Lazy Asynchronous I/O For Event-Driven Servers

Anupam Chanda, Alan L. Cox, and Khaled Elmeleegy
Rice University, Houston, Texas-77005, USA

Willy Zwaenepoel
EPFL, Lausanne, Switzerland

Abstract

In this paper, we introduce *Lazy Asynchronous I/O* (LAIO), a new API for performing I/O that is well-suited but not limited to the needs of high-performance, event-driven servers. In addition, we describe and evaluate an implementation of LAIO that demonstrably addresses certain critical limitations of the asynchronous and non-blocking I/O support in present Unix-like systems. LAIO is implemented entirely at user-level, without modification to the operating system's kernel. It utilizes scheduler activations. Using a micro-benchmark, LAIO was shown to be more than 3 times faster than AIO when the data was already available in memory. It also had a comparable performance to AIO when actual I/O needed to be made. An event driven web server (thttpd) achieved more than 38% increase in its throughput using LAIO. The Flash web server's throughput, originally achieved with kernel modifications, was matched using LAIO without making kernel modifications.

1 Introduction

In this paper, we introduce *Lazy Asynchronous I/O* (LAIO), a new API for performing I/O that is well-suited but not limited to the needs of high-performance, event-driven servers. In addition, we describe and evaluate an implementation of LAIO that demonstrably addresses certain critical limitations of the asynchronous and non-blocking I/O support in present Unix-like systems.

In general, to achieve the highest level of performance, an event-driven server must avoid blocking on any operation, from I/O to resource allocation. Thus, in event-driven servers, the use of asynchronous or non-blocking I/O is for all practical purposes mandatory. The problem is that the asynchronous and non-blocking I/O support in present Unix-like systems is limited by its lack of general applicability. For example, non-blocking I/O can be performed on network connections, but not disk files. In contrast, POSIX asynchronous I/O (AIO) can be performed on disk files, but only supports reading and writing; many widely-used operations that require disk access as a part of their implementation, such as opening a file or determining its size, are not supported.

In principle, this problem could be addressed by changes to the operating system. Such changes would affect the operating system's interface as well as its implementation. In practice, the scope of such changes has effectively impeded such a solution. As a consequence, developers faced with this problem have either (1) abandoned an event-driven architecture entirely for a multithreaded or multiprocess architecture, (2) accepted that some operations can block and the effect thereof on performance, or (3) simulated asynchronous I/O at user-level by submitting blocking operations to a queue that is serviced by a pool of threads.

The first of these options has received considerable attention in the literature. The consensus conclusion being that an event-driven architecture has potential performance advantages over a multithreaded or a multiprocess architecture [3, 4, 11, 2]. These advantages include greater control over scheduling, lower overhead for maintaining state, and lower overhead for synchronization. Recently, von Behren *et al.* [10] have, however, argued that a better designed threading library can enable the multithreaded architecture to achieve performance comparable to the event-driven architecture. For our part, we only observe that performance is not a reason to switch. Thus, we do not give this option further consideration.

Surprisingly, the second option does appear in practice. The `thttpd` web server is a notable example.

The *asymmetric multiprocess event driven* (AMPED) architecture that was employed by the Flash web server is representative of the third category [4]. The essence of its solution was a hybrid architecture that consisted of an event-driven core augmented by helper threads. Flash performs all non-blocking operations in an event-driven fashion and all potentially blocking operations are dispatched to helper threads.

The LAIO interface consists of three simple, easy to use functions. The essence of LAIO is to perform asynchronous I/O lazily. If the I/O operation completes without blocking then to the program it looks exactly like an ordinary synchronous I/O operation. In fact, there is no significant overhead. If, however, the operation would block, the program receives an indication, EINPROGRESS, that the operation is being performed asynchronously. On completion of the operation, a notice is delivered to the program.

LAIO is implemented entirely at user-level, without modification to the operating system's kernel. It only requires support for scheduler activations [1]. We use scheduler activations to deliver upcalls to the LAIO library when a process blocks and later unblocks in the kernel. In effect, these upcalls provide a method for converting arbitrary, potentially blocking I/O operations into asynchronous operations. Coupled with stack switching, these upcalls allow us to continue the execution of the application even if the process blocks. Stack switching is achieved by saving the context of the executing thread before it blocks, and jumping to the saved context on receiving an upcall when the thread blocks.

The contributions of this paper are three-fold.

First, the LAIO API is simpler to use than non-blocking I/O, and we illustrate this in the paper. To demonstrate the usability of LAIO we have integrated it with *libevent*, an event notification library [6]. We have also augmented the thttpd web server [5], which is an event driven web server, and the Flash web server [4] to use the LAIO API.

Second, the cost of LAIO is small compared to non-blocking I/O or POSIX asynchronous I/O. We constructed microbenchmarks and found that LAIO is only a little more expensive than non-blocking I/O. Compared to POSIX asynchronous I/O, LAIO is three times faster when the operation runs to completion without blocking. Performance is comparable when the operation blocks and has to be performed asynchronously.

Third, LAIO achieves better or comparable performance when integrated with existing event-driven servers. Using LAIO, thttpd attained 38% more throughput. Flash integrated with LAIO matched the performance of Flash which used kernel modifications to prevent blocking of the server.

The remainder of this paper is organized as follows. Section 2 provides background on asynchronous and non-blocking I/O, event-driven programming, and scheduler activations. Section 3 describes the LAIO API and its implementation; the integration with libevent is also described. Section 5 describes our modifications to the thttpd and Flash web servers. Section 6 describes our methodology. Section 7 describes our experiments. Section 9 concludes this paper.

