Lazy Asynchronous I/O For Event-Driven Servers

Khaled Elmeleegy, Anupam Chanda, and Alan L. Cox
Department of Computer Science
Rice University, Houston, Texas 77005, USA
{kdiaa,anupamc,alc}@cs.rice.edu

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

We introduce *Lazy Asynchronous I/O* (LAIO), a new asynchronous I/O interface that is well suited to event-driven programming. LAIO is *general* in the sense that it applies to all blocking I/O operations. Furthermore, it is *lazy* in the sense that it creates a continuation only when an operation actually blocks, and it notifies the application only when a blocked operation completes in its entirety. These features make programming high-performance, event-driven servers using LAIO considerably easier than with previous interfaces.

We describe a user-level implementation of LAIO, relying only on kernel support for scheduler activations, a facility present in many Unix-like systems.

We compare the performance of web servers implemented using LAIO to the performance obtained with previous interfaces. For workloads with an appreciable amount of disk I/O, LAIO performs substantially better than the alternatives, because it avoids blocking entirely. In one such case, the peak throughput with LAIO is 24% higher than the next best alternative. For in-memory workloads it performs comparably.

1 Introduction

We introduce *Lazy Asynchronous I/O* (LAIO), a new asynchronous I/O interface that is well suited to event-driven programs, in particular high-performance servers.

To achieve the best possible performance, an event-driven server must avoid blocking on any type of operation, from I/O to resource allocation. Thus, in an event-driven server, the use of asynchronous or non-blocking I/O is a practical necessity. Asynchronous and non-blocking I/O support in present Unix-like systems is, however, limited in its generality. Non-blocking I/O can be performed on network connections, but not on disk files. POSIX asynchronous I/O (AIO) [11] 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, have no asynchronous equivalents.

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 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 tradeoff between multithreaded and event-driven servers has received considerable attention [5, 6, 7, 13]. Event-driven servers exhibit certain advantages including greater control over scheduling, lower overhead for maintaining state, and lower overhead for synchronization. Recently, von Behren et al. [12] have argued that compiler support and non-preemption can enable multithreaded servers to achieve performance comparable to event-driven servers, but such compiler support is not generally available.

Surprisingly, the second option does appear in practice. The thttpd web server [8] is a notable example, but as we show in Section 9, performance suffers as a result of blocking.

The *asymmetric multiprocess event-driven*

(AMPED) architecture that was employed by the Flash web server is representative of the third category [7]. In essence, it is a hybrid architecture that consists of an event-driven core augmented by helper processes. Flash performs all non-blocking operations in an event-driven fashion and all potentially blocking operations are dispatched to helper processes.

Unlike non-blocking I/O and AIO, LAIO is *general*. LAIO offers a non-blocking counterpart for each blocking system call, thereby avoiding the tradeoffs and the programming difficulties present with previous asynchronous I/O systems.

In addition to its generality, LAIO offers *lazy continuation creation*. If a potentially blocking system call completes without blocking, no continuation is created, avoiding the implementation cost of its creation and the programming complexity of dealing with it. Lazy continuation creation distinguishes LAIO from AIO in which executing an AIO primitive always creates a continuation, regardless of whether the call blocks or not. Furthermore, the LAIO API provides an event notification when a blocked call is *completed in its entirety*. This feature distinguishes LAIO from non-blocking I/O in which a return from a non-blocking call may indicate partial completion and in which the application may need to maintain state related to that partial completion.

Our implementation of LAIO resides entirely in a user-level library, without modification to the operating system's kernel. It requires support in the kernel for scheduler activations [3] to deliver upcalls to the LAIO library when an I/O operation blocks or unblocks.

The contributions of this paper are three-fold. First, we introduce a new asynchronous I/O interface, which is easier to program and performs better than previous I/O interfaces for workloads with an appreciable amount of disk I/O. Second, we document the ease of programming by comparing LAIO to non-blocking I/O, AIO and AMPED. For LAIO, non-blocking I/O, and AMPED, we quantify this comparison by counting the lines of affected code in the Flash web server [7]. Third, we evaluate the performance of LAIO by comparing the performance of two web servers, thttpd [8] and Flash. We show that LAIO reduces blocking compared to non-blocking I/O and asynchronous I/O and incurs less overhead than AMPED.

The remainder of this paper is organized as follows. Section 2 describes the LAIO API. Section 3 provides an example of using LAIO. Section 4 discusses other I/O APIs and their use in event-driven programs, and compares them with LAIO. Section 5 describes our LAIO implementation. Section 6 describes our experimental environment. Section 7 characterizes the performance of our implementation using a set of microbenchmarks. Section 8 describes the web server software and the workloads used in our macrobenchmarks. Section 9 describes the experimental results obtained with these macrobenchmarks. Section 10 discusses related work. Section 11 concludes this paper.

2 The LAIO API

The LAIO API 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 correspond to the parameters of the system call being performed. If the system call completes 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()`." In such cases, any input parameters that are passed by reference, such as a buffer being written to a file, must not be modified by the caller until the background `laio_syscall()` completes.

`laio_gethandle()` returns an opaque handle for the purpose of identifying a background `laio_syscall()`. Specifically, this handle identifies the most recent `laio_syscall()` by the calling thread that reported `EINPROGRESS`. If the most recent `laio_syscall()` completed without blocking, `laio_gethandle()` returns `NULL`. `laio_gethandle()` is expected to appear shortly after the return of a `laio_syscall()` with return value `-1` and with `errno` equal to `EINPROGRESS`.

