Lazy Asynchronous I/O For Event-Driven Servers

Khaled Elmeleegy, Anupam Chanda and Alan L. Cox
Department of Computer Science
Rice University, Houston, Texas.

Willy Zwaenepoel
School of Computer and Communication Sciences
EPFL, Lausanne, Switzerland.

Presented By: Anirban Sinha (aka Ani), anirbans@cs.ubc.ca

About the Authors

- Alan L. Cox (http://www.cs.rice.edu/~alc/)
  - Associate Professor, mainly into Distibuted Systems, Concurrent Programming, Parallel Processing etc. Received PhD from University of Rochester.
- Willy Zwaenepoel (http://www.cs.rice.edu/~willy)
  - Also in Rice University currently, received his PhD from Stanford.
  - Somehow this paper is not listed in his homepage in the list of publications.
  - http://www.cs.rice.edu/~willy/publications.html
- Khaled Elmeleegy & Anupam Chandra
  - Both currently PhD students at Rice, both had their masters in the year 2003 under Alan & Willy. This paper is probably done during the time they were working on their masters thesis.

Outline

- The Problem.
- The Proposed Solution:
  - Lazy Asynchronous I/O (LAIO)
  - LAIO Implementation.
- Evaluation & Results.
- Conclusions
  - Analysis of the paper.

Problem

- Event Driven Servers must avoid blocking on I/O, resource allocation etc.
- Unix Like Systems have non-blocking I/O that can be performed only on network sockets, not files.
- POSIX AIO supports asynchronous I/O on only disk read & write, no other operations supported.
- We need to have a common all purpose asynchronous IO library.

The Solution
Lazy Asynchronous I/O (LAIO)

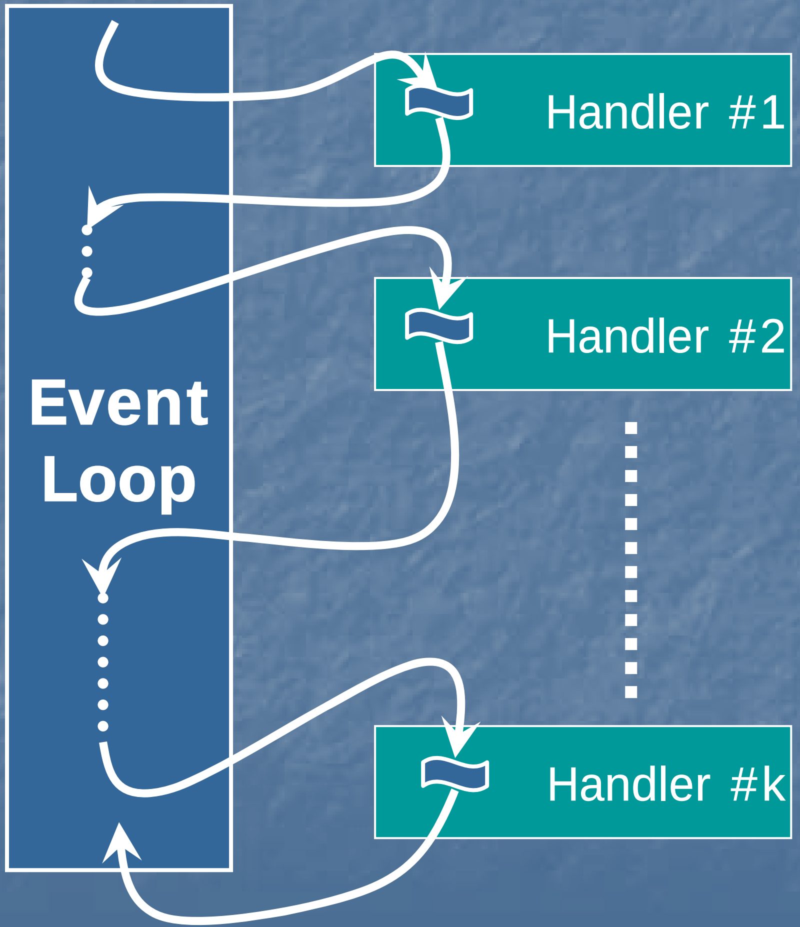
- Addresses problems with non-blocking I/O
  - Universality
    - Covers all I/O operations.
  - Simplicity
    - Requires less code.
  - Is Lazy, does asynchronous operation ONLY where required, falls back to older library system call when no blocking takes place.
  - Implemented fully in user level library
    - No modification to kernel.
  - LAIO notifies the application AFTER the event completes, not at any intermediate stage.

Why Lazy?

- Most potentially blocking operations don’t actually block.
  - Experiments: 73% - 86% of such operations don’t block
- Reduces overhead for those operations that do not really block.