2 Background

In this section, we provide background for the remainder of this paper. First, we compare and contrast asynchronous with non-blocking I/O. Then, we describe how an event-driven server uses asynchronous and non-blocking I/O.

2.1 Asynchronous I/O Versus Non-Blocking I/O

We present two examples of writing data to a network socket, the first using asynchronous I/O and the second using non-blocking I/O. That we use a network socket, rather than a file, reflects an important limitation of non-blocking I/O: In present Unix-like systems, non-blocking I/O only applies to sockets, pipes, and terminals.

For asynchronous I/O, we show the use of the POSIX Asynchronous I/O (AIO) API [9]. Under this API, there are four steps to performing an I/O. Figure 1 illustrates these steps. First, a control block describing the operation is initialized. This control block includes information such as the descriptor on which the operation is performed, the location of the buffer, and its size. Then, the operation is initiated. At this point, the application can engage in other activities. The control block is still, however, used as a handle to identify the unfinished operation. In most implementations, the application determines that the

operation has finished through polling or an asynchronous event notification mechanism, such as signals. For the sake of simplicity, Figure 1 illustrates polling. The same operation, `aio_error()`, is used both to poll for completion and to learn of any error that occurred during the operation. Finally, the return value of the operation is obtained, using `aio_return()`.

```
/* Step 1: initialize the control block */
aiocb.aio_fildes = socket;
aiocb.aio_buf = buffer;
aiocb.aio_nbytes = bytes_to_write;
...
/* Step 2: initiate the operation; returns immediately */
aio_write(&aiocb);
do {
    ...
    /* perform other activities */
    ...
    /* Step 3: poll for completion or fatal error */
    error = aio_error(&aiocb);
} while (error == EINPROGRESS);
if (error != 0) {
    /* handle fatal error */
}
/* Step 4: obtain the return value, in this case,
 * the number of bytes written */
return_value = aio_return(&aiocb);
```

Figure 1: An Example of Asynchronous I/O

From our standpoint, the principle limitation of the POSIX AIO API is the small set of supported operations. Only the basic operations, reading and writing, are supported. In other words, complex operations that include I/O as part of their implementation, such as opening a file or determining its size, are not supported.

In contrast, for non-blocking I/O, the number of steps to perform an I/O operation varies. The reason being that a non-blocking I/O operation may require several restarts before it completes. Moreover, the application, instead of the operating system, is responsible for maintaining the I/O operation's state of progress. Figure 2 illustrates both the steps and the state maintenance. First, the socket is configured for non-blocking I/O. This step is only performed once per socket. Second, any variables for maintaining the I/O operation's state are initialized. Third, the I/O operation is attempted. Discounting errors, there are two possible outcomes to this attempt, the operation completes or must continue. If the operation must continue, its state is updated to reflect any progress. At this point, the application can engage in other activities. Finally, the application must determine when the socket is ready for writing more of the buffer. The most common methods use `poll()`, `select()`, or `kevent()`. When the socket is ready, the application returns to the third step.

2.2 Event-Driven Programming With I/O

A request on an event driven server is handled like a finite state machine. For example the connection is first received, and then the request is put in a state waiting to read the request string from the socket. When data is available on the socket, the request is read and the request is put in a state where it's ready to write the response back to the socket. After the socket gets ready for writing, the response gets written back to the socket and now the request has been served and it reaches its final state, completion.

An event driven server has an event loop which is an infinite loop that receives event notifications, for example a socket ready for read or write. For each of those received events, the event loop invokes the

```
/* Step 1: enable non-blocking I/O */
fcntl(socket, F_SETFL, O_NONBLOCK);
...
/* Step 2: initialize the state of progress */
bytes_written = 0;
bytes_remaining = bytes_to_write;
for (;;) {
    /* Step 3: attempt the operation */
    return_value = write(socket, &buffer[bytes_written], bytes_remaining);
    if (return_value == bytes_remaining) {
        /* completed */
        break;
    } else if (return_value > 0) { /* and implicitly less than bytes_remaining */
        /* Step 4: update the state of progress */
        bytes_written += return_value;
        bytes_remaining -= return_value;
    } else if (return_value == -1 && errno != EAGAIN) {
        /* handle fatal error */
    }
    do {
        ...
        /* perform other activities */
        ...
        /* Step 5: use an API, such as poll(), select(), or kevent(), to
         * determine if the socket is ready for writing more of the buffer */
    } while (/* the socket is not ready */);
}
```

Figure 2: An Example of Non-Blocking I/O

corresponding event handler for that event. Figure 3 illustrates an event handler performing a non-blocking operation.

Event handlers run to completion and are never preempted. Consequently it is sufficient to guarantee that the shared state is consistent before returning to the event loop. In our example this would be the socket ready for write. The event handler unregisters interest in the event it is handling, then it continues execution until reaches a potentially blocking operation. It executes this operation, if the operation runs to completion the event handler continues execution. Otherwise like in the example above if not all the data could be written to the socket, the handler registers interest in doing that operation when available. Instructing the event loop to check for notifications for this newly registered event, in our example that would be the socket being ready for writing; Then the handler returns to the event loop after making sure it's leaving the server in a consistent state.

3 LAIO

In this section, we first present the LAIO API and then describe its implementation.