`laio_poll()` waits for the completion of background `laio_syscall()` operations. When one or more such operations have completed before a caller-specified timeout expires, `laio_poll()` returns a set of *LAIO com-*

pletion objects, one per completed background `laio_syscall()`. A completion object consists of a handle, a return value, and a possible error code. The handle identifies a background `laio_syscall()`, as returned by `laio_gethandle()`. The return value and possible error code are determined by the particular system call that was performed by the background `laio_syscall()`.

3 An LAIO Example

We present an example demonstrating how to write an event-driven program using LAIO. In this example, we also use *libevent* [9], a general-purpose event notification library. In general, such libraries provide a unifying abstraction for the various mechanisms that apply to different types of events. We have extended libevent to support completion of a background `laio_syscall()` as an event type. LAIO can, however, be used in isolation or with other event notification libraries.

Three functions from the libevent interface appear in this and subsequent examples: `event_add()`, `event_del()`, and `event_set()`. All of these functions work with an *event object* that has three principal attributes: the object being monitored, the desired state of that object, and a handler that is called when this desired state is achieved. For example, an event might designate a handler that is called when a socket has data available to be read. An event is initialized using `event_set()`. For the programmer's convenience, `event_set()` also accepts as its final argument an opaque pointer that is passed to the handler. `event_add()` enables monitoring for an initialized event. Unless an event is initialized as *persistent*, monitoring is disabled when the event occurs. `event_del()` disables monitoring for events that are either persistent or have not occurred.

In an event-driven program, event monitoring is performed by an infinite event loop. For each occurrence of an event, the event loop dispatches the corresponding event handler. Figure 1 shows the outline of the event loop in a program using LAIO. `laio_poll()` is used for event monitoring. It returns a set of LAIO completion objects, one for each completed background `laio_syscall()`. For each LAIO completion object, the event loop locates the corresponding event object (code not shown in the figure). It then invokes the continuation function stored in that event object with the arguments returned in the LAIO completion object.

Figure 2 presents an event handler that writes data to a network socket. The write operation is initiated using `laio_syscall()`. If the write does not block, the number of bytes written is returned. The execution continues to `client_write_complete()`, and the event handler returns to the event loop.

In the more interesting case in which the write blocks, `-1` is returned and `errno` is set to `EINPROGRESS`. The program calls `laio_gethandle()` to get the handle associated with the background `laio_syscall()` operation. It initializes an event object, and associates the continuation function `client_write_complete()` with that LAIO handle. The event handler then returns to the event loop. The continuation function is invoked by the event loop after the background `laio_syscall()` completes (see Figure 1).

4 Comparison to Other I/O APIs

We describe non-blocking I/O and AIO and explain their usage in event-driven programming. We compare the LAIO API with these other I/O APIs.

4.1 Non-blocking I/O

With non-blocking I/O the programmer declares a socket as non-blocking, and then performs the normal I/O system calls on it. If the operation does not block, the operation returns normally. Otherwise, the operation returns when some part of the operation has completed. The amount of work completed is indicated by the return value.

Figures 3 and 4 illustrate how the event loop and the event handler presented in Figures 1 and 2 are programmed using non-blocking I/O instead of LAIO.

The event loop is conceptually the same as with LAIO. For non-blocking I/O, polling in the event loop can be done using any standard event monitoring mechanism like `select()`, `poll()`, or `kevent()`.

The event handler clearly illustrates the consequences of the fact that to avoid blocking the system call may return with the operation only partially completed. In this case the return value is different from the number of bytes provided as an argument to the system call. The application needs to remember how much of the data has

```
for (;;) {
   ...
   /* poll for completed LAIO operations; laioc_array is an array of LAIO completion
    * objects; it is an output parameter */
   if ((ncompleted = laio_poll(laioc_array, laioc_array_len, timeout)) == -1)
      /* handle error */
   for (i = 0; i < ncompleted; i++) {
      ret_val = laioc_array[i].laio_return_value;
      err_val = laioc_array[i].laio_errno;
      /* find the event object for laioc_array[i].laio_handle */
      eventp->ev_func(eventp->ev_arg/* == clientp */, ret_val, err_val);
      /* disable eventp; completions are one-time events */
   }
   ...
```

Figure 1: Event loop using LAIO

```
client_write(struct client *clientp)
{
   ...
   /* initiate the operation; returns immediately */
   ret_val = laio_syscall(SYS_write, clientp->socket, clientp->buffer,
      clientp->bytes_to_write);
   if (ret_val == -1) {
      if (errno == EINPROGRESS) {
         /* instruct event loop to call client_write_complete() upon completion
          * of this LAIO operation; clientp is passed to client_write_complete() */
         event_set(&clientp->event, laio_gethandle(), EV_LAIO_COMPLETED,
            client_write_complete, clientp);
         event_add(&clientp->event, NULL);
         return; /* to the event loop */
      } else {
         /* client_write_complete() handles errors */
         err_val = errno;
      }
   } else
      err_val = 0;
   /* completed without blocking */
   client_write_complete(clientp, ret_val, err_val);
   ...
```

Figure 2: Event handler using LAIO

been written. When the operation unblocks, an event is generated, and the event handler is called again, with the remaining number of bytes to be written. Several such write system calls may be required to complete the operation. Hence, the event is defined as persistent and deleted by the event handler only when the operation has completed.

4.2 AIO

The AIO API provides asynchronous counterparts, `aio_read()` and `aio_write()`, to the standard read and write system calls. In addition, `aio_suspend()` allows the program to wait for AIO events, and `aio_return()` and `aio_error()` provide the return and the error values of asynchronously executed calls. An AIO control block is used to identify asynchronous operations.

Figures 5 and 6 show the outline of the event loop and an event handler, respectively, using AIO. The event loop is very similar to LAIO, except that two separate calls, to `aio_error()` and to `aio_return()`, are necessary to obtain the error and return values.

