

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 Sam Davis, sddavis@cs.ubc.ca

LAIO API

Return Type	Function Name	Parameters
int	laio_syscall	int number,...
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 the system call doesn't block
 - laio_syscall() returns immediately
 - With return value of system call
 - Else if it blocks:
 - laio_syscall() returns immediately
 - With return value -1
 - errno set to EINPROGRESS
 - Background LAIO operation

LAIO is General

laio_syscall wraps **ANY** system call:

- `laio_syscall(WRITE)`
- `laio_syscall(READ)`
- `laio_syscall(OPEN)`
- `laio_syscall(SLEEP)`
- Etc.

laio_gethandle()

- Returns a handle representing the last issued LAIO operation
 - If operation didn't block, NULL is returned

laio_poll()

- Returns a count of completed background LAIO operations
- Fills an array with completion entries
 - One for each operation
- Each completion entry has
 - Handle
 - Return value
 - Error value

Structure of Event Handler Using **NBIO**

Returns immediately (in theory)

call **write()** to attempt to write some bytes

IF an error occurred OR all requested bytes were written

tell the event loop that this I/O operation is complete

call a (user defined) function that handles I/O completion and errors

ELSE

make sure event loop will watch for completion of this I/O operation

give **the socket** for this operation to the event loop and tell it:

- the function to call when more data can be written without blocking
- the parameter that should be passed to that function
- the total number of bytes written so far (sum over all calls to **write**)
- the number of bytes remaining to be written

RETURN to the event loop

This is just the current function

Structure of Event Handler Using AIO

call `aiowrite()` to initiate I/O

IF an error occurred

call a (user defined) function that handles I/O completion and errors

ELSE

give **AIO control block** for this operation to the event loop and tell it:

- the function to call when this I/O operation has finished
- the parameter that should be passed to that function

RETURN to the event loop

Structure of Event Handler Using AIO

Does not block!



call `aio_write()` to initiate I/O

IF an error occurred

call a (user defined) function that handles I/O completion and errors

ELSE

give `AIO control block` for this operation to the event loop and tell it:

- the function to call when this I/O operation has finished
- the parameter that should be passed to that function

RETURN to the event loop



Structure of Event Handler Using AIO

Does not block!

call `aiowrite()` to initiate I/O

IF an error occurred

call a (user defined) function that handles I/O completion and errors

ELSE

give **AIO control block** for this operation to the event loop and tell it:

- the function to call when this I/O operation has finished
- the parameter that should be passed to that function

RETURN to the event loop

“Continuation”

Structure of Event Handler Using LAIO

Does not block!