3.1 Interface

The API for LAIO consists of three functions: `laio_syscall()`, `laio_gethandle()`, and `laio_poll()`.

`laio_syscall()` has the same signature as `syscall()`, a standard function for performing indirect system calls. The first parameter identifies the desired system call. Symbolic names for this parameter, representing all system calls, are defined in a standard header file. The rest of the parameters vary according to the

```
void
client_response_write(..., void *arg)
{
    struct client *client = arg;

    /* Step 0: assume that the one-time operations, enabling non-blocking I/O
     * and initializing the state of progress, have been performed elsewhere. */
    ...
    /* Step 1: attempt the operation */
    return_value = write(client->socket,
                         &client->buffer[client->bytes_written],
                         client->bytes_remaining);
    if (return_value == client->bytes_remaining) {
        /* tell the event loop that the operation has completed */
        event_del(&client->event);
        client_response_epilogue(client, ...);
        return; /* to the event loop */
    } else if (return_value > 0) { /* and implicitly less than bytes_remaining */
        /* Step 2: update the state of progress */
        client->bytes_written += return_value;
        client->bytes_remaining -= return_value;
    } else if (return_value == -1 && errno != EAGAIN) {
        /* handle fatal error; tell the event loop not to continue */
        event_del(&client->event);
        client_response_error(client, ...);
        return; /* to the event loop */
    }
    ...
    /* perform other activities */
    ...
    /* assume that this function remains registered with the event loop so
     * that it will be called when the socket is again ready for writing */
    return; /* to the event loop */
}
```

Figure 3: An Example of an Event Handler Performing Non-Blocking I/O

expectations of the desired system call. If the desired system call is able to complete without blocking, the behavior of `laio_syscall()` is indistinguishable from that of `syscall()`. If, however, the system call is unable to complete without blocking, `laio_syscall()` returns -1, setting the global variable `errno` to `EINPROGRESS`. Henceforth, we refer to this case as "a background `laio_syscall()`."

`laio_gethandle()` returns an opaque handle for the purpose of identifying a background `laio_syscall()`. Specifically, this handle identifies the most recent `laio_syscall()` that reported `EINPROGRESS`[1]. If, however, the most recent `laio_syscall()` completed without blocking, `laio_gethandle()` returns `NULL`. In other words, `laio_gethandle()` is expected to appear shortly after a background `laio_syscall()`.

`laio_poll()` returns a set of structures. Each structure represents the completion of a background `laio_syscall()`. A structure consists of a handle, a return value, and a possible error code. The handle identifies a background `laio_syscall()`. The return value and possible error code are determined by the particular system call that was performed by the background `laio_syscall()`.

[1] In a multithreaded environment, the intended meaning of "most recent `laio_syscall()`" is the most recent `laio_syscall()` that was performed by the same thread.

3.2 Implementation

We have implemented the LAIO API as a user-level library. In this section we explain the operation and the implementation of the LAIO APIs.

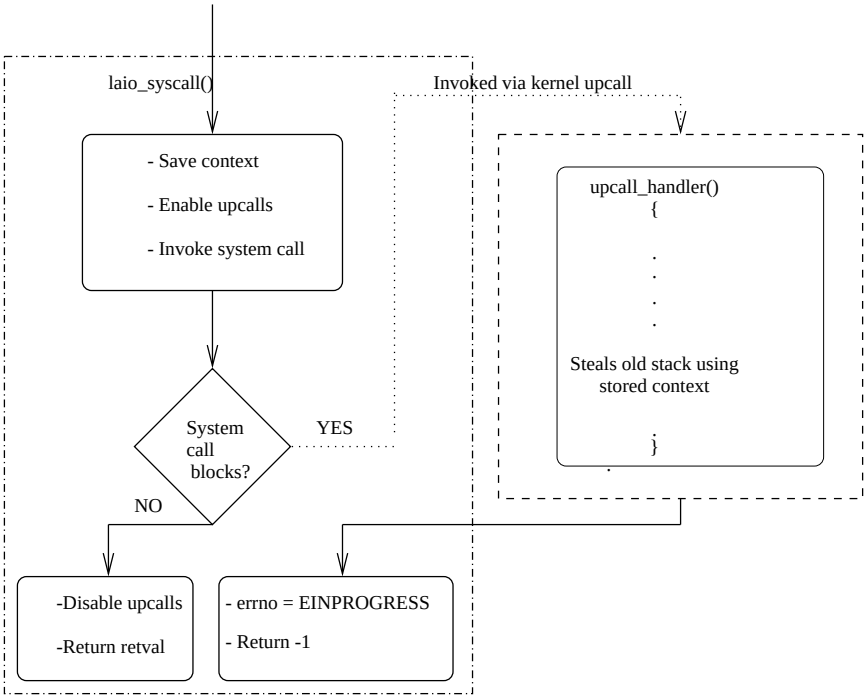


Figure 4: LAIO Syscall

Figure 4 illustrates the operation of `laio_syscall()`. It is a wrapper around any system call, where it saves the current thread's context and enables upcalls to be delivered. It then invokes the system call. If the system call doesn't block, it returns immediately to the application with the corresponding return value and upcalls are disabled. But if the system call blocks, blocking the current thread in the kernel, an upcall is generated by the kernel invoking the upcall handler on a new thread. The upcall handler then steals the previous thread's stack using the context previously saved by laio_syscall. Now running on the previous threads stack, the upcall handler returns from laio_syscall with the return value set to -1 and the `errno` set to `EINPROGRESS`. The EINPROGRESS errno value notifies the application that the laio_syscall has blocked in the kernel and a background laio_syscall is in progress. It then makes a call to `laio_gethandle` to get the LAIO descriptor associated with the last asynchronous I/O. It associates a continuation function with that LAIO descriptor. This continuation function is invoked after the background laio_syscall completes.

