A Scalable Locality-Aware Event Dispatching Mechanism for Network Servers

Hao-Ran Liu and Tien-Fu Chen

Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan 621, ROC

Abstract

Network servers often need to process a large amount of network events asynchronously. They usually use `select()` or `poll()` to retrieve events from file descriptors. However, previous researches have shown that these system calls scale poorly when the number of open connections are significantly increased. Several kernel-level solutions have been proposed. In this paper, we first compare several event dispatching mechanisms available under Linux, and then present our user-level solution that takes advantage of temporal locality among events while polling. We show that a memory-based web server with our approach can have about 20%-30% performance improvement.

Keywords: event dispatching; event polling; scalable servers; network servers

1 Introduction

As the Internet continues to grow in popularity and size, scalability of network servers is more and more important. Network servers without good scalability will result in performance drop or live-lock[12].

Traditional web servers, like early Apache, serve every connection with one dedicated process. This approach simplifies code complexity, but it blocks its processes inside OS kernel when they call system calls like `read()` or `write()`. It is operating system kernel's responsibility to switch contexts of processes to provide concurrent services to all connections. In most operating systems, context switching time could grows in proportion to the number of processes in the system. When there is a large number number of connections, a large portion of CPU time is wasted on context switches[1].

[1] An O(1) scheduler[4] is introduced in Linux 2.5.

```
struct pollfd ev_array[MAX_FD];
int num_event, ev_num, i;

while (TRUE) {
    num_event = poll(ev_array, ev_num, 0);
    if (num_event == 0) {
        do_sleep(msec);
        continue;
    }

    for (i = 0; i < num_event; i++) {
        if (ev_array[i].revents & POLLIN)
            read_handler(ev_array[i].fd);
        else if (ev_array[i].revents & POLLOUT)
            write_handler(ev_array[i].fd);
    }
}
```

Figure 1: Using `poll()` to detect network events

Another approach, called single process event driven (SPED), is to serve all connections in a single process. Server will not block itself waiting for any read or write on a file descriptor, instead, nonblocking I/Os are used. It is inefficient if server calls `read()` or `write()` operations excessively on all of these opened file descriptors to see whether they are ready for read or write. Usually, a system call, like `poll()`, is used to ask kernel which file descriptors are ready. Server then performs read or write operations on those ready file descriptors. The goal of this approach is to reduce context switching and synchronization overhead. Squid, a well-known web proxy cache, is based on SPED architecture. Figure 1 is a simple code demonstrating the use of `poll()`.

Although single process event driven (SPED) architecture is more efficient than multiple process (MP) architecture, previous studies[1, 16] have shown that `poll()` is not scalable; more than 30% of CPU time is spent on such a system call on a normal squid proxy server. The key point is that `poll()` performs the amount of checking overhead in proportion to the number of file descriptors in the event array.

On a regular web server, most connections are idle, because users usually think awhile before they click on next URL and packets may be delivered across a network that suffers from traffic congestion. `poll()` spends most of time on polling useless idle connections. The HTTP 1.1 persistent connection allows multiple requests in a single connection. As more and more web clients support HTTP 1.1 protocol, the methods of persistent connection will increase connection time and make server polling on idle connections performance even worse.

The overhead of handling event detection for all connections severely limits scalability. Most solutions proposed for this problem are based on efforts in the kernel level [1, 16, 8] or even new operating system architectures (such as Novell's Internet Caching System-ICS [18] and Welsh's Staged Event-Driven Architecture[19]). Another approach to obtain performance improvement is to avoid data copy[14, 15] or to reduce the communication

code path by directly interfacing with TCP/IP and overloading those event handlers with HTTP specific processing[7].

In this paper, we focus on improving the performance of web service applications in the user mode. We compare different event dispatching mechanisms in Linux, and give a summary of them. Based on the observation that most web connections are idle, we present a temporal-locality aware library for event dispatching in the user mode. It improves server performance by reducing time spent on `poll()` system call, and provide better code portability for various web applications. We conducted the performance evaluation of the proposed library on a memory-based event-threading server. Our studies show that performance of the server can be significantly improved by 30% or more. Programming interface and implementation details of web server and library are discussed. Performance analysis is given based on two metric: event dispatching overhead and dispatching throughput.

The rest of the paper is organized as follows. Section 2 describes various event dispatching mechanisms in Linux. Section 3 shows our polling strategy, programming interface and how to integrate it into your own code. Section 4 talks about implementation details of the library and important parameters in server code that may influence overall performance. Section 5 evaluates performance of the library and other event dispatching mechanisms. We give conclusion and future work in Section 7.

2 Event Dispatching Mechanisms in Linux

There are many event dispatching mechanisms introduced and being discussed in the literature. Some of them are already incorporated into Linux 2.4 and some are only available in the form of kernel patch. Here we give an overview of these mechanisms and discuss their advantages and disadvantages.

2.1 `poll()` and `select()`

`poll()` and `select()` system calls are state-based event dispatching mechanisms. They report current states of a set of file descriptors specified in arguments. Their implementations are similar with a difference in calling interface, as shown in Figure 2.

`select()` uses a large bitmap `fd_set` to represent the set of file descriptors that are opened for reading, writing or exceptional conditions. Applications have to set the corresponding bits on the three sets for all file descriptors of interest before calling `select()`. On return, kernel overwrites these sets with new values, which are a subset of input sets, telling readiness of each descriptor of interest.

