Lazy Asynchronous IO for Event Driven Servers
(by Khaled, Anupam and Alan Rice University, Houston, Texas)
Presenter: Emalayan
Agenda

- Contribution
- Different Types of IO
- Non Blocking IO
- AIO
- LAIO
- LAIO implementation
- Workload Characteristics
- Performance Results
- Conclusion
Contribution

- LAIO (Lazy asynchronous input and output)
  - Non blocking counter part of each blocking calls
  - Continuation only when operation blocks
Type of IO

- Non Blocking IO
  - mainly using poll
  - partial completion and maintaining state
  - limited to network IO only

- AIO
  - OS may or may not provide a signal to the application
  - Continuation always there

- LAIO
  - Asynchronous
  - Continuation only when operation blocks
Non Blocking IO with blocking notifications (poll/select)
Event loop using non-blocking IO

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--;
        }
    }
    ...

Event handler using non-blocking I/O

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);
...
Asynchronous I/O (AIO)
Event loop using AIO

```c
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 */
    }
    /* handle error */
    ...
```
Event handler using AIO

```c
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 */
    ...
    /* 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 */
    }
    ...
```
LAIO Implementation

- Some of the following slides are extracted from the authors presentation.
  (http://www.cs.rice.edu/~kdiaa/laio/)
## LAIO API

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Function Name</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>laio_syscall</td>
<td>int number,...</td>
</tr>
<tr>
<td>void*</td>
<td>laio_gethandle</td>
<td>void</td>
</tr>
<tr>
<td>int</td>
<td>laio_poll</td>
<td>laio_completion[] completions, int ncompletions, timespec* ts</td>
</tr>
</tbody>
</table>
laio_syscall() – Non-blocking case

- Save context
- Enable upcalls

Issue operation

System call blocks?

No

- Disable upcalls
- Return retval
laio_syscall() – Blocking case

Application

laio_syscall()

- Save context
- Enable upcalls

Issue operation

System call blocks?

Yes

Library
laio_syscall() – Blocking case

Application

laio_syscall()

• Save context
• Enable upcalls

Issue operation

Library

System call blocks?

Yes

Kernel

Background laio_syscall

Upcall on a new thread
laio_syscall() – Blocking case

- Save context
- Enable upcalls

Application

Issue operation

System call blocks?

Yes

Library

Kernel

Background laio operation

Upcall on a new thread

Library

upcall handler

Steals old stack using stored context
laio_syscall() – Blocking case

1. Application
   - laio_syscall()
   - • Save context
   - • Enable upcalls

2. Issue operation
   - System call blocks?
     - Yes
     - Background laio operation

3. Library
   - • Disable upcalls
   - • errno = EINPROGRESS
   - • Return -1

4. Upcall handler
   - Steals old stack using stored context

5. Kernel
   - Upcall on a new thread
Unblocking case

- List of completions is retrieved by the application using `laio_poll()`

upcall handler()  
Library

- Construct completion structure:
  - `laio operation handle`
  - `System call return value`
  - `Error code`
- Add completion to list of completions

Kernel

- Background `laio operation` completes, thread dies
- Upcall on the current thread
Event loop using LAIO

```c
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 */
    }
    ...
```
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);
    }
    ...
}
Workload Characteristics

<table>
<thead>
<tr>
<th>Web Workload</th>
<th>No. of requests</th>
<th>Small (≤ 8 KB)</th>
<th>Medium (&gt; 8 KB and ≤ 256 KB)</th>
<th>Large (&gt; 256 KB)</th>
<th>Total footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rice</td>
<td>245,820</td>
<td>5.5%</td>
<td>20.2%</td>
<td>74.3%</td>
<td>1.1 Gigabytes</td>
</tr>
<tr>
<td>Berkeley</td>
<td>3,184,540</td>
<td>8.2%</td>
<td>33.2%</td>
<td>58.6%</td>
<td>6.4 Gigabytes</td>
</tr>
</tbody>
</table>

- **Warm cache**
  - Once entire experiment was done
- **Cold Cache**
  - Cache is empty
# Different version of Servers