Unblocking of the background laio_syscall generates another upcall. This upcall returns the LAIO descriptor for the laio_syscall that had earlier blocked. The library adds this descriptor to a list of descriptors corresponding to background laio_syscalls that have completed. The application calls `laio_poll` to retrieve this list. The application then invokes the continuation function associated with each descriptor from this list. One thing to note here is that when the background laio_syscall completes it does not resume execution from the point where it blocked. Instead, a continuation function associated with the background laio_syscall is invoked.

The LAIO library maintains a pool of LAIO descriptors. It associates a descriptor with each laio_syscall. If a laio_syscall blocks, the upcall returns the corresponding LAIO descriptor. This is followed by a call to `laio_gethandle`, which returns the LAIO descriptor corresponding to the background laio_syscall.

We require scheduler activations [1] to provide upcalls for the implementations of the LAIO library. Scheduler activations are kernel support for management of parallelism at user level. Currently, many operating systems support scheduler activations, including FreeBSD [8], NetBSD [12], Solaris, and Tru64.

They provide a way of communication between the kernel and the user level program about events the application thread goes through inside the kernel. Blocking of the thread in the kernel, due to I/O, for example, and later unblocking of the thread are principal examples of such events. Such information could be used at the user level to make scheduling decisions. This information is provided to the application via upcalls. When the application thread blocks a new thread is spawned by the kernel and the upcall is provided on this thread. The application can continue execution on this newly created thread. An `upcall_handler` is defined in the LAIO library to handle upcalls for the application.

4 Integration of LAIO with libevent

In order to demonstrate the flexibility and completeness of the LAIO API, we augmented Niels Provos' libevent, which is a general-purpose event notification library [6], with support for LAIO. Libevent has been used to implement numerous event-driven servers.

Internally, libevent implements a conventional event loop. The principle modification to this event loop was the addition of a call to `laio_poll()` in order to detect the completion of background `laio_syscall()`s.

Externally, libevent provides an API to do event registration. The event registration API allows for the association of an event handler function with an event such that the handler is invoked when the event occurs. To integrate LAIO, we had to extend it to support an LAIO completion event.

Figure 5 illustrates an event handler using LAIO and libevent. Specifically, this event handler performs a `write()` using `laio_syscall()`. In contrast to Figure 3, the application is not required to maintain state on the progress of the I/O operation. If the `laio_syscall` reports `EINPROGRESS`, the function `client_response_epilogue` is registered to be invoked when the `write` operation has completed.

```
/* Step 1: attempt the operation */
return_value = laio_syscall(SYS_write,
                            client->socket,
                            client->buffer,
                            client->bytes_to_write);
if (return_value == client->bytes_to_write) {
    /* completed without blocking */
    client_response_epilogue(..., client);
    return; /* to the event loop */
} else if (return_value == -1) {
    if (errno == EINPROGRESS) {
        /* Step 2: tell the event loop to call client_response_epilogue
         * upon completion of the LAIO */
        event_set(&client->event, laio_gethandle(), EV_LAIO_COMPLETED,
                  client_response_epilogue, client);
        event_add(&client->event, NULL);
    } else {
        /* handle fatal error */
    }
    return; /* to the event loop */
}
...
/* perform other activities */
```

Figure 5: An Example of an Event Handler Performing LAIO

5 Servers

In this section, we focus on the two servers, thttpd [5] and Flash [4], that we use in our evaluation. In particular, we describe their respective architectures and the changes to them that we implemented for our evaluation.

5.1 thttpd

thttpd has an event driven architecture. It has an event loop which polls for events and invokes the associated event handlers. All sockets are configured in non-blocking mode. An event is received in the event loop when a socket becomes ready for a `read` or a `write`, and the corresponding event handler is invoked. thttpd makes calls to blocking operations like `open`, `stat`, and `sendfile` from within its event handlers. thttpd stalls if these operations block on disk I/O.

We modified the event driven version of flash to use the libevent event library API [6]. We call this version of thttpd as *thttpd-libevent*.

We have modified thttpd-libevent to use LAIO in order to convert its blocking operations into asynchronous ones. All blocking system calls are invoked via `laio_syscall()`. Continuation functions are defined to handle laio_syscalls that block and finish asynchronously. This version of thttpd is termed as *thttpd-LAIO*.

5.2 Flash

Flash employs the asymmetric multiprocess event driven (AMPED) architecture [4]. It has an event driven core to handle all non-blocking operations. Additionally, it has helper threads to handle all blocking operations. In Flash all sockets are used in non-blocking mode. All non-blocking socket operations are handled by the event driven core, and all potentially blocking operations like file open or read are handled by the helper threads. The thread running the event driven core dispatches work to the helper threads via remote procedure call (RPC) mechanism. Likewise, the helper threads employ RPC to notify the main thread of the completion of their tasks.