`poll()` uses an array of `pollfd` structure for the same purpose. `events` field, filled by application, indicates events of interest of a file descriptor. `revents` field, filled by kernel, indicates readiness of read, write or exception when the function returns.

When the number of file descriptors is large, `select()` is more suitable since fewer data is

```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);

struct pollfd {
    int fd;
    short events;
    short revents;
}
int poll(struct pollfd *ufds,
         unsigned int nfds;
         int timeout);
```

Figure 2: Prototype of `select()` and `poll()`

copied to and from the kernel. However, both `poll()` and `select()` are not scalable when a server is overloaded with a large set of file descriptors. Because kernel must perform a linear scan check on all descriptors of interest and call device driver's poll callback respectively. In addition, when either `poll()` or `select()` returns, application does another linear scan on return value. Frequent execution of `poll()` or `select()` and the two linear scans make a network server scales poorly when it is overloaded with connections.

2.2 POSIX.4 Real Time Signal

POSIX.4 real time signal[5] (*RT signals*) is an extension to traditional UNIX signal. It allows multiple instances of a signal to be queued by kernel for a process. Each signal delivery carries a `siginfo` payload. Information such as process ID and user ID of signal sender process can be delivered together with a signal.

Linux 2.4 extends POSIX.4 RT signal by allowing delivery of socket readiness via a particular real time signal. Though this feature is Linux specific, it scales pretty well with large set of descriptors. RT signals are event-based event dispatching mechanism. Events are put into a process-specific signal queue just at the time they arrive. This removes the need to call expensive device driver's poll callback functions when server application calls get-event system call. `fcntl()` can associate a file descriptor with an RT signal. Associated RT signal are usually blocked and either `sigwaitinfo()` or `sigtimedwait()` is used to dequeue signals synchronously. Figure 3 illustrates how a RT signal is associated with a file descriptor. Notice that asynchronous I/O must be enabled to allow the occurrence of RT signals. Figure 4 demonstrates how RT signals are used on a single process web server. Signals associated with file descriptors are blocked to prevent signal handlers from being invoked. When `sigwaitinfo()` receives `SIGIO`, it means that the signal queue in the kernel is overflowed and some events are lost. Event loss will probably make some connections deadlock. To avoid this, we need to clean up the signal queue and fallback to traditional `poll()` or `select()` to check all file descriptors. Notice that the program in Figure 4 always calls `poll()` first. This prevents events from being lost when any event comes between

```
// accept a new connection
int sfd = accept(conn, ...);

// associate an RT signal with a connection
fcntl(sfd, F_SETSIG, SIGRTMIN);
// set the process ID to receive the signal
fcntl(sfd, F_SETOWN, getpid());

// Enable nonblocking and asynchronous I/O
fcntl(sfd, F_SETFL, O_NONBLOCK | O_ASYNC);
```

Figure 3: Associating an RT signal with a connection

`accept()` and `fcntl(F_SETSIG, ...)` system calls. When server application accepted a new file descriptor, it needs to poll the descriptor first before receiving RT signals from it.

Chandra and Mosberger introduced *signal-per-fd*[2] to prevent signal queues from overflow. It collapses multiple events of the same file descriptor. If the length of signal queue is equal to the maximum number of file descriptors a process can have, no overflow of signal queue will happen. Although this prevents the signal queue from overflow, the code related to `poll()` is still necessary. Because RT signals do not return an initial state of a file descriptor when a file descriptor is associated with a RT signal. Luban's Linux patch for *signal-per-fd* is available at [17].

One disadvantage of RT signals is the complexity of server code comparing to that of other mechanisms. Yet another one is that only one readiness event can be fetched per `sigwaitinfo()` call. This leads to many switches between user mode and kernel mode. The advantage of RT signals is that it is scalable and users can associate some descriptors with one signal, while some with another. This divides file descriptors into several "interesting sets", allowing server to process them differently.

2.3 `/dev/poll`

`/dev/poll`[9] is first introduced in Solaris 7 in order to remove the need to specify interesting set on every `poll()`. It is a state-based event dispatching mechanism. The idea behind it is that applications can open the device file to build a set of descriptors of interest inside kernel. This set is built gradually after every acceptance of a new connection. The process of building a interesting set is separated from that of event retrieval. This can reduce the amount of interesting set information copied between user space and kernel space. Interesting sets are built by writing `pollfd` structure to an opened `/dev/poll`. Server application use `ioctl()` to fetch events from kernel, which in turn calls device driver's poll callback functions to find the state of registered descriptors. Figure 5 explains how to use `/dev/poll` on a network server. Banga proposed `declare_interest`[1] early in 1999 and `/dev/poll` is the first approximate implementation of the idea.