Event-Driven Servers

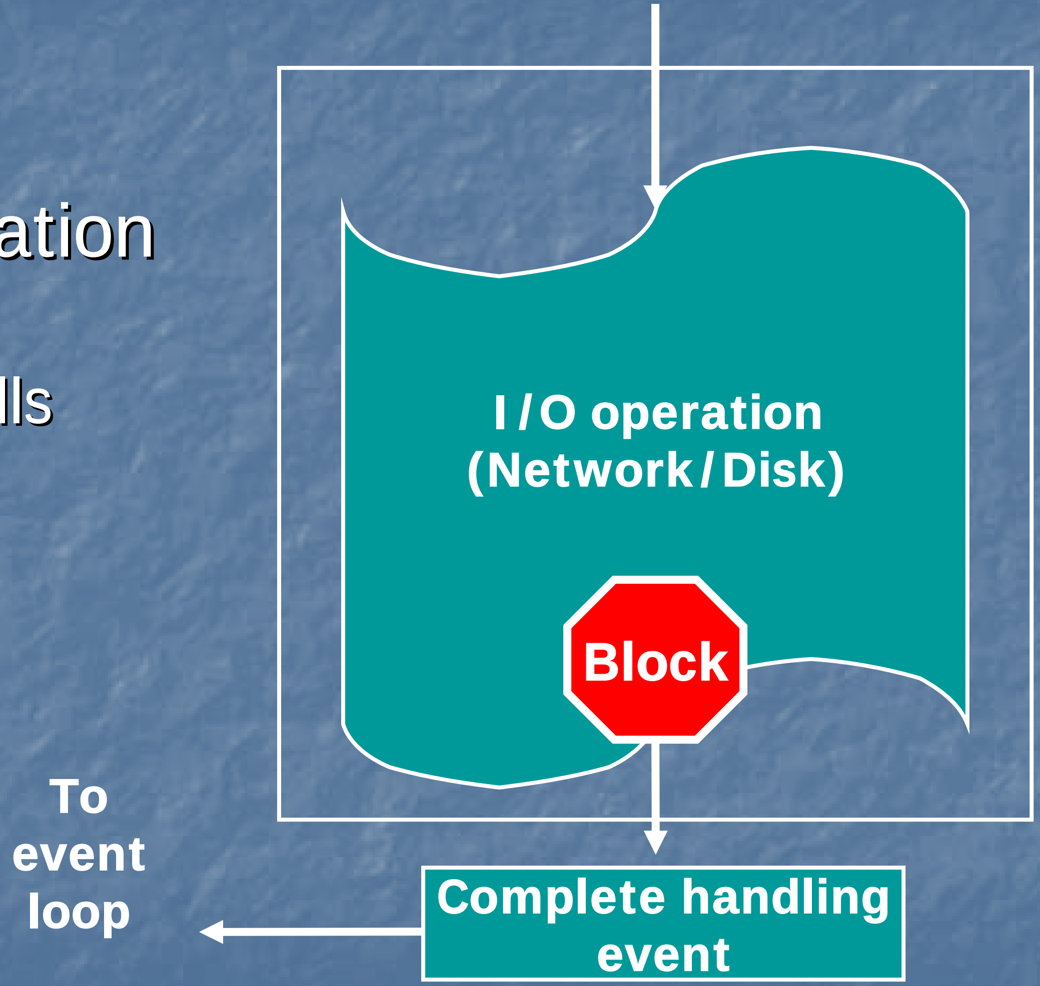
- Event loop processes incoming events
- For each incoming event, it dispatches its handler
- Single thread of execution



Slide taken from original presentation slide by authors

Event Handler

- If the I/O operation blocks
  - The server stalls



Slide taken from original presentation slide by authors

THE LAIO API

- LAIO Library consists of three functions:
  - int laio_syscall(int num, ...)
    - wrapper around the original syscall()
  - void* laio_gethandle(void)
  - int laio_poll (laio_completion[] completions, int ncompletions, timespec* ts)

laio_syscall()

- Lazily converts any system call into an asynchronous call

  If (! block) {
  - laio_syscall() returns immediately
  - With return value of system call

  } else if (block) {
  - laio_syscall() returns immediately
  - With return value -1
  - errno set to EINPROGRESS
  - Background LAIO operation

  }

laio_syscall()

- Lazily converts any system call into an asynchronous call

  If (! block) {
  - laio_syscall() returns immediately
  - With return value of system call

  } else if (block) {
  - laio_syscall() returns immediately
  - With return value -1
  - errno set to EINPROGRESS
  - Background LAIO operation

  }

Wow!!!

laio_gethandle()