The original version of Flash mapped files in memory and subsequently wrote them to sockets [4]. After accepting a connection and reading the requested URL, Flash opens the requested file and maps it to its address space by the `mmap` system call. Since opening a file is a potential blocking operation it is done by a helper thread. After the file is mapped it is written to the corresponding connection by the `writev` system call. Although the socket for the connection is non-blocking, the writev system call could still block if the memory mapped file is not resident in memory and the thread would incur a page fault. Since memory residency of files is not guaranteed and page faults are not acceptable in the event driven core thread, Flash employed the `mincore` system call (on FreeBSD) to check for memory residency of concerned pages. If pages were found to be not in memory, explicit I/O was issued by helper threads; we refer to these helper threads as read helper threads. This is how blocking due to page faults was avoided. We call this version of Flash as *Flash-AMPED-mmap*.

Current version of Flash supports the `sendfile` system call. In this version the event driven core reads the requested URL after accepting the connection. The corresponding file is opened by a helper thread. Then the event driven core uses sendfile to write the file to the corresponding socket. Although the socket is used in non-blocking mode, sendfile can block if the file data is not in memory, causing the main thread to block. This is avoided by using an *optimized* sendfile [7]. This requires a patch in the kernel which prevents sendfile from blocking on disk I/O and returns a special `errno` . The event core catches this errno and issues explicit I/O via read helper threads so that the file data is brought in memory. We call this version of Flash as *Flash-AMPED-sendfile*. Note, this version of Flash does I/O in a lazy manner which is analogous to LAIO. It calls sendfile expecting it not to block, but if it blocks on disk I/O, the sendfile call returns with a special error code, which is used to initiate I/O via helper threads. As a result, the server main thread does not block. As such we do not expect the corresponding LAIO version of Flash to outperform Flash-AMPED-sendfile but to match it.

We have modified both Flash-AMPED-mmap and Flash-AMPED-sendfile to get the corresponding event driven versions. We call these versions Flash-Event-mmap and Flash-Event-sendfile respectively. In either

version, there are no helper threads, instead, all helper functions are called directly by the main thread. This means, for operations like file `open`, the server may block, as would happen in a pure event driven server. For Flash-Event-mmap we do not use the mincore operation. This is because, even if mincore detects that some page is not in memory, the main thread would have to block at the I/O for the page. In such cases the mincore operation is nothing but an overhead. Instead, we allow writev to incur page faults when pages are not in memory. For Flash-Event-sendfile we do not use the optimized sendfile because of similar reason.

We have modified both Flash-Event-mmap and Flash-Event-sendfile to use the LAIO library to prevent any blockings. We call these versions Flash-LAIO-mmap and FLash-LAIO-Sendfile respectively. In either version, all potentially blocking operations are identified and the LAIO API is used to prevent (possible) blockings, for example, the writev or sendfile system calls. Since LAIO is a universal API to handle any potentially blocking operation asynchronously, we do not use non-blocking sockets in Flash-LAIO-mmap and Flash-LAIO-sendfile. Instead, all sockets are synchronous and we let the LAIO API to take care of any blocking events that may happen. In Flash-LAIO-mmap we do not use the mincore operation, because, even if pages are not in memory for a writev operation, the server does not block because of the LAIO API. Similarly, for Flash-LAIO-sendfile we do not use the optimized sendfile.

6 Methodology

In this section we present the methodology we used to evaluate the LAIO library. We used two web server applications for this purpose - thttpd and Flash. As described earlier we have two different versions of thttpd, viz., thttpd-libevent and thttpd-LAIO, and six different versions of Flash, viz., Flash-Event-mmap, Flash-Event-sendfile, Flash-AMPED-mmap, Flash-AMPED-sendfile, Flash-LAIO-mmap, and Flash-LAIO-sendfile. We compare the performance of thttpd-Event and thttpd-LAIO, similarly we compare the three versions of Flash using mmap and the other three versions of Flash using sendfile.

We used a couple of trace based web workload for this purpose. These workloads are obtained from the academic web servers at Rice University (Rice workload) and the University of California at Berkeley (Berkeley workload). Use of these workloads exists in published literature [13]. Each workload is associated with a trace file of web requests.

Web Workload	No. of requests	Small ($\leq$ 8 KB)	Medium ($>$ 8 KB and $\leq$ 256 KB)	Large ($>$ 256 KB)	Total footprint
Rice	245,820	5.5%	20.2%	74.3%	1.1 Gigabytes
Berkeley	3,184,540	8.2%	33.2%	58.6%	6.4 Gigabytes

Table 1: Web Trace Characteristics

Table 2 shows the characteristics of the Rice and Berkeley web workloads. The total number of requests in the Rice and the Berkeley workload are 245,820 and 3,184,540 respectively. Off the total bytes transferred, the fractions contributed by small files (size less than or equal to 8 Kilobytes), medium files (size in between 8 Kilobytes and 256 Kilobytes), and large files (size greater than 256 Kilobytes) are shown in columns *Small*, *Medium*, and *Large* respectively. Total footprint for the Rice and the Berkeley workloads are 1.1 Gigabytes and 6.4 Gigabytes respectively.

The trace file associated with each workload is a set of request sequences. A sequence consist of one or more requests. Each sequence begins with a connection set up and ends with a connection tear down. Requests in a sequence are sent one at a time, the response is read completely, and then the next request is sent. All requests in a sequence are sent over a persistent HTTP connection. If the server does not support persistent connections, like thttpd, each request is sent over a non-persistent HTTP connection, that is, the connection is set up and tore down before and after each request respectively.

We used a program that simulates concurrent clients sending web requests to a web server. The number of concurrent clients can be varied with this program. It supports persistent and non-persistent connections. The program simulates multiple clients that play the request sequences in the trace against the server. The program terminates when the trace is exhausted and reports overall throughput and response time.