Provos[16] implements this idea on Linux. The implementation caches latest results from device driver poll callback functions. Given a file descriptor, if there is no event between two

```
struct pollfd ev_array[MAX_FD];
struct timespec wtime = {0, 0};
sigset_t sset; siginfo_t sinfo;
int to_poll = 1, num_event, ev_num, i;

// block SIGIO and SIGRTMIN
sigemptyset(&sset); sigaddset(&sset, SIGIO); sigaddset(&sset, SIGRTMIN);
sigprocmask(SIG_BLOCK, &sset, NULL);

while (TRUE) {
    if (to_poll) {
        num_event = poll(ev_array, ev_num, 0);
        to_poll = 0;

        // call read or write handler if there is any event.
    } else {
        while (sig = sigtimedwait(&sset, &sinfo ,&wtime) > 0) {
            if (sig == SIGRTMIN) {
                if (sinfo.si_band & POLLIN)
                    read_handler(sinfo.si_fd);
                else if (sinfo.si_band & POLLOUT)
                    write_handler(sinfo.si_fd);
            } else if (sig == SIGIO) {
                // signal queue overflow, call poll() at next loop
                to_poll = 1;
                // clean signal queue
                while (sigtimedwait(&sset, &sinfo, &ts) > 0);
            }
        }
    }
}
```

Figure 4: Handling network events with RT signals

```
/* struct dvpoll {
 *     struct pollfd *dp_fds;
 *     int dp_nfds;
 *     int dp_timeout; } */
int i, num_event, dpfd;
struct pollfd fds[MAX_FD];
struct dvpoll dp = {fds, 0, 0};

dpfd = open("/dev/poll", O_RDWR);
while (TRUE) {
    num_event = ioctl(dpfd, DP_POLL, &dp);
    if (num_event == 0) {
        do_sleep(msec);
        continue;
    }

    for (i = 0; i < num_event; i++) {
        if (fds[i].revents & POLLIN)
            read_handler(fds[i].fd);
        else if (fds[i].revents & POLLOUT)
            write_handler(fds[i].fd);
    }
}
```

Figure 5: `/dev/poll` on a network server

`ioctl()` wait event call, previous cache result will be used for the second `ioctl()` and the corresponding device driver poll callback function will not be executed. On the contrary, if there is any event, device driver marks corresponding cache entries dirty. At the next execution of `ioctl()`, dirty descriptors will be polled.

Since Provos's implementation of `/dev/poll` is state-based, there is no problem about event collapsing or event queue overflow. `/dev/poll` also allows multiple interesting sets in Linux kernel. Libenzi[8] also shows an event-based implementation called `/dev/epoll`[1].

2.4 Summary

RT signals are event-based and reported events may not be the latest states of file descriptors. `/dev/poll`, like `select()` and `poll()`, is state-based and reported events are always the states of file descriptors at the polling time. Both RT signals and `/dev/poll` are new scalable event dispatching mechanisms supported by Linux kernel. However, these features are not available on all platforms. RT signals is officially supported since Linux 2.4 and `/dev/poll` is only available with kernel patch. Table 1 summarize those features of all mechanisms discussed above. In Section 5, we will evaluate performance of event dispatching mechanisms available in Linux 2.4.

[1] `/dev/epoll` is incorporated into Linux 2.5.

features methods	Scalable to large set of file descriptors	Event collapsing	Dequeue multiple events per system call	Event queue overflow
`select()`	No	NA	Yes	NA
`poll()`	No	NA	Yes	NA
`/dev/poll`[1]	Yes	NA	Yes	NA
`RT signals`	Yes	No	No	Yes
`RT sig-per-fd`[1]	Yes	Yes	No	No
`declare_interest`[2]	Yes	Yes	Yes	Yes
	Kernel fallbacks to traditional poll() when event queue overflow	Return initial state of FDs on declaration of interesting sets	Maintain multiple interesting sets in kernel for per process	
`select()`	NA	NA	NA	
`poll()`	NA	NA	NA	
`/dev/poll`	NA	NA	Yes	
`RT signals`	No	No	Yes	
`RT sig-per-fd`	NA	No	Yes	
`declare_interest`	Yes	Yes	No	

Table 1: Comparison of various event dispatching mechanisms

3 Proposed Solution: Temporal-locality-aware Poll

Most traffic of a TCP connection has the property of temporal locality. Packets usually arrive in a burst. The definition of temporal locality here is slightly different from that of the traditional way. Traditional definition is about repeated references to the same object in a short time. Here, it is the tendency for a process, once it receives a network packet, to receive a sequence of other packets within a short time. Many researches have shown or exploited the property. Mogul[11] shows the characteristics of persistence and temporal locality at the scale of processes in a LAN environment. Many PCB-lookup algorithms also exploit the property for faster TCP demultiplexing[10].

Temporal locality can be observed at larger scale too. An obvious example is HTTP persistent connection. Usually, A bulk of HTTP requests are sent in a single connection to a web server when browser opens a new page. HTTP requests usually arrive at a web server in a burst during a short period of time.

In the life time of a HTTP connection, it is idle most of the time. Either there is no event for a long time or there are many events in a short time. If a web server is loaded with many connections, most file descriptors in a single `poll()` loop have no event at all. This means that most calls to `poll()` is redundant. No matter how few events there are, the cost spent on `poll()` is the same. This tragedy is caused by the improper design of `poll()` interface. The two linear scans of a long descriptor event array, one inside kernel

[1] Use of these mechanisms needs kernel patch. They are not available in Linux 2.4.

[2] `declare_interest`[1] is listed only for reference. There is no Linux ported version.