call `laio_syscall()` to initiate I/O

IF an error occurred **OR** the I/O operation completed

call a (user defined) function that handles I/O completion and errors

ELSE

call `laio_gethandle()` to get the handle associated with the I/O operation

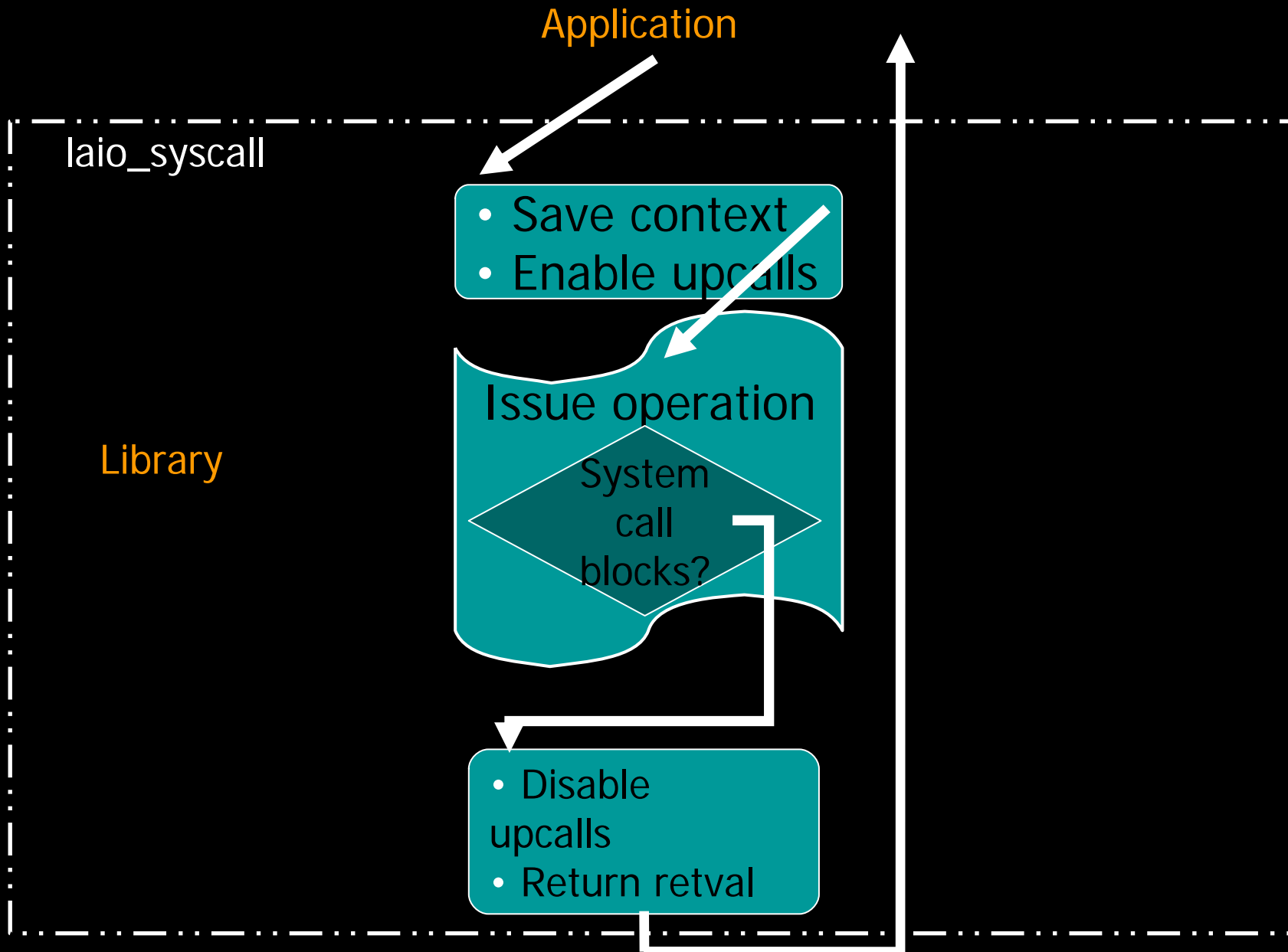
give this `handle` to the event loop and tell it:

- the function to call when this I/O operation has finished
- the parameter that should be passed to that function

RETURN to the event loop

“Continuation”

laio_syscall() – Non-blocking case



laio_syscall() – Blocking case

Application

laio_syscall

- Save context
- Enable upcalls

Library

Issue operation
System
call
blocks?