We used a Pentium Xeon 2.4 GHz machine with 2 Gigabytes of memory as our server machine. It runs FreeBSD-5 which supports the *KSE*, FreeBSD's scheduler activations implementation. An identical machine was used as the client machine. The server and client machines were connected by a Gigabit ethernet switch.

7 Results

In this section we present experimental results obtained. Section 7.1 shows some micro-benchmarks' results. Section 7.2 covers the thttpd experiments. Section 7.3 covers the Flash experiments.

7.1 Microbenchmarks: LAIO vs. non-blocking I/O vs. POSIX AIO

In order to compare the cost of performing I/O using LAIO, non-blocking I/O, and POSIX AIO, we implemented a set of microbenchmarks. Specifically, these microbenchmarks measured the cost of 100,000 iterations of reading a single byte from a pipe under various circumstances. For POSIX AIO, the microbenchmarks include calls to `aio_error` and `aio_return` in order to obtain the read's error and return values, respectively. We used a pipe so that irrelevant factors, such as disk access latency, did not affect our measurements. Furthermore, the low overhead of I/O through pipes would emphasize the differences between the three mechanisms. In one case, when the read occurs a byte is already present in the pipe, ready to be read. In the other case, the byte is not written into the pipe until the reader has performed either the LAIO or the `aio_read`. In this case, we did not measure the cost of a non-blocking read because the read would immediately return `EAGAIN`.

As would be expected, when the byte is already present in the pipe before the read, non-blocking I/O performed the best. LAIO was a factor of 1.4 slower than non-blocking I/O; and AIO was a factor of 4.48 and 3.2 slower than non-blocking I/O and LAIO, respectively. In the other case, when the byte was not present in the pipe before the read, we found that LAIO was a factor of 1.08 slower than AIO.

In these microbenchmarks, only a single byte was read at a time. Increasing the number of bytes read at a time, did not change the ordering among LAIO, non-blocking I/O, and POSIX AIO as to which performed best.

In addition, we observed that enabling scheduler activations within an address space resulted in a small added overhead to all system calls by any thread within that same address space. To characterize this overhead, we measured the cost of 1,000,000 iterations of calling `getpid`. We found that enabling scheduler activations made `getpid` a factor of 1.05 slower. In effect, this represents an indirect cost to users of LAIO or asynchronous I/O implementations using a pool of threads.

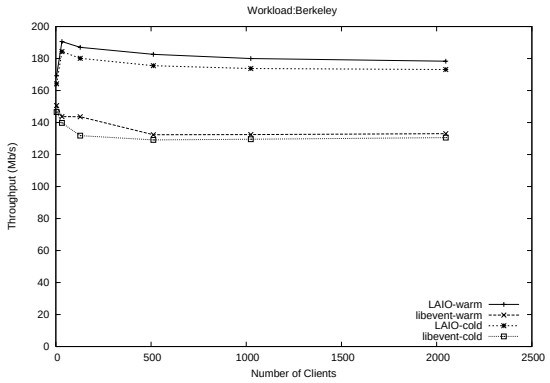
7.2 thttpd Results

We subjected the thttpd server to traces collected from real web servers to obtain the following results. In Figure ??(a) we see the throughput of the cold and warm cache cases for Berkeley CS department workload. thttpd-LAIO achieves more than 38% more throughput than libevent, because the workload is to big to fit in memory, thus system calls like sendfile, stat and open block on disk reads often, which is not the case for thttpd-libevent. For the warm cache case, we note there isn't much difference, this is because compulsory misses are not the dominating factor.

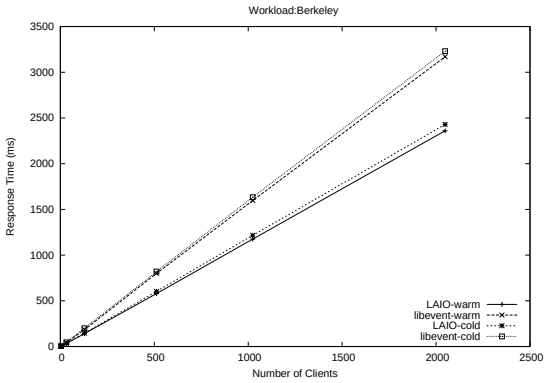
Figure ??(b) shows the response time for the Berkeley trace for cases of cold and warm caches. We notice that under heavy load there is a significant improvement in the response time for thttpd-LAIO , more than 30% reduction.

Figures ??(a) shows the throughput for the Rice CS department workload for cold and warm caches. Unlike the Berkeley workload this workload fits in memory. So for the cold cache we see some gain from LAIO, this is basically due to compulsory misses, causing the server to make disk reads. For the warm cache we don't see any gain from LAIO as all the files are already in memory and there is no blocking on I/O. We actually see a little performance degradation (less than 2%).

Figure ??(b) shows the response time for the Rice trace for cases of cold and warm caches. For the warm case we find the thttpd-LAIO follows the libevent closely. For the cold cache case thttpd-LAIO has a lower response time, again this is due compulsory misses.



(a) Throughput for Berkeley workload



(b) Response time for Berkeley workload

Figure 6: Results for Berkeley workload with thttpd

7.3 Flash Results

In this section we compare the performance of different versions of Flash, as described in Section 5.2, for the Rice workload and the Berkeley workload. 500 concurrent clients were used to send the workload requests to the server. We measured the overall throughput reported by the client program. For each server version we do one cold cache run followed immediately by a warm cache run, for each workload.