- If (block) {

    Returns a handle representing the last issued LAIO operation

  }

  else {

    NULL is returned

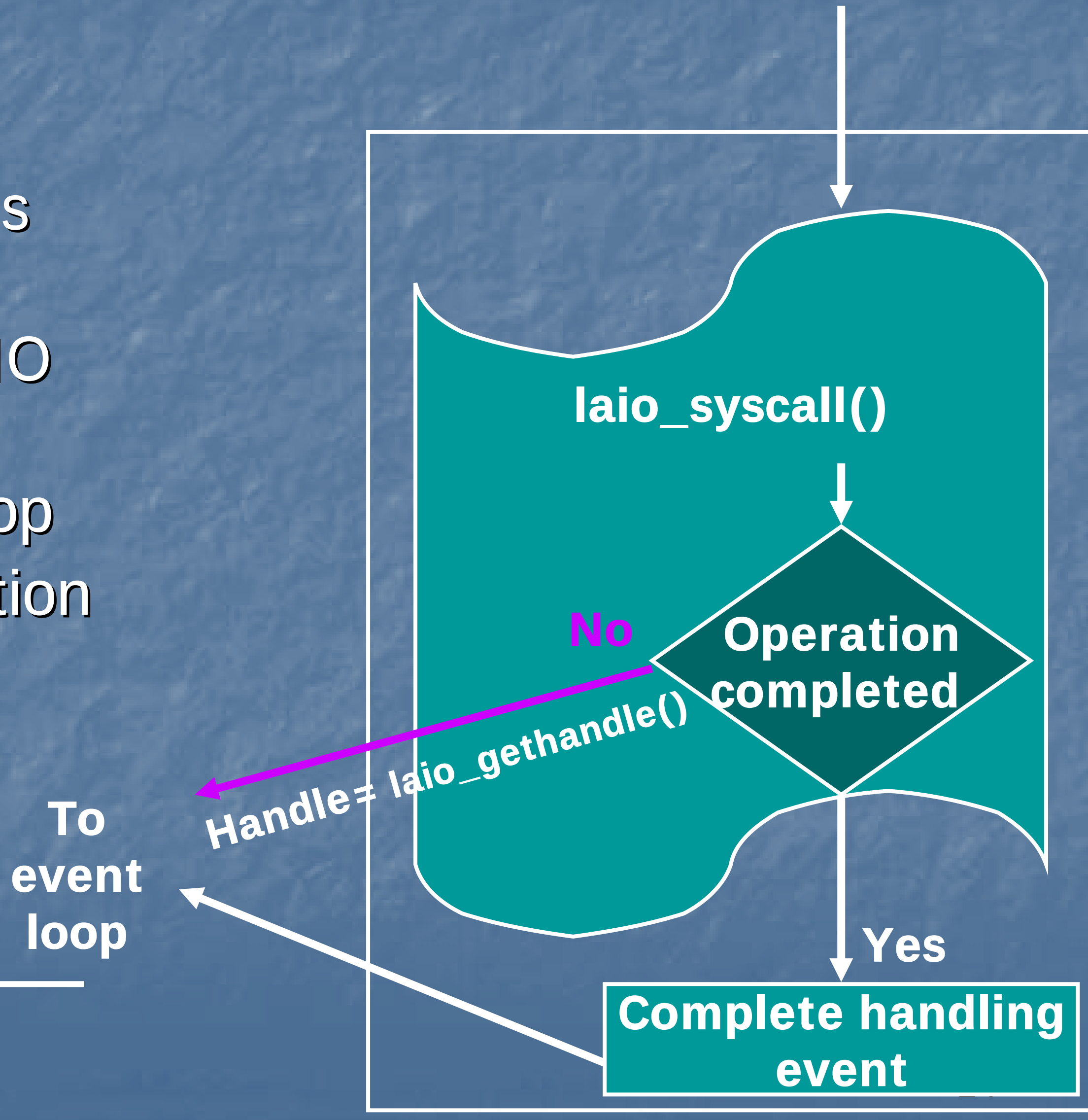
  }

laio_poll()

- Waits for the completion of background laio_syscall() blocking operation.
- Returns a count of completed background LAIO operations.
- Fills an array with completion entries within the timeout interval.
  - One for each blocking operation.
- Each completion entry has
  - Handle
  - Return value
  - Error value

Event Handler With LAIO

- If operation blocks
  - laio_syscall() returns immediately
  - Handler records LAIO handle
  - Returns to event loop
  - Completion notification arrives later



Slide taken from original presentation slide by authors

The Event Loop in LAIO

```
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 */
   }
   ...
```