The more important differences are in the event handler in Figure 6. First, an AIO 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, its size and the completion notification mechanism. Then, the operation is initiated. In the absence of errors, the event han-

```
for (;;) {
   ...
   /* poll for fds that are ready to read and/or write; pfd_array is an array of
    * pollfd objects listing blocked fds; it is an input and output parameter */
   if ((nready = poll(pfd_array, pfd_array_len, timeout)) == -1)
      /* handle error */
   for (i = 0; nready > 0 && i < pfd_array_len; i++) {
      if (pfd_array[i].revents & (POLLIN | POLLOUT)) {
         if (pfd_array[i].revents & POLLIN) { /* ready to read */
            /* find the read event object for pfd_array[i].fd */
            eventp->ev_func(eventp->ev_arg/* == clientp */);
         }
         if (pfd_array[i].revents & POLLOUT) { /* ready to write */
            /* find the write event object for pfd_array[i].fd */
            eventp->ev_func(eventp->ev_arg/* == clientp */);
         }
         nready--;
      }
   }
   ...
```

Figure 3: Event loop using non-blocking I/O

dler returns to the event loop afterwards.

The control block is used as a handle to identify the asynchronous operation. In most implementations, the application determines that the operation has finished through polling or an asynchronous event notification mechanism, such as signals. Figure 5 illustrates polling using the `aio_suspend()` operation.

4.3 Comparison

The three key distinguishing features of LAIO are:

Generality LAIO works for all operations, e.g., file and socket operations.

Laziness LAIO creates a continuation only if the operation blocks.

Completion notification LAIO notifies the application when the event completes, not at some intermediate stage.

A principal difference between LAIO on one hand and non-blocking I/O and AIO on the other hand is that LAIO allows any system call to be executed asynchronously. Non-blocking I/O only supports sockets, but does not support file operations. AIO only supports basic I/O operations, like reading and writing. Common operations that include I/O as part of their implementation, such as opening a file or determining its size, are not supported.

Furthermore, asynchronous I/O systems can be distinguished along two dimensions: whether or not they create continuations if the operation does not block, and whether they provide a completion or a partial completion notification. LAIO creates continuations lazily and provides a completion notification. The examples in Sections 3, 4.1 and 4.2 clearly show the benefits of this combination in terms of programmability. Non-blocking I/O provides partial completion notification, requiring the application to maintain state about non-blocking calls in progress and to issue multiple I/O calls (see Figure 4). AIO creates continuations eagerly, requiring extra programming effort and extra system calls even if the operation does not block (see Figures 5 and 6).

5 An LAIO Implementation

LAIO is implemented as a user-level library.

The LAIO library maintains a pool of free LAIO handles. Each handle is an opaque pointer. The library passes an LAIO handle to the current running thread through an opaque pointer field in the thread structure. This is done via the KSE interface [10] before making any `laio_syscall()`. The library remembers the current handle in a variable `current_handle`. The library maintains another variable `background_handle` which is initialized to `NULL` before any `laio_syscall()`.

`laio_syscall()` is a wrapper around any system call. It saves the current thread's context and enables upcalls to be delivered. It then invokes the system call. If the operation does not block, it returns immediately to the appli-

```
client_write(struct client *clientp)
{
   ...
   /* assume that the one-time operations, enabling non-blocking I/O and
    * initializing the state of progress, have been performed elsewhere. */
   ...
   /* attempt the operation; returns immediately */
   ret_val = write(clientp->socket, &clientp->buffer[clientp->bytes_written],
      clientp->bytes_remaining);
   if (ret_val == clientp->bytes_remaining) { /* this write has completed */
      err_val = 0;
   } else if (ret_val > 0) { /* and implicitly less than bytes_remaining */
      if (clientp->bytes_written == 0) {
         /* instruct event loop to call client_write whenever clientp->socket
          * is ready to write; clientp is passed to client_write() */
         event_set(&clientp->event, clientp->socket, EV_PERSIST | EV_WRITE,
            client_write, clientp);
         event_add(&clientp->event, NULL);
      }
      /* update the state of progress */
      clientp->bytes_written += ret_val;
      clientp->bytes_remaining -= ret_val;
      return; /* to the event loop */
   } else if (ret_val == -1 && errno != EAGAIN) {
      /* client_write_complete() handles errors */
      err_val = errno;
   }
   if (clientp->bytes_written != 0) {
      /* instruct libevent that calls are no longer needed */
      event_del(&clientp->event);
   }
   client_write_complete(clientp, ret_val, err_val);
   ...
```

Figure 4: Event handler using non-blocking I/O

cation with the corresponding return value and upcalls are disabled. If the operation blocks, an upcall is generated by the kernel. The kernel creates a new thread and delivers the upcall on this new thread. At this point the thread associated with the handle `current_handle` has blocked. This handle is remembered in `background_handle` which now corresponds to the background `laio_syscall()`. Since, the current running thread has now changed, `current_handle` is set to a new handle from the pool of free LAIO handles. This new value of `current_handle` is associated with the current running thread via the KSE interface. `laio_gethandle()` returns the handle corresponding to the most recent background `laio_syscall()` which, as explained above, is saved in `background_handle`. The upcall handler then steals the blocked thread's stack using the context previously saved by `laio_syscall()`. Running on the blocked thread's stack, the upcall handler returns from `laio_syscall()` with the return value set to `-1` and the `errno` set to `EINPROGRESS`.

Unblocking of a background `laio_syscall()` generates another upcall. The upcall returns the handle corresponding to the unblocked thread via an opaque pointer field within the thread structure. The library adds this handle to a list of handles corresponding to background `laio_syscall()`s that have completed and frees the thread. The application calls `laio_poll()` to retrieve this list. After `laio_poll()` retrieves a handle it is returned to the pool of free LAIO handles.