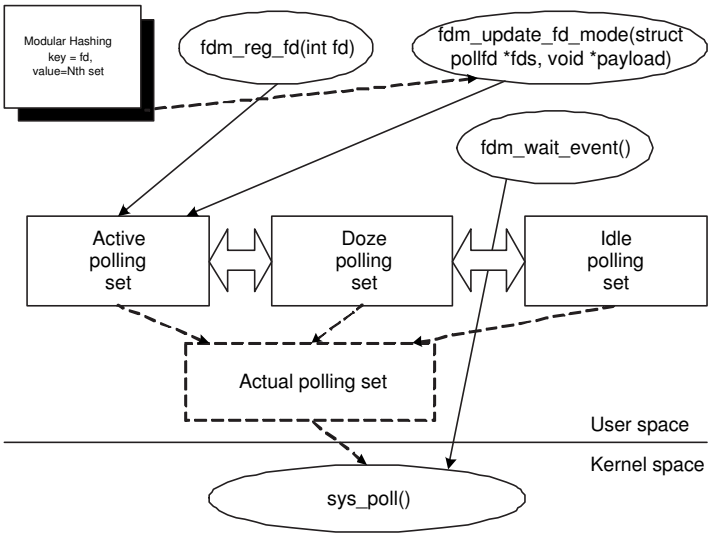


Figure 6: Architecture view of event dispatching library

and one outside kernel, are the main performance bottleneck we want to overcome.

The idea of temporal-locality-aware `poll()` is: since a file descriptor is idle most of the time, it doesn't need to be polled at every poll loop. It should be polled only when it is likely to have events in recent future. Future events of a descriptor can be predicted based on its event history. Instead of estimating event's arrival time, we associate a counter with every descriptor. At each loop of polling, the counter is increased by one if there is a event; otherwise, it is decreased by one. All file descriptors, by default, are divided among three polling sets according to this counter. The frequency of calling `poll()` system call for each set are different. In this way, much time wasted on idle file descriptors is saved and active file descriptors are checked more efficiently.

All server applications, not just web server, featuring the property of temporal locality and a large percentage of idle connections will benefit from the idea. Proxy server and telnet server, for example, are this type of servers. FTP server, on the contrary, does not fit into this type.

3.1 Proposed Library Interface

Our library is a user-mode library, layering between server application and kernel system call. We named it FDM (File Descriptor Management Library), because its goal is to manage all file descriptors in a server application. During the life time of file descriptors, file descriptors are kept in one of the three polling sets in the library, and are polled by the library according to its live counter. Figure 6 illustrates the architecture of the library. Figure 7 shows the function prototypes of the library.

In the library, we have separate functions for specifying file descriptors of interest and for

```
typedef struct {
   int fd;
   short events;
   void *payload;
} fdm_event_t;    // data structure return by wait event
extern int fdm_setnum;
extern int fdm_default_live_counter;

int fdm_start();  // library init
int fdm_stop();   // library shutdown
int fdm_reg_fd(const struct pollfd *fds, int lock);
int fdm_unreg_fd(int fd);

// tell library if a fd is read interest or write interest
int fdm_update_fd_mode(const struct pollfd *fds,
                       void *payload,
                       struct timeval *tv);
int fdm_wait_event(fdm_event_t *ev_array,
                   unsigned int array_size,
                   int timeout);
```

Figure 7: Interface of event dispatching Library

retrieving events. Server may register a file descriptor with `fdm_reg_fd()` once it is created by `accept()`. Initially, `fdm_reg_fd()` puts the descriptor specified in first argument into most frequently polling set. All registered file descriptors will migrate between three polling sets depending on their respective live counters. The second argument specifies whether or not to lock the descriptor specified in the first argument from migrating between polling sets. If a file descriptor is locked, it stays in most frequently polling set regardless of the value of its live counter. Listening sockets are usually being locked to achieve better performance. This concept is similar to *multi-accept*[2]. Because all connections and events of connections are derived from `accept()` of listening sockets, limiting the polling frequency of listening sockets will limit the performance of server. Besides, listening sockets have little temporal locality. Polling frequency of each polling set is discussed in next section.

`fdm_wait_event()` is used to receive network events. It creates a new polling set from the three polling sets and invokes `poll()` on the new set. `array_size` specifies the size of memory space pointed by `ev_array`. `ev_array` stores retrieved events when the function returns. `timeout` specifies the time to wait before function returns if there is no event. Internally, this parameter is passed directly to `poll()` system call.

Given a file descriptor, server application may be interested in either reading or writing events, but not both, at the same time. It waits for either a read ready event before reading data or a write ready event before writing output. Server application must specify current interest in either read or write of a file descriptor before calling `fdm_wait_event()`. This is done through `fdm_update_fd_mode()`. Current interest of a file descriptor are specified in `events` field of `fds` argument.

In general case, server application may need to maintain an array or a hash table to keep

track of the mapping between a file descriptor and its threading or callback data structure (these data structure are needed in a SPED server). When application retrieves some events from kernel, file descriptors associated with these events are searched throughout the mapping array or hash table for their threading or callback data structures. Server needs a threading (or callback) data structure to resume the execution of a thread (callback function) that deals with a specific connection.

To remove the need for the server application to maintain another mapping data structure mentioned above, the data structure for file descriptor maintained in this library has an additional payload field. It saves the address of associated threading or callback data structure. The data structure for events returned by `fdm_wait_event()` also carries a payload field. `payload` is a void pointer that can point to anything like threading or callback data structure.

In addition, server applications may need to close a timeout connection if it is idle for a long period. The feature support in this library can remove another time-stamp array (also the corresponding linear search) in server code. Both of the threading (callback) data structure and the timeout time-stamp of a file descriptor can be updated through `fdm_update_fd_mode()`.

`fdm_unreg_fd()` removes a file descriptor from the polling sets of the library. This function is usually called when the associated connection is closed. `fdm_setnum` and `fdm_default_live_counter` are two parameters that gives users more flexibility to select how many polling sets to create and to decide the polling frequency of each polling set besides default setting.