Event Handler in 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);
   ...
```

LibEvent- A Event Notification Library

http://monkey.org/~provos/libevent/

- We use three methods from this library
  - event_set()
    - Event Initialization
  - event_add()
    - Monitoring of this initialized event; has to be done explicitly except for persistent events.
  - event_del()
    - Event Deletion.
- All these methods work with event objects with three attributes
  - object being monitored, like a socket.
  - Desired state of the object when the event triggers, like data availability in socket.
  - The event handler itself.

What Happens with Completion Objects??

- With each completion object, event loop has to locate each associated event object.
- Call the continuation function stored in the event object with the returned arguments in the completion object.

Outline

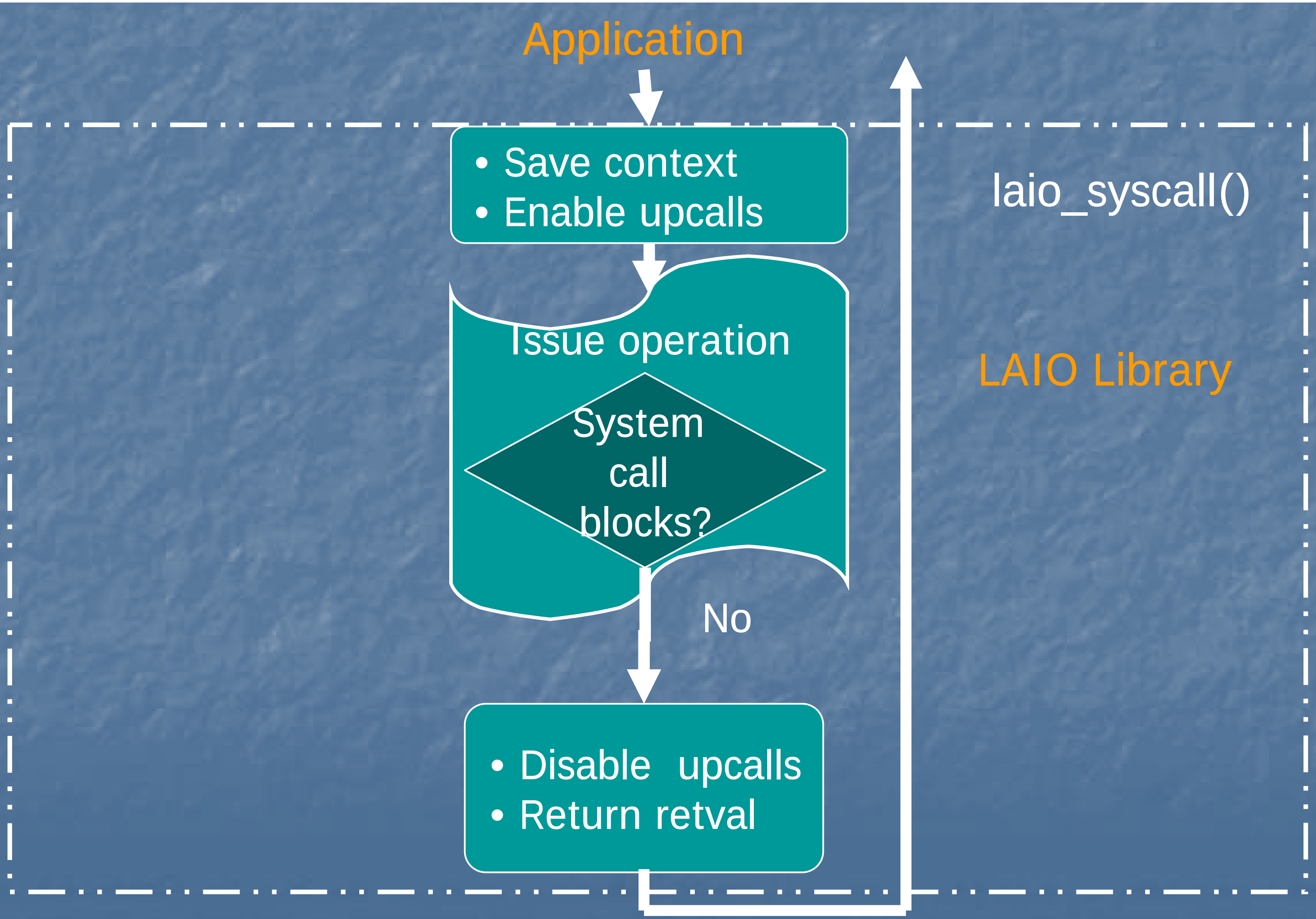
- The Problem. ✓
- The Proposed Solution:
  - Lazy Asynchronous I/O (LAIO) ✓
  - LAIO Implementation.
- Evaluation & Results.
- Conclusions
  - Analysis of the paper.

LAIO Implementation

- LAIO requires scheduler activations.