We rely on scheduler activations [3] to provide upcalls for the implementation of the LAIO library. Many operating systems support scheduler activations, including FreeBSD [10], NetBSD [14], Solaris, and Tru64.

6 Experimental Environment

All of our machines have a 2.4GHz Intel Xeon processor, 2GB of memory, and a single 7200RPM ATA hard drive. They are connected by a gigabit Ethernet switch. They run FreeBSD

```
for (;;) {
   ...
   /* poll for completed AIO operations; aiocbp_array is an array of pointers
    * to the unfinished aiocbs; it is an input parameter */
   if (aio_suspend(aiocbp_array, aiocbp_array_len, timeout) == -1)
      /* handle error */
   for (i = 0; i < aiocbp_array_len; i++) {
      err_val = aio_error(aiocbp_array[i]);
      if (err_val == 0) { /* this aiocbp has completed */
         ret_val = aio_return(aiocbp_array[i]);
         /* find the event object for this aiocbp */
         eventp->ev_func(eventp->ev_arg/* == clientp */, ret_val, err_val);
         /* disable eventp; completions are one-time events */
      } else if (err_val == EINPROGRESS) { /* this aiocbp has not completed */
         continue;
      } else
         /* handle error */
   }
   ...
```

Figure 5: Event loop using AIO

5.2-CURRENT which supports *KSE*, FreeBSD's scheduler activations implementation.

7 Microbenchmarks

In order to compare the cost of performing I/O using LAIO, non-blocking I/O, and AIO, we implemented a set of microbenchmarks. These microbenchmarks measured the cost of 100,000 iterations of reading a single byte from a pipe under various conditions. For 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_syscall()` 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 AIO as to which performed best.

8 Macrobenchmarks

We use web servers as our benchmark applications. We make two sets of comparisons. First, we compare the performance of single-threaded event-driven servers using different I/O libraries, in particular using non-blocking I/O, AIO and LAIO. Second, we compare the performance of an event-driven server augmented with helper processes to the performance of a single-threaded event-driven server using LAIO.

We use two existing servers as the basis for our experiments, thttpd [8] and Flash [7]. We modify these servers in various ways to obtain the desired experimental results. We first describe thttpd and Flash. We then document the changes that we have made to these servers. We conclude with a description of the workloads used in the experiments.

8.1 thttpd

thttpd has a conventional single-threaded event-driven architecture. It uses non-blocking network I/O and blocks on disk I/O. All sockets are configured in non-blocking mode. An event is received in the event loop when a socket becomes

```
client_write(struct client *clientp)
{
   ...
   /* initialize the control block */
   aiocbp->aio_fildes = clientp->socket;
   aiocbp->aio_buf = clientp->buffer;
   aiocbp->aio_nbytes = clientp->bytes_to_write;
   aiocbp->aio_sigevent.sigev_notify = SIGEV_NONE; /* do nothing; event loop polls
   ...                                              * for completion */
   /* initiate the operation; returns immediately */
   if (aio_write(aiocbp) == -1) {
      /* client_write_complete() handles errors */
      client_write_complete(clientp, -1, errno);
   } else {
      /* instruct event loop to call client_write_complete() upon completion
       * of the AIO operation; clientp is passed to client_write_complete() */
      event_set(&clientp->event, aiocbp, EV_AIO_COMPLETED,
         client_write_complete, clientp);
      event_add(&clientp->event, NULL);
      return; /* to the event loop */
   }
   ...
```

Figure 6: Event handler using AIO

ready for reading or writing, and the corresponding event handler is invoked. From these event handlers thttpd makes calls to operations that may block on disk I/O, which may in turn cause the entire server to block.

We use the following terminology to refer to the different versions of the servers. A version is identified by a triple server-network-disk. For instance, **thttpd-NB-B** is the thttpd server using non-blocking network I/O and blocking disk I/O, as described in the previous paragraph.

8.2 Flash

Flash employs the asymmetric multiprocess event driven (AMPED) architecture. It uses non-blocking I/O for networking, and helper processes for operations that may block on disk I/O. All potentially blocking operations like file open or read are handled by the helper processes. The event-driven core dispatches work to the helper processes through a form of non-blocking RPC [2]: after the event-driven core sends a request to a helper process, it registers an event handler for execution upon receipt of the response. Using the notation introduced in Section 8.1, we refer to this server as **Flash-NB-AMPED**.

We use the most recent version of Flash, which uses `sendfile()`. The event-driven core reads the requested URL after accepting a connection. The corresponding file is opened by a helper process. Then, the event-driven core uses `sendfile()` to write the file to the corresponding socket. If `sendfile()` blocks in the kernel on a disk I/O, it returns a special `errno`. The event-driven core catches this `errno`, and instructs a helper process to issue explicit I/O to bring the file data into memory. After the event-driven core receives the helper process's response, indicating that the file data is in memory, it re-issues the `sendfile()`. Flash thus performs I/O in a lazy manner, 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 processes.

8.3 Introducing LAIO

We modify thttpd to use the LAIO API, both for its network and disk operations. Blocking sockets are used for networking. All blocking system calls are invoked via `laio_syscall()`. Continuation functions are defined to handle `laio_syscall()` invocations that block and finish asynchronously. This version is called **thttpd-LAIO-LAIO**.

We first modify Flash to get a conventional single-threaded version. This version has no helper threads. Instead, all helper functions are called directly by the main thread. Non-blocking sockets are used as before to perform network I/O. We refer to this version as **Flash-NB-B**.

We then modify Flash-NB-B to use LAIO.

All sockets are configured as blocking, and potentially blocking operations are issued via `laio_syscall()`. This version is referred to as **Flash-LAIO-LAIO**.