We also conducted a series of experiments where we varied the number of clients, as with the thttpd experiments. The results for the various versions of Flash conform to our expectations. In particular, the results scale similarly to the thttpd results as described in Section 7.2. The relative ordering of the various versions of the flash remain the same as described in the next two sections. Hence, we argue that the the results with 500 clients, for the various versions of Flash, are representative of the overall experiments.

7.3.1 Mmap Results

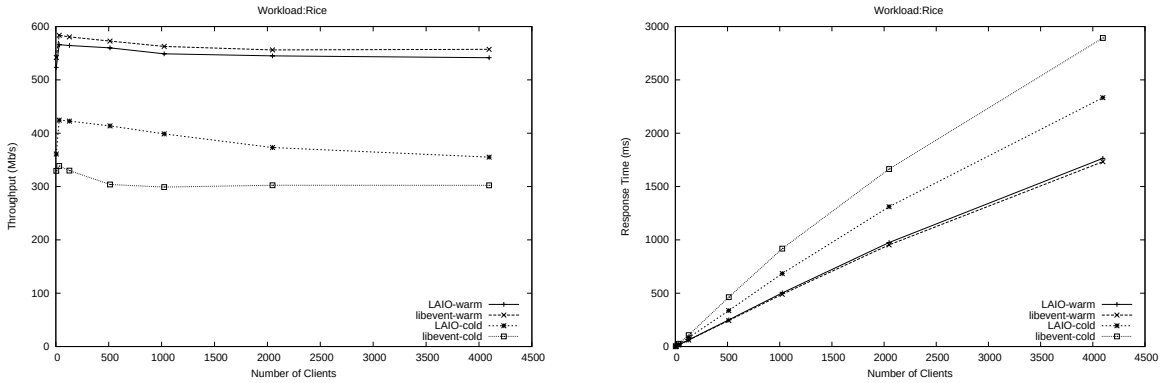
First, we compare the performance of Flash-Event-mmap, Flash-AMPED-mmap and Flash-LAIO-mmap. All these versions use `mmap` to map the requested file to memory, and `writev` to send the file across the connection. Additionally, Flash-AMPED-mmap uses the `mincore` system call to check for memory residency of pages before sending them out (as described in Section 5.2).

Configuration	Flash-Event-mmap	Flash-AMPED-mmap	Flash-LAIO-mmap
Cold cache	203.16 Mbps	385.83 Mbps	298.76 Mbps
Warm cache	830.22 Mbps	799.74 Mbps	796.82 Mbps

Table 2: Mmap results: Throughput for the Rice workload with 500 clients

Configuration	Flash-Event-mmap	Flash-AMPED-mmap	Flash-LAIO-mmap
Cold cache	81.18 Mbps	133.59 Mbps	132.02 Mbps
Warm cache	78.36 Mbps	126.67 Mbps	131.21 Mbps

Table 3: Mmap results: Throughput for the Berkeley trace with 500 clients



(a) Throughput for Rice workload (b) Response time for Rice workload

Figure 7: Results for Rice workload with thttpd

Table 3 and Table 4 show the throughput of the three server configurations for the Rice workload and Berkeley workload with 500 concurrent clients respectively. Results for a cold cache run followed by a warm cache run are shown. Recall, that the total footprint of the Rice workload is 1.1 Gigabytes and that of the Berkeley workload is 6.4 Gigabytes. The server machine had 2 Gigabytes of memory, so in the warm cache run the entire workload fits in memory for the Rice workload. But, for the Berkeley workload, there is considerable I/O even for the warm cache run.

For both the workloads in cold cache runs, the throughput attained by Flash-AMPED-mmap is higher than that of Flash-Event-mmap. Specifically, for the Rice workload Flash-AMPED-mmap has about 90% performance improvement over Flash-Event-mmap, while, for the Berkeley workload it has about 65% performance improvement. Flash-Event-mmap blocks on operations like `open`, `stat`, and `writev`, and hence the performance difference. For cold cache runs in both workloads Flash-LAIO-mmap betters Flash-Event-mmap because of the same reasons stated above. For warm cache runs in both the workloads Flash-LAIO-mmap closely matches Flash-AMPED-mmap.

For the Rice workload in warm cache run, Flash-Event-mmap outperforms Flash-AMPED-mmap by about 4%. For this workload in warm cache run the entire workload fits in memory. In this case, the mincore operations of Flash-AMPED-mmap is nothing but an overhead, and hence the performance difference.

For the Rice workload in cold cache run, Flash-AMPED-mmap outperforms Flash-LAIO-mmap by about 29%. We measured the number of time Flash-AMPED-mmap calls the read helper, which does I/O on behalf of the server main thread. This number was 41072. Flash-LAIO-mmap does not have any helper thread, instead it page faults on the *writev* system call if the data is not memory. The number of page faults was 46486, which is about 13% higher than the number of calls to the read helper in Flash-AMPED-mmap. Note, that the purpose of the read helper and the page faults is to perform I/O and bring the required data in memory. So, the same amount of I/O is being performed in both cases but it takes 13% more I/O operations in Flash-LAIO-mmap. The difference in the prefetching algorithm of the file system prefetch of *read* system call and the virtual memory system prefetch of page faults is causing the performance difference here. In particular, file system prefetch is more aggressive than virtual memory system prefetch.