- Scheduler activations
  - The kernel delivers an upcall when an operation
    - Blocks - laio_syscall()
    - Unblocks - laio_poll()

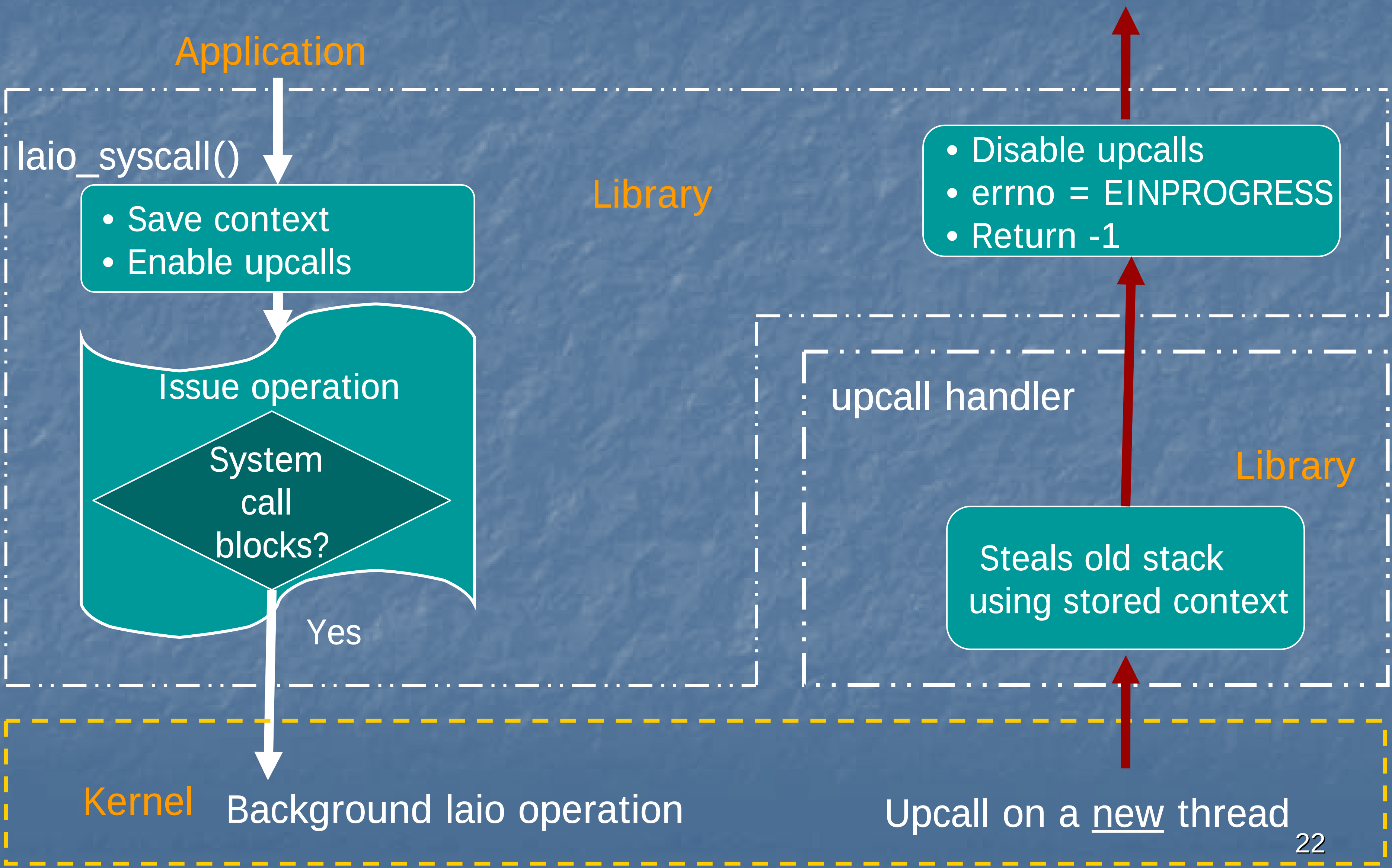
LAIO Implementation
laio_syscall() – Non-blocking case

Diagrams in this slide taken from the authors' presentation slides



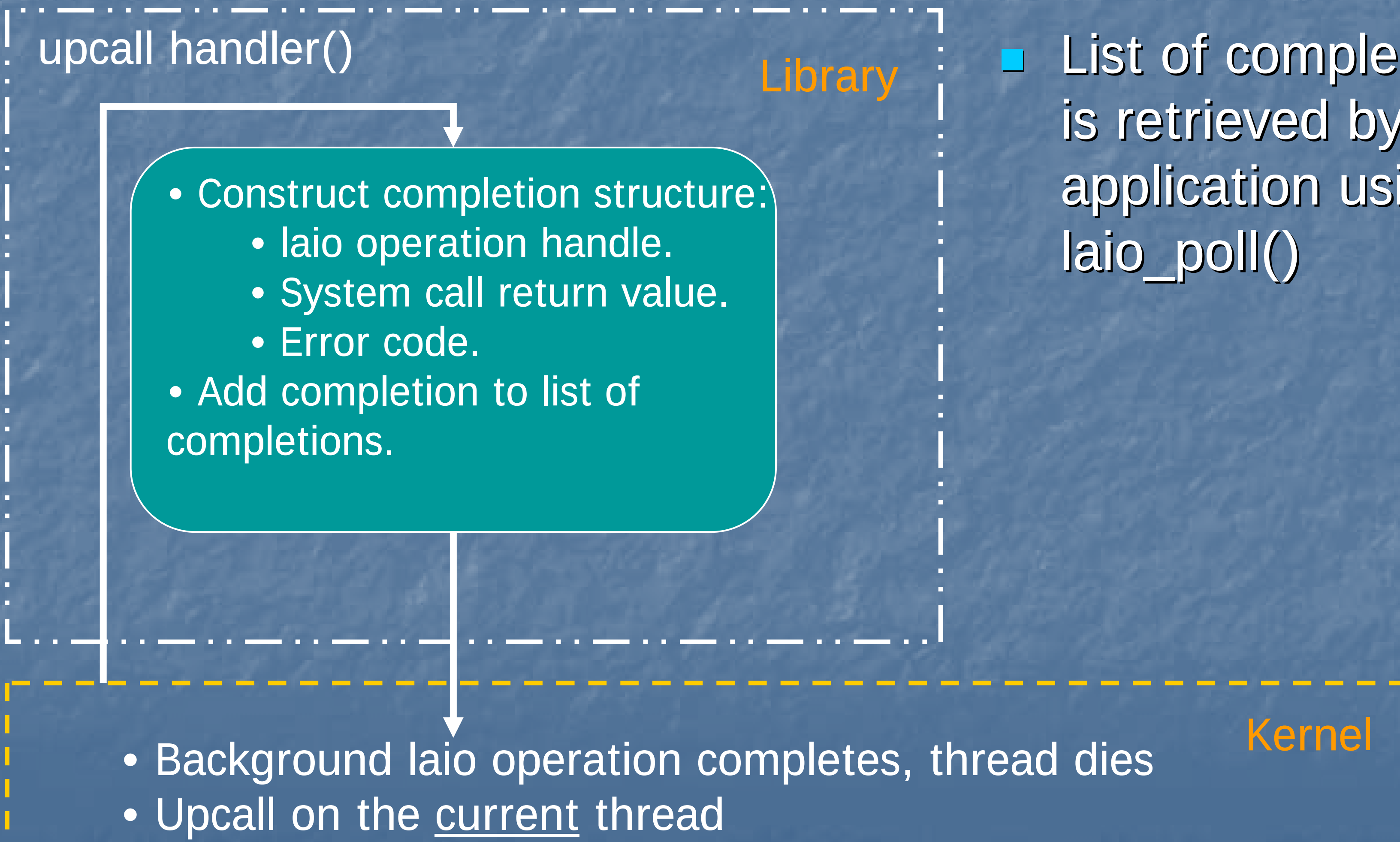
laio_syscall() – Blocking case

Diagrams in this slide taken from the authors' presentation slides



When timeout occurs ...

Diagrams in this slide taken from the authors' presentation slides



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

Outline

- The Problem. ✓
- The Proposed Solution:
  - Lazy Asynchronous I/O (LAIO) ✓
  - LAIO Implementation. ✓
- Evaluation & Results.
- Conclusions
  - Analysis of the paper.

Evaluation

- Micro Benchmark
  - Reading a single byte through pipes, 100,000 times both when pipe was full & when empty.
  - Eliminated the redundant times of disk access.
  - When full, no blocking I/O took place, LAIO was 1.4 slower than non-blocking I/O & AIO was even slower than LAIO.
  - When empty, LAIO was a factor of 1.08 slower than AIO.