8.4 Additional Versions for Comparison

We further modify Flash-NB-B to use AIO for file reads. This version of Flash uses sockets in a non-blocking mode. Since there is no AIO API for `stat()` and `open()`, it may block on those operations. We call this version of Flash **Flash-NB-AIO**.

We also modify Flash-NB-B to use LAIO for file I/O only. We call this version of Flash **Flash-NB-LAIO**.

Finally, we modify Flash-NB-AMPED to use kernel-level threads instead of processes to implement the helpers. Thus, the event-driven core and the helper threads share a single address space, reducing the cost of context switches between them. We call this version of Flash **Flash-NB-AMTED**.

8.5 Summary of Versions

The versions considered in the rest of the paper are summarized in Table 1.

8.6 Workloads

We use two trace-based web workloads for our evaluation. These workloads are obtained from the web servers at Rice University (Rice workload) and the University of California at Berkeley (Berkeley workload).

Table 2 shows the characteristics of the two workloads. The Rice and Berkeley workloads contain 245,820 and 3,184,540 requests, respectively. The columns *Small*, *Medium*, and *Large* indicate the percentage of the total number of bytes transferred in small files (size less than or equal to 8 Kilobytes), medium-sized files (size in between 8 Kilobytes and 256 Kilobytes), and large files (size greater than 256 Kilobytes). For the purposes of our experiments the most important characteristic is the working set of the workload: 1.1 Gigabytes for the Rice workload and 6.4 Gigabytes for the Berkeley workload. With the server machine we are using, which has 2 Gigabytes of main memory, the Rice workload fits in memory, while the Berkeley workload does not.

The trace file associated with each workload contains a set of request sequences. A sequence consist of one or more requests. Each sequence begins with a connection setup and ends with a connection teardown. Requests in a sequence are sent one at a time, the response is read completely, and then the next request is sent. In Flash, which supports persistent HTTP connections, all requests in a sequence are sent over a persistent connection. For thttpd, which does not support persistent connections, a new connection is set up before each request and torn down afterwards.

We use a program that simulates concurrent clients sending requests to the web server. The number of concurrent clients can be varied. Each simulated client plays the request sequences in the trace to the server. The program terminates when the trace is exhausted, and reports overall throughput and response time.

For each experiment we show a *cold cache* and a *warm cache* case. In the cold cache case, the cache is empty when the experiment is started. In the warm cache case, the cache is warmed up by running the entire experiment once before any measurements are collected.

9 Results

First, we compare single-threaded event-driven servers using different I/O libraries. Then, we compare LAIO to the Flash AMPED architecture with user-level helper processes.

9.1 Single-Threaded Event-Driven Servers

9.1.1 LAIO vs. Non-blocking I/O

Figure 7(a) shows the throughput for the Berkeley workload for thttpd-NB-B, thttpd-LAIO-LAIO, Flash-NB-B and Flash-LAIO-LAIO in both the cold and warm cache cases. thttpd-LAIO-LAIO achieves between 12% and 38% higher throughput than thttpd-NB-B, depending on the number of clients. Flash-LAIO-LAIO achieves between 5% and 108% higher throughput than Flash-NB-B. Figure 7(b) shows the response times for the four servers under the Berkeley workload. In both the cold and warm cache cases, thttpd-LAIO-LAIO and Flash-LAIO-LAIO achieve lower response times than thttpd-NB-B and Flash-NB-B, respectively. The Berkeley workload is too large to fit in memory. Thus, thttpd-NB-B and Flash-NB-B frequently

Server	Threaded	Blocking Operations	Comments
thttpd-NB-B	Single	disk I/O	stock version conventional event-driven
thttpd-LAIO-LAIO	Single		normal LAIO
Flash-NB-AMPED	Process-based Helpers		stock version multiple address spaces
Flash-NB-B	Single	disk I/O	conventional event-driven
Flash-LAIO-LAIO	Single		normal LAIO
Flash-NB-AIO	Single	disk I/O other than read/write	
Flash-NB-LAIO	Single		
Flash-NB-AMTED	Thread-based Helpers		single, shared address space

Table 1: Different versions of the servers

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 2: Web trace characteristics

block on disk I/O regardless of whether the cache is cold or warm, while thttpd-LAIO-LAIO and Flash LAIO-LAIO do not.

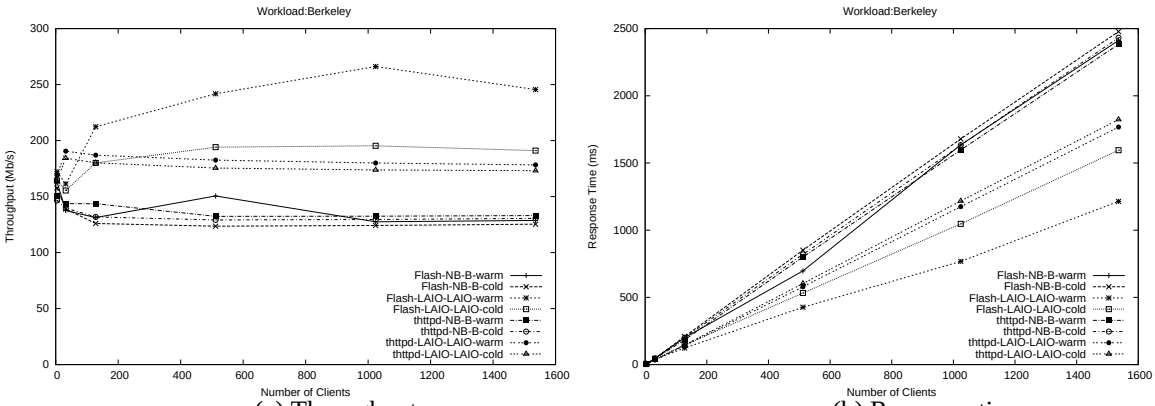
Figure 8(a) shows the throughput for the Rice workload for the same servers in both the cold and warm cache cases. For the cold cache case, thttpd-LAIO-LAIO achieves between 9% and 36% higher throughput than thttpd-NB-B, depending on the number of clients. Flash-LAIO-LAIO achieves between 12% and 38% higher throughput than Flash-NB-B. No gain is achieved in the warm cache case. Figure 8(b) shows the response times. Similarly, thttpd-LAIO-LAIO and Flash-LAIO-LAIO only achieve lower response times than thttpd-NB-B and Flash-NB-B, respectively, in the cold cache case. For this workload, which fits in memory, LAIO shows gains in throughput and response time only in the cold cache case, stemming from compulsory misses, which block in the absence of LAIO. In the warm cache case, there is no disk I/O, and therefore no improvement as a result of using LAIO.