- Disable upcalls
- errno = EINPROGRESS
- Return -1

upcall handler

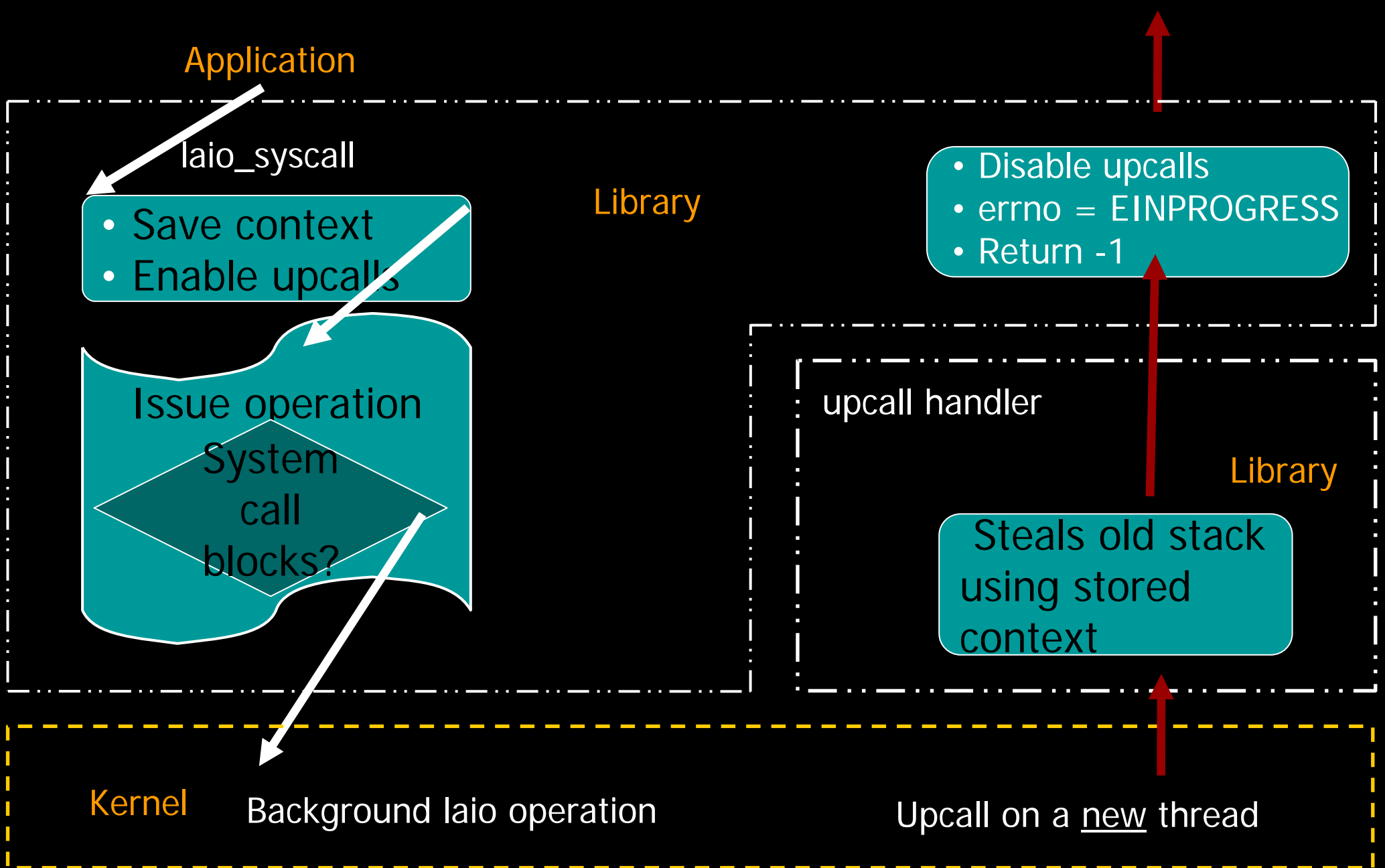
Library

Steals old stack
using stored
context

Kernel

Background laio operation

Upcall on a new thread



Unblocking case

upcall handler()