  - Slowness, I guess can be attributed to the extra logic that is added to check whether an I/O actually blocks – the price of being LAZY !!!!!.

Evaluations - Macrobenchmarks

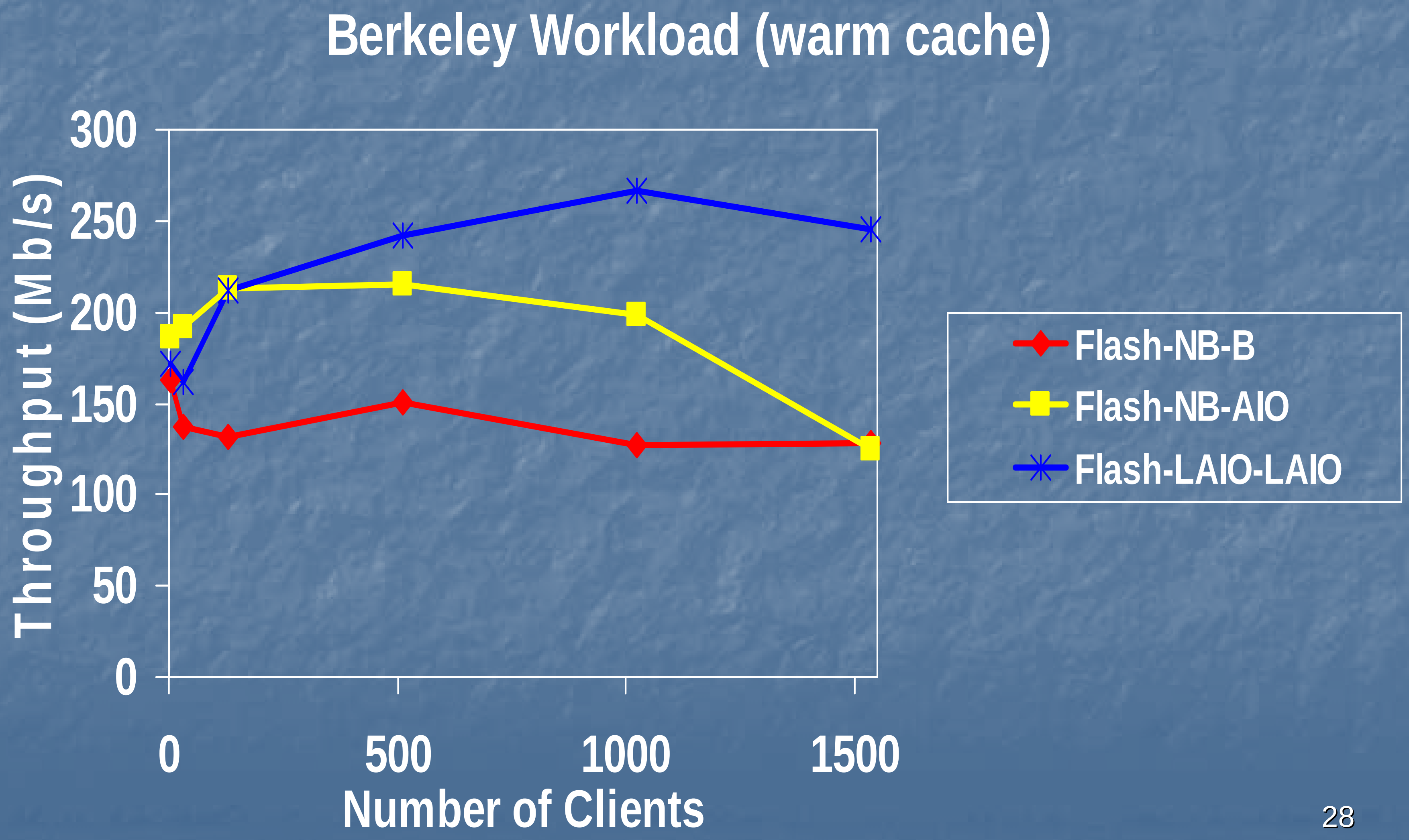
- Flash web server & thttpd web server
  - Each of them modified to use AIO, LAIO & Non-Blocking IO.
- Intel Xeon 2.4 GHz with 2 GB memory.
- Gigabit Ethernet between machines.
- FreeBSD 5.2-CURRENT.
- Two web workloads
  - Rice 1.1 GB footprint – fits in server memory.
  - Berkeley 6.4 GB footprint – oops! Does not fit!
- Two test cases for each workload
  - Cold Cache – when cache is previously empty.
  - Warm cache – when cache is previously full.

Summary of Modified Webservers

Server-Network-Disk	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

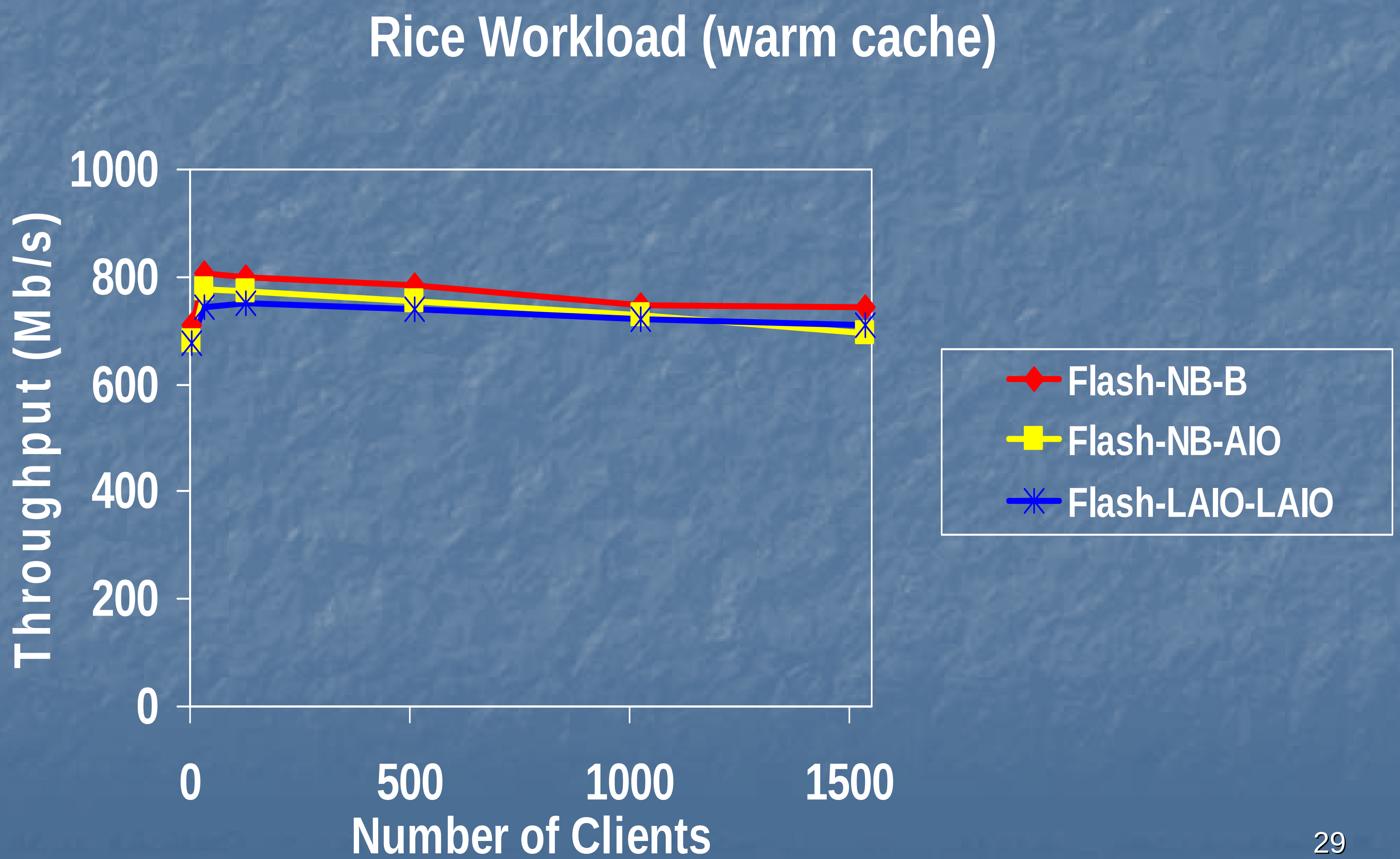
Performance: Berkeley Workload

Diagrams in this slide taken from the authors' presentation slides



Performance: Rice Workload

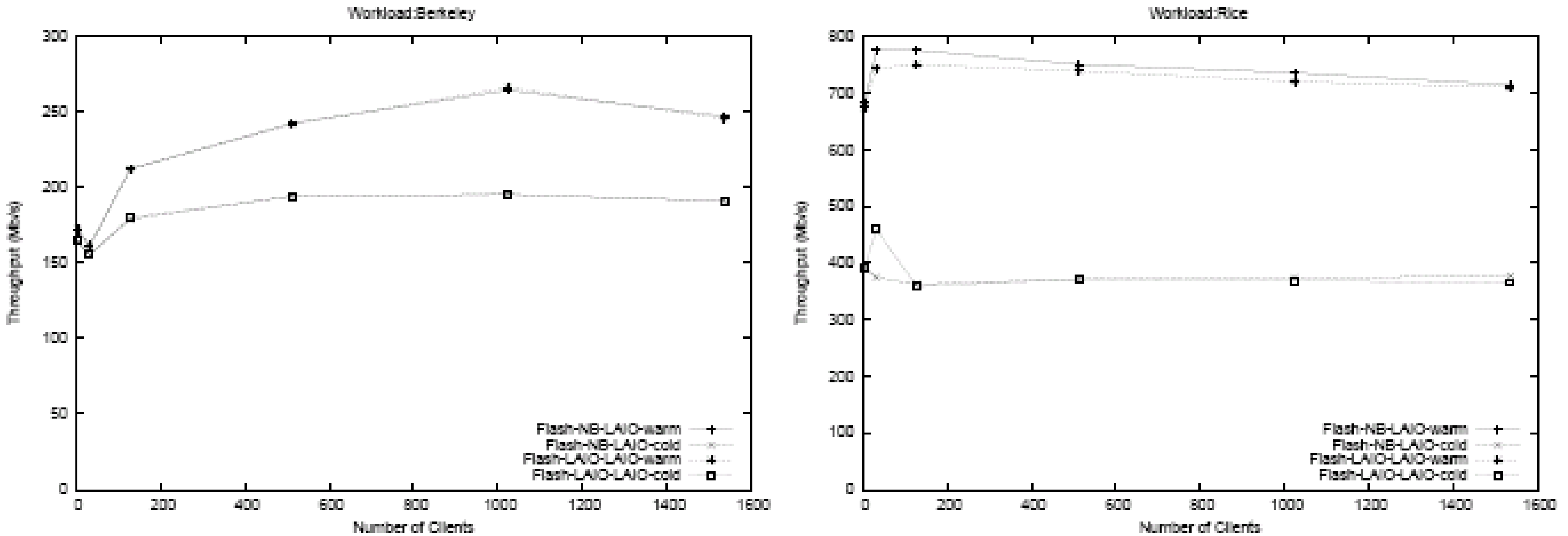
Diagrams in this slide taken from the authors' presentation slides



Inference from Figures