We conclude that LAIO substantially improves the performance of conventional single-threaded servers using non-blocking I/O. As explained in Section 4.1, LAIO is also easier to program than non-blocking I/O.

9.1.2 Additional Comparisons

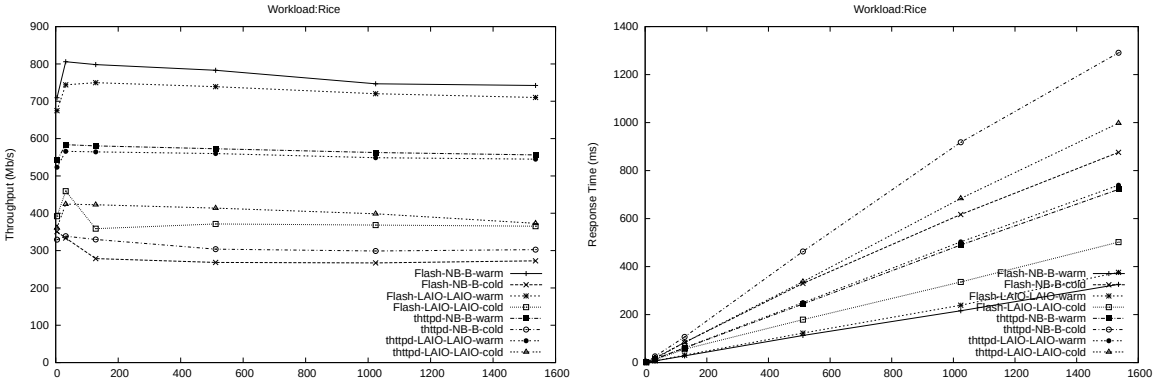
One may wonder whether even better performance results could be obtained by using non-blocking I/O for the network and LAIO for file I/O. Figures 9(a) and 9(b) show the throughput results for Flash-LAIO-LAIO and Flash-NB-LAIO, for the Berkeley and the Rice workloads, respectively. The results for thttpd and for the response times follow the same general trend, so from now on we only show throughput results for Flash. The throughputs of Flash-NB-LAIO and Flash-LAIO-LAIO are very close in all cases. Given that no performance improvements result from using non-blocking I/O for networking, and given the simpler programming model offered by LAIO, we argue that it is better to use LAIO uniformly for all I/O.

In our final experiment we measure the difference between using LAIO uniformly for all I/O versus using non-blocking sockets for network I/O and AIO for disk operations. Figures 10(a) and 10(b) show the throughput of Flash-NB-AIO and Flash-LAIO-LAIO for the Berkeley and Rice workloads, respectively. For the Berkeley workload, Flash-LAIO-LAIO dominates Flash-NB-AIO after 128 clients, achieving its largest improvement of 34% at 512 clients. This is because



(a) Throughput (b) Response time

Figure 7: Results for the Berkeley workload with single-threaded event-driven web servers



(a) Throughput (b) Response time

Figure 8: Results for the Rice workload with single-threaded event-driven web servers

AIO does not support asynchronous versions of `open()` and `stat()`. For the Rice workload, there is little difference between the two for either the cold or warm cache cases. In the cold cache case, the small number of files in the Rice workload results in relatively few of the operations that AIO does not support. In the warm cache case, the working set is in memory, so there is no disk I/O.

We conclude that the unified programming model offered by LAIO is preferable over mixed models combining non-blocking sockets and AIO or LAIO, both in terms of ease of programming and in terms of performance.

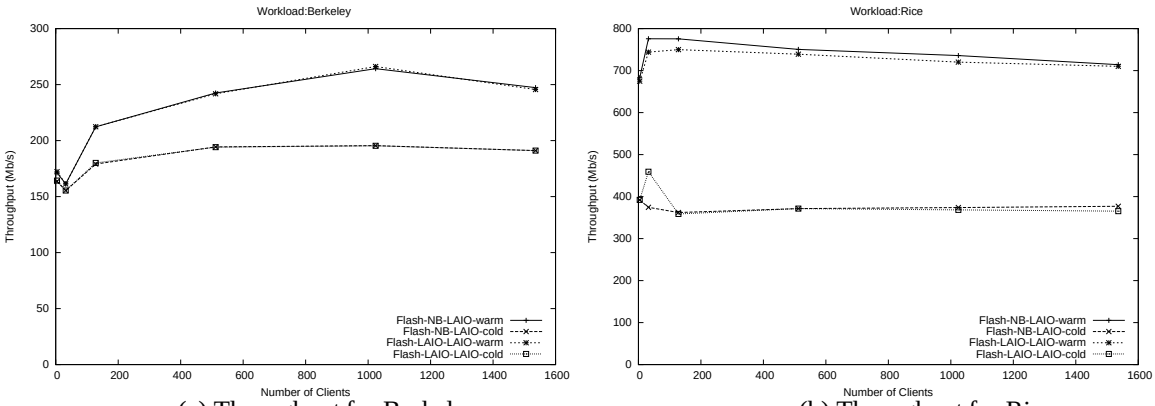
9.2 AMPED vs. Single-Threaded with LAIO

We compare the AMPED architecture with the event-driven architecture using LAIO for all I/O operations. First, we contrast these two architectures from the view point of programming complexity. Next, we compare the performance of these two architectures. We use the Flash web server for these results.