4 Implementation Details

4.1 Polling Frequency

Polling frequency will influence the response time of a connection and overall throughput of the web server. `poll()` is invoked on every `fdm_wait_event()` call; however, not all three polling sets are polled. It depends on the value of default live counter. Assuming that default live counter is N, then active polling set is polled on every `fdm_wait_event()` call, doze polling set is polled on every N calls and idle polling set is polled on every N^2 calls. A file descriptor is downgraded from active to doze or doze to idle if there is no event for $N-1$ or N^2-N calls, and is upgraded from idle to active or doze to active if events are detected twice in two consecutive calls.

A file descriptor is initially put into active polling set once registered. If it has no event for $N-1$ `fdm_wait_event()` calls, it falls into the category of second polling set, because second polling set assumes file descriptors inside it have events every N `fdm_wait_event()` calls. The same logic holds true for descriptors downgraded from second polling set to the third one.

Since `poll()` has a state-based view of events, the time when a event occurs is unknown.

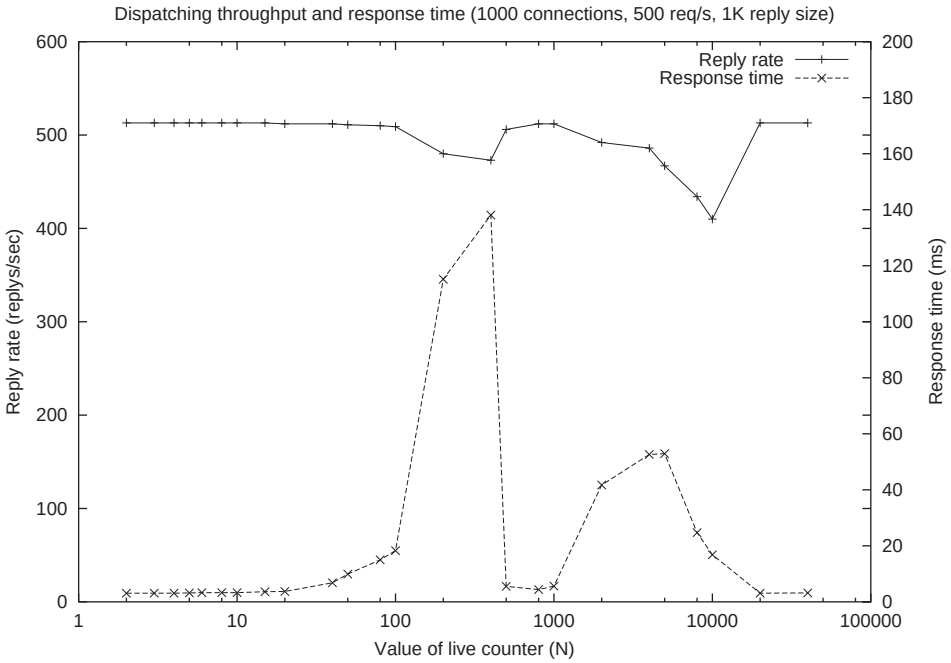


Figure 8: Performance with different value of default live counter

There is no way to determine the frequency of events of a file descriptor exactly; so the best way is to poll them on different frequency and let all file descriptors fall into the best categories by themselves. Although a more frequently polling can be used to better predict frequency of events, this approach cannot be used since it conflicts with our goal to reduce the polling time of idle file descriptors.

When a descriptor has two events in two consecutive `poll()` calls, we upgrade it to active polling set immediately. This is because of the property of temporal locality of network events. Other events for the same descriptor may arrive very soon. Our upgrade strategy keeps a fairly well response time for server application using this library.

In our experience, when web server is overloaded, we found out that a small value of N maintains a good response time yet good improvement on throughput. A medium value of N further improves throughput but the response time is longer. A large value of N renders locality-aware `poll()` useless, because most file descriptors will not downgrade and stay at active polling set. This results in long, large standard deviation of response time and worst throughput (some downgraded connections will suffer unacceptable response time). Our experience shows that a feasible value of N is between 3 and 10.

Figure 8 shows the influence of default live counter on reply rate and response time of a memory-based web server. In this experiment, we establish 1000 persistent connections to the server. Each connection sends a request every 2 seconds, so that total request rate is fixed at 500 requests per second. This simulates 1000 users browsing the web with 2 seconds think time. N influences the response time of requests significantly. We take advantage

of both, saving CPU time on polling more active descriptors and good response time, only at small value of N. With large value of N, temporal-locality-aware `poll()` approximates to plain `poll()` because file descriptors are not differentiated. Notice that response time and throughput doesn't always increase or decrease with the increase of N. The reason is: When $N < 400$, most file descriptors fall into idle polling set and increasing N may also decrease the polling frequency of idle set. This limits the number of events a server can have and the reply rate drops too. However, when N is increased from 400 to 800, more and more descriptors in idle polling set are moving to doze polling set along with the increase of N; i.e. the number of descriptors in doze polling set is increased and polling frequency is increased from every 400^2 wait event calls to every 800 calls. So reply rate increases again. At $N > 800$, most descriptors fall into doze polling set; further increase of N decreases polling frequency and thus limits the number of events, making performance worse.

4.2 Multiple Threads Implementation

Gooch[6] mentioned the idea of dividing file descriptors among two threads. One thread polls mostly active file descriptors, another one polls the rest. However, multiple threads implementation is impractical. There are two reason. First, the overhead of synchronization and context-switching between threads will limit the performance of a server. Second, `poll()` is invoked by all threads asynchronously. Polling frequency of each threads are determined by their respective sleep time, which is regulated by 10ms Linux timer. However, 10ms is so long that server is sleeping most of the time. This results in a long response time and poor throughput on the server.