- LAIO performs better in Berkeley workload both in cold & warm cases.
  - The workload does NOT fit in memory, so blocking on I/O is inevitable.
  - Response time accordingly falls.
- LAIO performs poorly in Rice warm case
  - No blocking I/O occurs, program entirely in memory
  - Response time poor.
- LAIO gains in cold cache case with rice workload
  - Compulsory misses during initial stages – blocking.

Is it OKAY to use NB for network & LAIO for disk?



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

- No significant gain in using flash-NB-LAIO.
- Conclusion – USE LAIO for both.

Compare: LAIO vs. AMPED

Server-Network-Disk	Threaded	Blocking operations
Flash-LAIO-LAIO	Single	None
Flash-NB-AMPED	Process-based helpers	None

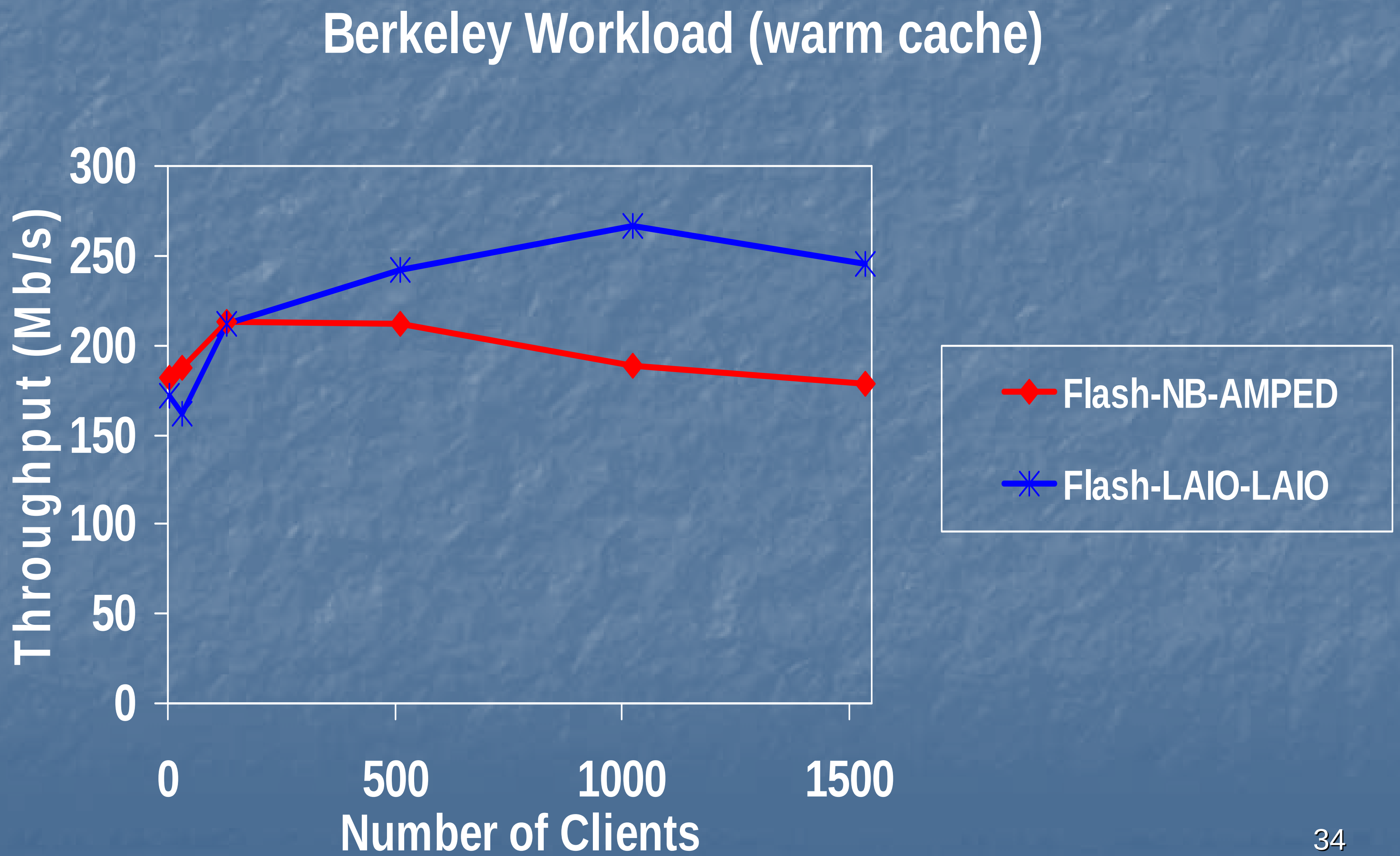
Slide taken from original presentation slide by authors

AMPED

- Asymmetric multiprocess event-driven.
- Simulates asynchronous behaviour by submitting blocking IO operations to a pool of threads – helper threads.

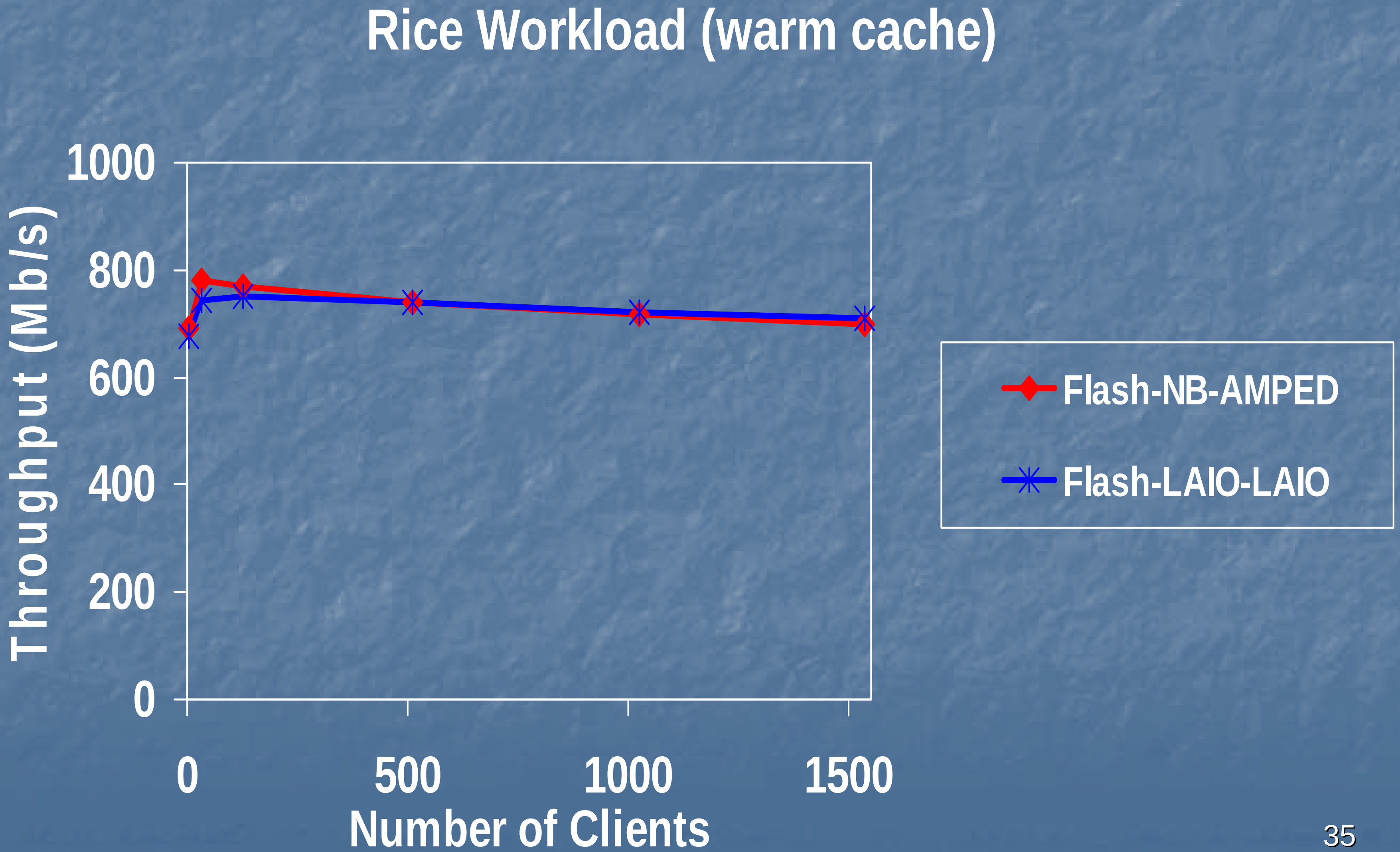
Performance of LAIO vs. AMPED

Diagrams in this slide taken from the authors' presentation slides



Performance of LAIO vs. AMPED

Diagrams in this slide taken from the authors' presentation slides



COMPARE LOC: AMPED VS LAIO

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

9.5% reduction in lines of code

Outline

- The Problem. ✓
- The Proposed Solution:
  - Lazy Asynchronous I/O (LAIO) ✓
  - LAIO Implementation. ✓
- Evaluation & Results. ✓
- Conclusions
  - Analysis of the paper.

Conclusions

- LAIO provides uniform platform.
  - Supports all system calls.
- LAIO is also simpler.
  - Used uniformly.
  - No state maintenance.
  - No helpers.
  - Less lines of code.

Analysis

- Weaknesses
  - No analysis in the paper to show that being LAZY is really necessary & fruitful.
  - Why would people really care about LOC once we already build LAIO library?
  - “Flash LAIO-LAIO utilizes disk more efficiently”, thus outperforms flash-NB-AMPED but HOW??? Not addressed.
  - Is there a way to increase the response time for LAIO ??? – Suggestions??
- Strengths
  - Addresses a pertinent problem.
  - Good analysis, taking all different test cases.
  - Considers all possible available present day alternatives.

Questions & Discussions ...