9.2.1 Programming Complexity of AMPED vs. LAIO

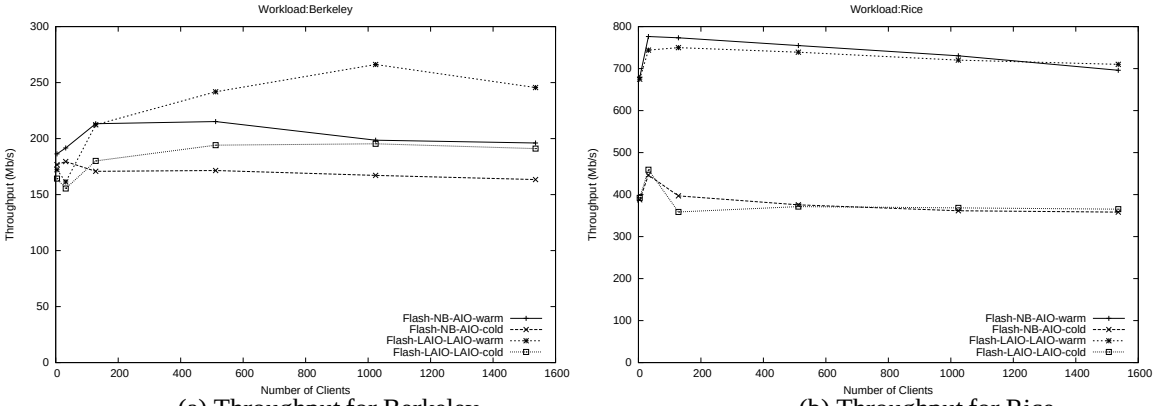
We compare the programming complexity of the AMPED architecture to the LAIO API. We use lines of code as a quantitative measure of programming complexity.

By using LAIO we avoid the coding complexities of using helper processes and communicating between the main process and helper processes. We also avoid the state maintenance for incomplete write operations on sockets. As a result the programming effort is much reduced while using LAIO. This is illustrated in Table 3 which shows the lines of code associated with the different components of Flash for Flash-NB-AMPED



(a) Throughput for Berkeley (b) Throughput for Rice

Figure 9: Results for Flash using non-blocking sockets and LAIO for disk operations



(a) Throughput for Berkeley (b) Throughput for Rice

Figure 10: Results for Flash using non-blocking sockets and AIO for disk operations

and Flash-LAIO-LAIO.

Flash-NB-AMPED has two helper processes, the read helper and the name conversion helper. The read helper issues a `read()` on the file from disk, while the name conversion helper does `stat()` on the file and path name and checks permissions. The read helper accounts for 550 lines of code in Flash-NB-AMPED. In Flash-LAIO-LAIO the read helper code results in less than 100 lines of code. The name conversion helper required more than 600 lines of code in Flash-NB-AMPED. The name conversion function requires less than 400 lines of code in Flash-LAIO-LAIO. Eliminating the partial-write state maintenance results in a saving of 70 lines of code in Flash-LAIO-LAIO.

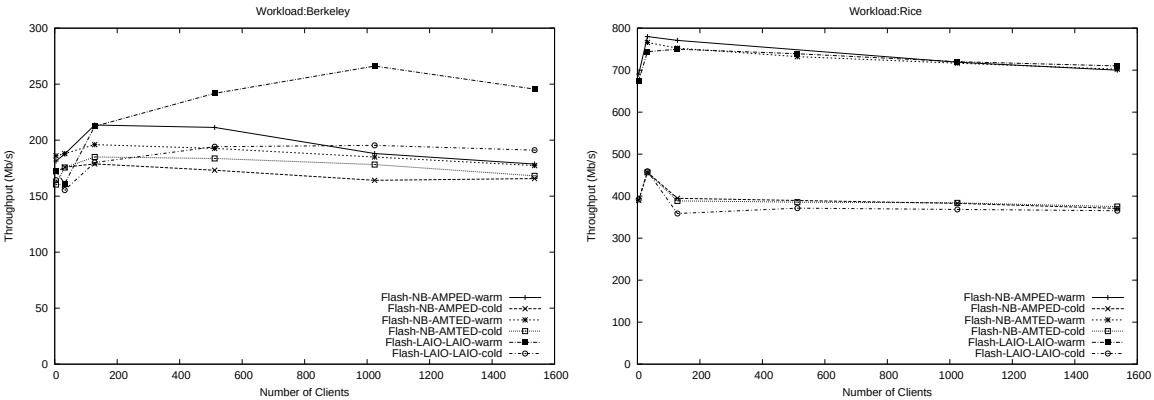
Considering the three components of Flash where Flash-NB-AMPED differs from Flash-LAIO-LAIO, Flash-NB-AMPED has more than three times the number of lines of code as Flash-LAIO-LAIO. Excluding comment lines and blank lines, Flash-NB-AMPED has about 8,860 lines of code in total. Flash-LAIO-LAIO has about 8,020 lines of code. So in Flash-LAIO-LAIO we have reduced the code size by almost 9.5%.

