues from a software viewpoint.

On the other hand, we have to recognize the disadvantage of our solution that there is an overhead of frequent switches of the scheduling policy. That’s the reason why the reduction in the response time is not so large compared with the result in our previous work [1]. However, we can utilize the solution in a situation where the fine-grained control on the processor usage is strongly demanded.

References