In fact, we do have a multi-thread implementation of our library. It has worse performance than a simply `poll()`. Even with a multiple threads implementation, polling frequencies of three polling sets are still under exponential relationship, limited by the same reason that `poll()` is state-based and event arrival time is unknown. It is not beneficial to server applications.

4.3 Web Server Sleep Threshold

Web server sleep threshold is the number of consecutive wait event calls without any event before web server sleeps. If the number of this kind of wait event calls reaches the threshold, web server is put into sleep.

When there are too few events to keep the server busy, put the server into sleep is often desired. However, as just mentioned, granularity of Linux timer is too coarse. If server is always put into sleep when there is no event. Server may sleep too much and have a lower throughput.

Sleep threshold is machine dependent, because a fast machine consumes events more quickly and is more likely to reach the sleep threshold than a slow one. Fast machine needs to have larger sleep threshold to prevent it from limiting the processing power. Sleep threshold is also event dispatching mechanism dependent. A scalable, fast event delivery

mechanism like `/dev/poll` or RT signals needs to sleep less frequently too, for the same reason.

Notice that the value of default live counter must be considered together with web server sleep threshold. Larger value of default live counter generally decreases the number of file descriptors to be polled, making event delivery faster too. In conclusion, a high performance web server requires fine tunes of sleep threshold.

5 Performance Evaluation

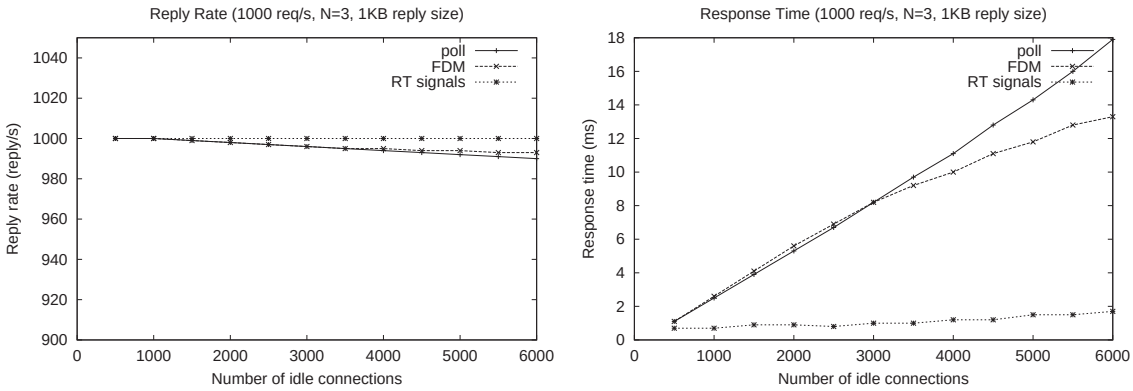
This section shows the performance of memory-based web server on event dispatching mechanisms available in standard Linux 2.4, including our user-mode solution. We describe evaluation environment and the implementation of our test web server. Server performance is evaluated on two metrics: dispatching overhead and dispatching throughput.

5.1 Evaluation Environment

To test various event dispatching mechanisms on a event-driven web servers, we modify *dphttpd*[8] to take advantage of our event-driven threading architecture. *dphttpd* is a very simple memory-based web server which does very simple HTTP protocol processing. It allows us to focus on performance of various event dispatching mechanisms without other constraints like disk I/O. Event-threading architecture provides both the advantage of easy programming and event-driven architecture to network servers application. It minimizes synchronization and kernel context-switching overhead by associating every connection with a user level thread. Context switches between user level threads happen only when a user level thread explicitly gives up control by waiting for I/O completeness or calling schedule yield function.

On our modified memory-based event-driven web server, three event dispatching mechanisms: `poll()`, temporal locality-aware poll and RT signals, all of them are available in standard Linux 2.4 without patch, are implemented. Many Parameters on the web server are optimized separately for each event dispatching mechanisms. Server sleep threshold is fine tuned for maximum performance. Our server does multiple `accept()` on a listening socket descriptor when `poll()` reports a ready event on such a descriptor. This strategy minimizes the effect of `poll()` overhead. Because more connections are accepted and more useful works will be identified by follow-up `poll()`. The overhead of `poll()` are amortized(*multi-accept*[2]).

In the experiment, server runs on a Pentium III 600MHZ, 128MB RAM machine. Client loads are generated from three Pentium III 1GHZ, 512MB RAM machine with httperf[13]. Idle connections are made from another low-end machine. All machines are equipped with a Intel EtherExpress Pro 100 ethernet card, and connected together within a single 100Mb ethernet LAN.