9.2.2 Performance Comparison

Figure 11(a) shows the performance of Flash-NB-AMPED, Flash-NB-AMTED and Flash-LAIO-LAIO for the Berkeley workload under cold and warm cache conditions. Flash-LAIO-LAIO outperforms both Flash-NB-AMPED and Flash-NB-AMTED by about 10% to 40%. There is no noticeable performance difference between Flash-NB-AMPED and Flash-NB-AMTED. The Berkeley workload does not fit in memory and the CPU is not the bottleneck for any of these servers. The better performance of Flash-LAIO-LAIO comes from its better disk utilization than Flash-NB-AMPED or Flash-NB-AMTED. We

Component	Flash-NB-AMPED	Flash-LAIO-LAIO
File read	550	15
Name conversion	610	375
Partial-write state maintenance	70	0
Total code size	8860	8020

Table 3: Lines of code count for different versions of Flash



(a) Throughput for Berkeley (b) Throughput for Rice

Figure 11: Results for Flash using the AMPED architecture

validate this by measuring the disk I/O statistics during a run of Flash-LAIO-LAIO and Flash-NB-AMPED under warm cache conditions. In particular, we measure total disk I/O (in bytes transferred) and average disk transfer rate (in bytes transferred per second). Flash-LAIO-LAIO transfers about 15.6 Gigabytes of data from the disk, which is about 15.7% lower than Flash-NB-AMPED which transfers about 18.5 Gigabytes. Flash-LAIO-LAIO achieves a disk transfer rate of 5.64 Mbytes/sec, which is about 5.8% higher than Flash-NB-AMPED which achieves a disk transfer rate of about 5.33 Mbytes/sec. Thus, Flash-LAIO-LAIO utilizes the disk more efficiently which translates into its higher performance. The disk I/O statistics for Flash-NB-AMTED are similar to that of Flash-NB-AMPED. Since the CPU was not the bottleneck, the context switching overhead (between address spaces) in Flash-NB-AMPED does not degrade its performance compared to Flash-NB-AMTED, and as such both these servers have almost the same performance.

Figure 11(b) shows the performance of Flash-NB-AMPED, Flash-NB-AMTED and Flash-LAIO-LAIO for the Rice workload under cold and warm cache conditions. Performance of these three servers is almost the same. The Rice workload fits in memory. Hence, there is no disk I/O with a warm cache and the three servers perform the same. There is disk I/O starting from a cold cache, but the amount is much smaller than for the Berkeley workload. Flash-NB-AMPED and Flash-NB-AMTED perform almost the same because the CPU is not the bottleneck in the cold cache case and no I/O is required by the helpers in the warm cache case.

We conclude that an event-driven server using LAIO outperforms a server using the AMPED (or AMTED) architecture when the workload does not fit in memory, and matches the performance of the latter when the workload fits in memory.

10 Related Work

Some prior work has been done in this area. Other than AIO, the Windows NT [4] and VAX/VMS [1] operating systems have provided asynchronous I/O.

In Windows NT, the application can start an I/O operation then do other work while the device completes the operation. When the device finishes transferring the data, it interrupts the application's calling thread and copies the result to its address space. The kernel uses a Windows NT asynchronous notification mechanism called

asynchronous procedure call (APC) to notify the application's thread of the I/O operation's completion.

Like NT, VAX/VMS allows for a process to request that it gets interrupted when an event occurs, such as an I/O completion event. The interrupt mechanism used is called an asynchronous system trap (AST), which provides a transfer of control to a user-specified routine that handles the event.

Similar to AIO, asynchronous notifications in both VAX/VMS and Windows NT are limited to a few events, mainly I/O operations and timers. This is not broad enough to support any system call like in LAIO. Also asynchronous I/O in either Windows NT or VAX/VMS is not lazy.

11 Conclusions

We have introduced Lazy Asynchronous I/O (LAIO), a new asynchronous I/O interface that is well suited to event-driven programming, particularly the implementation of high-performance servers. LAIO is general in that it supports all system calls, and lazy in the sense that it only creates a continuation if the operation actually blocks. In addition, it provides notification of completion rather than partial completion.

LAIO overcomes the limitations of previous I/O mechanisms, both in terms of ease of programming and performance. We have demonstrated this claim by comparing LAIO to non-blocking I/O, AIO and AMPED.

By means of an example we have shown the programming advantages of LAIO over the alternatives. Furthermore, we have quantified the comparison between LAIO, non-blocking I/O, and AMPED by counting the affected lines of code within the Flash web server. Flash-LAIO-LAIO, a version of Flash using LAIO for both networking and disk, has 9.5% fewer lines of code than Flash-NB-AMPED, the stock version of Flash using a combination of non-blocking I/O for networking and AMPED for disk.

We have experimented with two web servers, thttpd and Flash, to quantify the performance advantages of LAIO. We have shown that for workloads which cause disk activity LAIO outperforms all alternatives, because it avoids blocking in all circumstances and because it has low overhead in the absence of blocking. For one such workload, Flash-LAIO-LAIO achieved a peak throughput that was 25% higher than Flash-NB-AMPED and 24% higher than the next best event-driven version, Flash-NB-AIO, using non-blocking I/O for networking and AIO for disk. Under workloads with no disk activity, there was little difference in throughput among the servers.

References

[1] *VAX Software Handbook*. Digital Equipment Coroporation, 1981.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, pages 143–154, June 1998.

[3] 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.

[4] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[5] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *2002 USENIX Annual Technical Conference*, pages 103–114, June 2002.

[6] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation at the 1996 USENIX Annual Technical Conference, Jan. 1996.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[13] 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.

[14] N. Williams. An Implementation of Scheduler Activations on the NetBSD Operating System. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 99–108, June 2002.