<table>
<thead>
<tr>
<th>Server</th>
<th>Threaded</th>
<th>Blocking Operations</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>thttpd-NB-B</td>
<td>Single</td>
<td>disk I/O</td>
<td>stock version conventional event-driven</td>
</tr>
<tr>
<td>thttpd-LAIO-LAIO</td>
<td>Single</td>
<td></td>
<td>normal LAIO</td>
</tr>
<tr>
<td>Flash-NB-AMPED</td>
<td>Process-based Helpers</td>
<td></td>
<td>stock version multiple address spaces</td>
</tr>
<tr>
<td>Flash-NB-B</td>
<td>Single</td>
<td>disk I/O</td>
<td>conventional event-driven</td>
</tr>
<tr>
<td>Flash-LAIO-LAIO</td>
<td>Single</td>
<td></td>
<td>normal LAIO</td>
</tr>
<tr>
<td>Flash-NB-AIO</td>
<td>Single</td>
<td>disk I/O other than read/write</td>
<td></td>
</tr>
<tr>
<td>Flash-NB-LAIO</td>
<td>Single</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash-NB-AMTED</td>
<td>Thread-based Helpers</td>
<td></td>
<td>single, shared address space</td>
</tr>
</tbody>
</table>
Results: NB –AIO Vs LAIO - LAIO

(a) Throughput for Berkeley
Results with AMTED, AMPED
Conclusion

- LAIO outperforms the other servers when the workload does not fit into the memory
- No state maintenance, Lesser code
- No performance improvements
  - both LAIO & Non-blocking IO is same
  - But LAIO offers simpler programming model
THANK YOU
Questions

- While the authors suggested that there is benefit to have no partial completion for a blocking operation, I do think it may also make sense some time we may want partial completion (at least we may some time want to know the progress of the operation), is it possible to create partial completion in LAIO?

- A random thought, can LAIO be implemented without scheduler activation?
Questions

- While their results look good, the authors do not push any of the benchmarks to performance degradation (the graphs are all essentially straight lines). Is there something about LAIO that would make it perform worse under heavy load, or is it just an uninteresting case?

- In event based systems blocking I/O operations seem serve as a somewhat logical partition of a task, making it easy to 'figure out where you are' in a task once an operation completes. It seems that the ability perform any blocking syscall asynchronously might complicate this. How much would switching to AIO complicate the situation of a event based system in practise?
Questions

- Just like in capriccio, it seems that a page fault will pause execution. While I agree that this is the sane thing to do, is there any way we could make an event-based system 'tolerate' page-faults? or is that just insane/gross
Questions

- This paper is full of win (lazy stuff, events only on completion) but it sadly mentions that LAIO requires Scheduler Activations. Can it be made to work on top of kernel having 1:1 thread model (kernel space threads)? If yes, then discuss a possible solution.

- Non-blocking I/O works with sockets, AIO works with disks and many I/O related system calls don't have asynchronous versions. This motivated the work on LAIO. Why not fix the kernel itself (once and for all)? By fix I mean, build a unified asynchronous API which does everything, write asynchronous support for I/O system calls?
Questions

- In the paper, authors said that AIO and non-blocking IO always creates a continuation, regardless of whether the call blocks or not, however, LAIO creates a continuation only when an blocking really happens. But it seems quite straightforward to doing so. So what is the limitation of AIO and non-blocking IO to provide this feature?

- Their benchmarks seem to be unfair. In the real runtime multicore system, which system call happens in a reasonable degree, is it still necessary for this LAIO?
Questions

- "Non-blocking I/O can only be performed on network connections. AIO can be performed on basic disk files" Why not extend them to do more I/O operations by extending the functions they call?

- Will the use of scheduler activation in LAIO break the abstraction in event-based system, and make the situation more complex when we use multi-cores?
Questions

- If LAIO is used in Multi-threads, due to the thread interleaving, will laio_sycall and laio_gethandle match incorrectly? In other word, will the variable "background_handle" work in a multi-thread environment?

- In the first experiment (figure 7 & 8) "non-blocking vs LAIO", I think the experiment setup cannot support the conclusion that "LAIO substantially improves the performance of conventional single-threaded servers using non-blocking I/O" because the non-blocking I/O version (flash-NB-B, thttpd-NB-B) will block on the disk I/O. The performance gain of LAIO may come from the disk I/O. And in fact, in figure 9, flash-NB-LAIO and flash-LAIO-LAIO have the same trend and performance. That says, LAIO cannot substantially outperform non-blocking I/O.
Questions

There is two types of workload in the experiment. I think they are not enough because the two types of workloads incur different results in all experiments. Therefore, more workloads are needed to prove the authors' point. Otherwise, LAIO's performance can not convince us to use it. After all, LAIO needs kernel support (scheduler activation).

LAIO done a great work in easing the programmers work. However it seems that gaining an advantage over other approaches in this case, to some extent depends on the nature of the execution and the way the program is coded. This is because, even in LAIO one can not access the data until the call is completed so program has to wait until the call is done, if the program do not have any other work other than manipulating the data from a specific call.