(a) Reply rate

(b) Response time

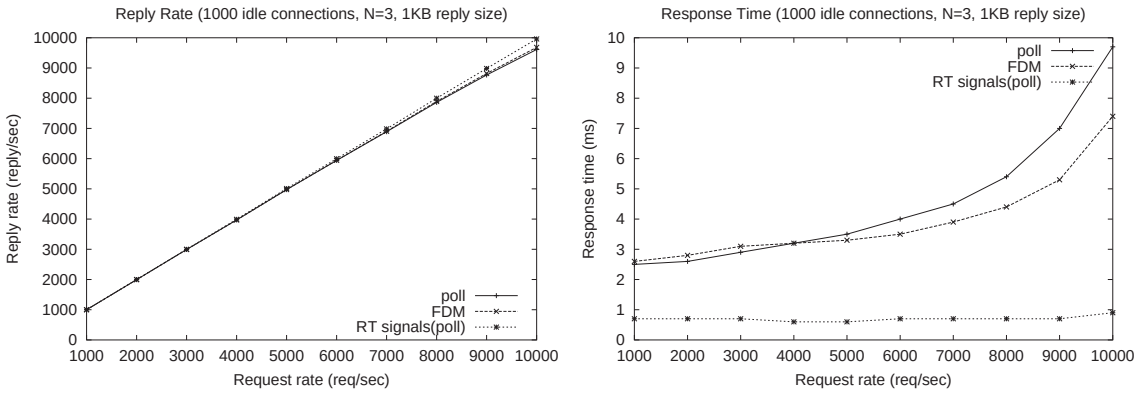
Figure 9: Event dispatching overhead in terms of number of idle connections

5.2 Dispatching Overhead

Dispatching overhead experiment simulates the condition when a web server is loaded with large number of slow, long distance, idle connections. To see the influence of these connections, web server is not overloaded. Request rates are fixed at comparatively light load. We measure the overhead of request handling as a function of number of idle connections.

Figure 9 shows the influence of such overhead on web server reply rate and response time. RT signals mechanism provides a stable reply rate and very short response time. Although RT signals mechanism is scalable, idle connections still incurs small overhead on RT signals (Figure 9(b)). Because web server has to maintain an array of timeout stamp and `pollfd` data structure on every connection. On a regular web server, idle connections are timeout and disconnected when there is no activity for a period of time. To show the effect of overhead maintaining these arrays, code maintaining `pollfd` array and scanning through time-stamp array for timeout connections remains in our test web server, but timeout connections are not disconnected.

When the number of idle connections is less than 3000, FDM (temporal-locality-aware poll) has equal reply rate comparing to plain `poll()`, but a slightly longer response time. The reason is that the polling time (money) saved by FDM is less than the time (cost) spent on polling set management. Notice that the response time on a plain `poll()` system grows linearly with number of idle connections. When the number of idle connections is greater than 3000, a large portion of time wasted in a plain `poll()` system can be saved in a event-locality-aware system. The time saved is greater than the time spent on polling set management and is invested in active file descriptors with more frequently `fdm_wait_event()` calls. This results in shorter response time for FDM comparing to the response time for plain `poll()`.



(a) Reply rate (b) Response time

Figure 10: Server performance with 1000 idle connections

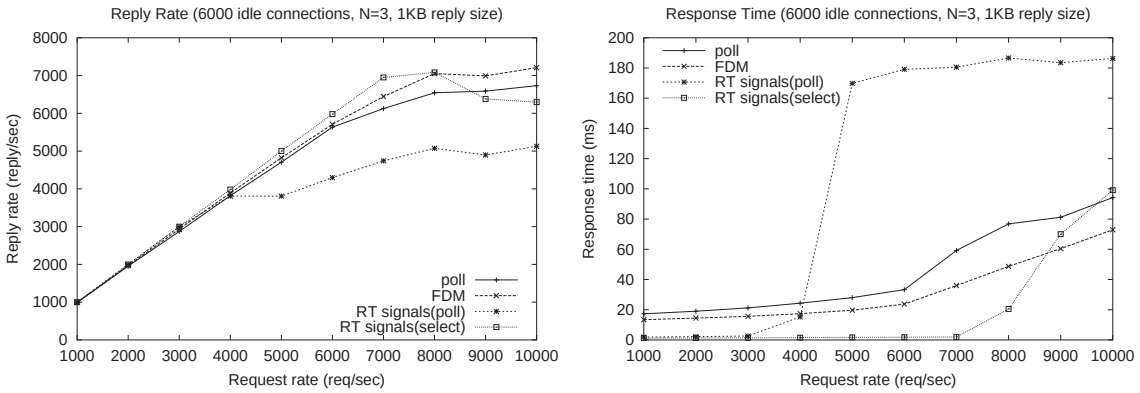
5.3 Dispatching Throughput

In dispatching throughput experiment, we want to see how much throughput a web server can achieve when the server is given a fixed number of idle connections. The performance is measured as a function of request rates. RT signals with `poll()` and RT signals with `select()` are evaluated respectively to see the problem of `poll()` when it is used to recover the server from RT signal queue overflow.

Figure 10 shows server performance with 1000 idle connections. Notice that all mechanisms scale pretty well with respect to the reply rate (Figure 10(a)). Request rate beyond 10000 requests per second is not feasible since throughput is limited by 100Mbit ethernet. Because we implement *multi-accept* on plain `poll()` test server, the reply rate of plain `poll()` is comparatively good. However, `poll()` is inherently not scalable to a large number of connections. Time spent on event detection is too long to maintain good response time when request rate increase. In Figure 10(b), you can see the response time of plain `poll()` increases with request rate. We also observe that the response time of `poll()` is a lot longer than that of RT signals even at light load, since overhead of 1000 idle connections is there for `poll()` no matter how request rate is. Under unscalable `poll()`, FDM improves response time with different polling strategy, but it can't change the fact that underlying mechanism costs time in proportion to number of connections. So response time of FDM increases with the increase of request rate.