For the Berkeley workload in cold cache run the number of calls to the read helper in Flash-AMPED-mmap was 689167, while the number of page faults in Flash-LAIO-mmap was 690835. For the warm cache run, these numbers were 681320 and 670782. These explain why Flash-AMPED-mmap performs a little better than Flash-LAIO-mmap in cold cache case and vice versa in the warm cache case.

For the Berkeley workload in cold cache run there is considerable blocking in Flash-Event-mmap because

the workload does not fit in memory. Hence, Flash-Event-mmap performs worse than the other two servers.

7.3.2 Sendfile Results

In this section we compare the versions of Flash that use `sendfile` to transfer data across the network. Sendfile writes file referenced by an opened file descriptor to a socket. It can block if the data for the file is not resident. However, Flash-AMPED-sendfile utilizes an optimized version of sendfile that sends a special `errno` on blocking on disk, and the server can continue its operation without blocking. The server main thread then does an I/O via the read helper thread to bring the file data in memory. In our experiments Flash-AMPED uses this optimized sendfile, while Flash-Event and Flash-LAIO use the normal sendfile. A kernel patch from the original developers of Flash is required to get the optimized sendfile; this is described in Section 5.2. Recall that we do not expect Flash-LAIO-sendfile to outperform Flash-AMPED-sendfile, but to match its performance.

Configuration	Flash-Event-sendfile	Flash-AMPED-sendfile	Flash-LAIO-sendfile
Cold cache	276.87 Mbps	397.78 Mbps	381.54 Mbps
Warm cache	844.76 Mbps	843.45 Mbps	815.49 Mbps

Table 4: Sendfile results: Throughput for the Rice workload with 500 clients

Configuration	Flash-Event-sendfile	Flash-AMPED-sendfile	Flash-LAIO-sendfile
Cold cache	121.59 Mbps	170.59 Mbps	171.43 Mbps
Warm cache	124.60 Mbps	180.42 Mbps	178.89 Mbps

Table 5: Sendfile results: Throughput for the Berkeley trace with 500 clients

Table 5 and Table 6 show the results for the Rice workload and the Berkeley workload under cold cache and warm cache runs respectively.

For both the workloads under cold cache run Flash-Event-sendfile performs worse than the other two servers. This is because it blocks at file `open`, `stat`, and `sendfile` while the other two servers do not block under such conditions.

For Rice workload in warm cache run, Flash-Event-sendfile matches the performance of Flash-AMPED-sendfile. This is because the entire workload is cached and fits in memory, so the calls to sendfile do not block for disk I/O.

For Berkeley workload in warm cache run, however, Flash-Event-sendfile performs worse than Flash-AMPED-sendfile. This is because the workload does not fit in memory and Flash-Event-sendfile blocks on disk I/O.

For Rice workload in cold cache run, Flash-LAIO-sendfile closely matches the performance of Flash-AMPED-sendfile. For Berkeley workload in both cold cache run and warm cache run, Flash-LAIO-sendfile matches the performance of Flash-AMPED-sendfile. This is because the optimized sendfile is doing I/O lazily like LAIO.

For Rice workload in warm cache run, Flash-LAIO-sendfile performs about 3% worse than Flash-AMPED-sendfile. This is because of the cost of LAIO as explained in Section 7.1.

8 Conclusions

In this paper we have introduced Lazy Asynchronous I/O (LAIO), a new API for performing I/O that befits event-driven servers. We have pointed out the shortcomings of both asynchronous and non-blocking I/O with respect to server performance. We have explained the design and implementation of LAIO, demonstrated that it subdues the shortcomings of the earlier APIs. Using a micro-benchmark, LAIO was shown to be more than 3 times faster than AIO when the data was already available in memory. It also had a comparable

performance to AIO when actual I/O needed to be made. We have also shown that thttpd achieved more than 38% increase in its throughput using LAIO. The Flash web server's throughput, originally achieved with kernel modifications, was matched using LAIO without making kernel modifications.

We also argue that better scheduling decisions could be made if more information is made available to the scheduler activation upcall. Like what was the reason for blocking. For example, knowing that there are many threads that have blocked on disk I/O, the scheduler may choose not to start new threads that are likely to do disk operations.

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *2002 USENIX Annual Technical Conference*, pages 103–114, June 2002.

[3] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[4] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[5] J. Poskanzer. thttpd - tiny/turbo/throttling http server. Version 2.24 is available from the author's web site, http://www.acme.com/software/thttpd/, Oct. 2003.

[6] N. Provos. Libevent - an event notification library. Version 0.7c is available from the author's web site, http://www.monkey.org/~provos/libevent/, Oct. 2003. Libevent is also included in recent releases of the NetBSD and OpenBSD operating systems.

[7] Y. Ruan and V. S. Pai. Making the "Box" transparent: System call performance as a first-class result. Technical report, Princeton University, 2003.

[8] The FreeBSD Project. FreeBSD KSE Project. At http://www.freebsd.org/kse/.

[9] The Open Group Base Specifications Issue 6 (IEEE Std 1003.1, 2003 Edition). Available from http://www.opengroup.org/onlinepubs/007904975/, 2003.

[10] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. To appear at the 19th ACM Symposium on Operating Systems Principles, Oct. 2003.

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

[12] N. Williams. An implementation of scheduler activations on the netbsd operating system, June 2002.

[13] H. youb Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *Architectural Support for Programming Languages and Operating Systems*, pages 239–250, San Jose, California, Oct. 2002.