To further observe the behavior of FDM library and RT signals, In Figure 11, we increase the number of idle connections to 6000. The advantage of FDM is clear and obvious. FDM-based web server delivers highest throughput and shortest response time.

However, it is surprising that the performance of RT signals(poll) falls behind even that of traditional `poll()` when request rate are greater than 4000 req/sec. The reason is because we use `poll()` to handle lost event when the signal queue is overflowed. To



(a) Reply rate (b) Response time

Figure 11: Server performance with 6000 idle connections

use `poll()` with RT signals, web server has to prepare and maintain a `pollfd` array for such signal queue overflow emergency. Rearranging this `pollfd` array on every several `sigtimedwait()` calls and the timeout time-stamp array is costly and costs CPU time in proportion to the number of idle connections. At 6000 idle connections, RT signals web server spends much more time on the work we just mentioned than the time spent at 1000 idle connections. When client request rate is greater than 4000 req/sec, signal arrival rate become faster than signal dequeue rate. This results in many `SIGIO` signal queue overflow events and significantly increase of response time. This scenario of performance drop can't be observed when server is loaded with just 1000 idle connection.

Though, RT signals web server can be improved by replacing `poll()` with `select()` to eliminate the need to maintain a `pollfd` array. `select()`'s bitmap parameters allow direct access to any file descriptor's event fields. Thus, linear scan and maintenance of `pollfd` array are eliminated. The overhead of finding timeout connections can also be alleviated by reducing times of scan on timeout time-stamp array.

Figure 11 also shows the performance improvement of RT signals(select) over RT signals(poll). The reply rate and the response time of RT signals(select) is the best over all event dispatching mechanisms when request rate is smaller than 8000 req/sec. When request rate is greater than 8000 req/sec, excessively switching back and forth between RT signals and `select()` when RT signal queue is overflowed makes the performance drop. The performance of FDM also starts overtaking RT signals (select) at 8000 req/sec. RT signal queue overflow problem greatly influences the performance of both RT signals(poll) and RT signals(select) at different load.

6 Future Work

In our library, the responsiveness and performance of a server are controlled together by default live counter and sleep threshold, but responsiveness and performance conflicts and cannot be achieved at the same time. If there is an adaptive solution which can automatically pick the most feasible default live counter and sleep threshold, users would not have to worry about tuning these parameters. However, choosing these parameters depends so heavily on CPU processing power, the version of operating system kernel, the type of your service and workload characteristics that we have not found a simple and practical solution yet. Typically, System profiling and workload trace are generally required.

GNU Portable Threads[3] is a user-space thread library with an interface combining thread management and event handling. Different type of events can be placed in a data structure called event rings, and threads can block on these event rings. It frees programmers from managing another thread library, but it is not temporal-locality-aware. We manages to incorporate the idea of temporal locality into this library in the future.

7 Conclusion

In this paper, we summarized the advantages and disadvantages of various event dispatching mechanisms. `select()` and `poll()` suffers from poor scalability and long response time when the number of connections is large. `/dev/poll` and RT signals are two scalable mechanisms available in Linux 2.4. `/dev/poll` caches latest poll result and do not poll a file descriptor again if there is no state change. RT signals mechanism scales well but has signal queue overflow problem. Our proposed user-mode library solution extends `poll()` by exploiting temporal locality property among events in a file descriptor. This approach provides good code portability and reduces total number of descriptors to be polled and more CPU time is saved for useful work.

It is surprising that the performance of RT signals web server is not scalable when it is loaded with 6000 idle connections. This is because of the excessively switching between the two mechanisms, `poll()`(or `select()`) and RT signals, when RT signal queue is overflowed. RT signals(poll) server has worse performance than RT signals(select) one. The overhead of maintaining `pollfd` array associated with `poll()` is very large and will slow down event processing speed. We concluded that `select()` should be used to protect RT signals when signal queue is overflowed.

Availability

The temporal-locality-aware poll library and related program mentioned in this paper is available at http://www.cs.ccu.edu.tw/~lhr89/fdm/.

References

[1] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[2] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *USENIX Annual Technical Conference*, 2001.

[3] Ralf S. Engelschall. GNU Portable Threads. http://www.gnu.org/software/pth/.

[4] O(1) Scheduler for Linux. http://people.redhat.com/mingo/O(1)-scheduler/.

[5] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, 1995.

[6] Richard Gooch. I/O event handling under Linux. http://www.atnf.csiro.au/people/rgooch/linux/docs/io-events.html.

[7] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey. High performance memory based web caches: Kernel and user space performance. In *USENIX Annual Technical Conference*, 2001.

[8] Davide Libenzi. Improving (network) I/O performance. http://www.xmailserver.org/linux-patches/nio-improve.html.

[9] Solaris 8 man pages for poll(7d). http://docs.sun.com/?q=%2fdev%2fpoll&p=/doc/816-3330/6m9kamh9d&a=view.

[10] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM'92*, pages 269–279, 1992.

[11] Jeffrey C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[12] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[13] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[14] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference*, 1999.

[15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[16] Niels Provos and Chuck Lever. Scalable network I/O in Linux. In *USENIX Annual Technical Conference, FREENIX Track*, 2000.

[17] Signal-per-fd patch for Linux. http://www.luban.org/GPL/gpl.html.

[18] Gary Tomlinson, Drew Major, and Ron Lee. High-capacity Internet middleware: Internet cache system architectural overview. In *Proceedings of the Workshop on Internet Server Performance (WISP99)*, 1999.

[19] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.