Library

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

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

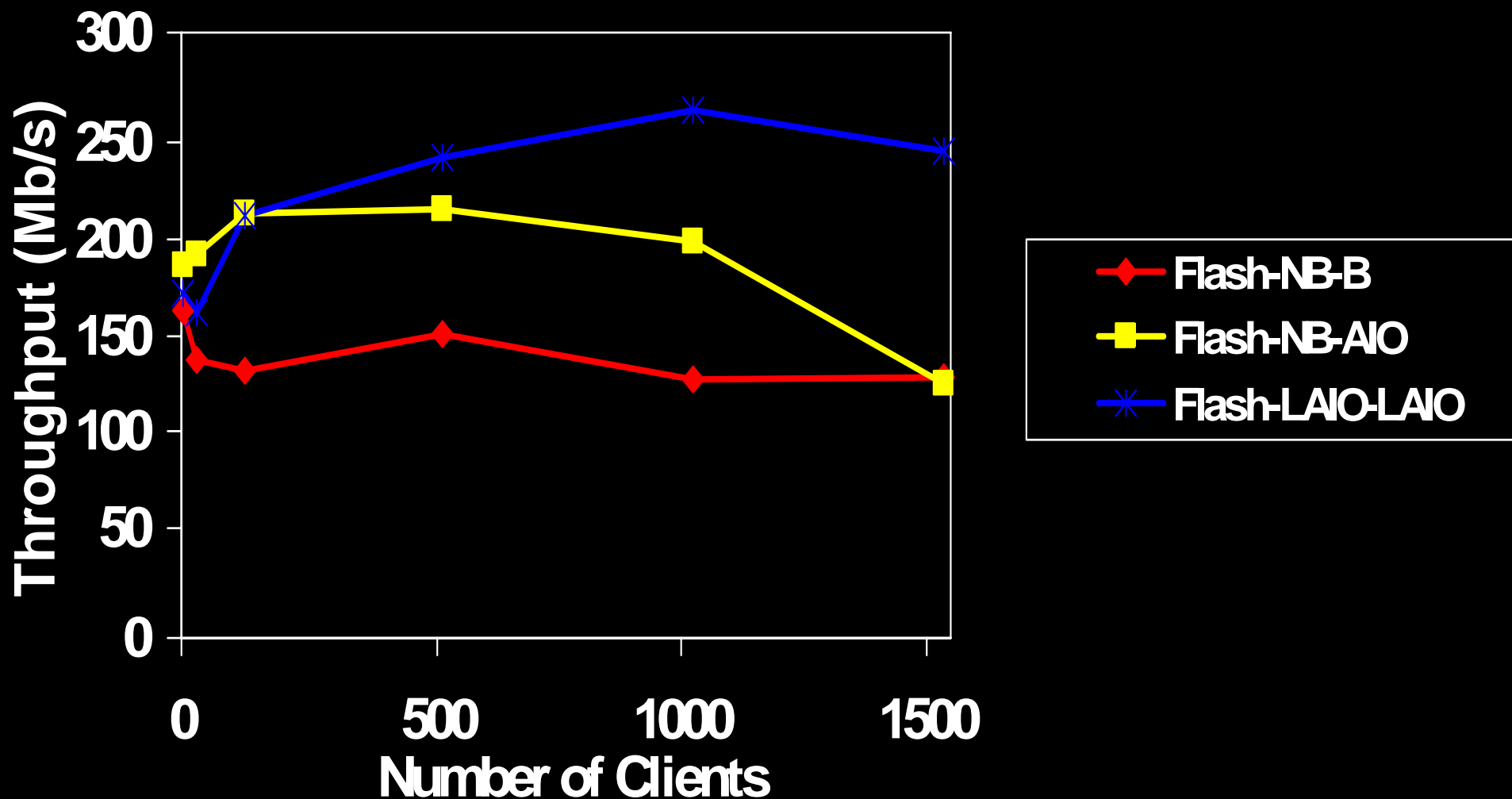
- Background laio operation completes, thread dies
- Upcall on the current thread

Kernel

- Most potentially blocking operations don't actually block
 - Experiments: 73% - 86% of such operations don't block
- Lower overhead
 - Micro-benchmark: AIO is 3.2 times slower than LAIO for non-blocking operations

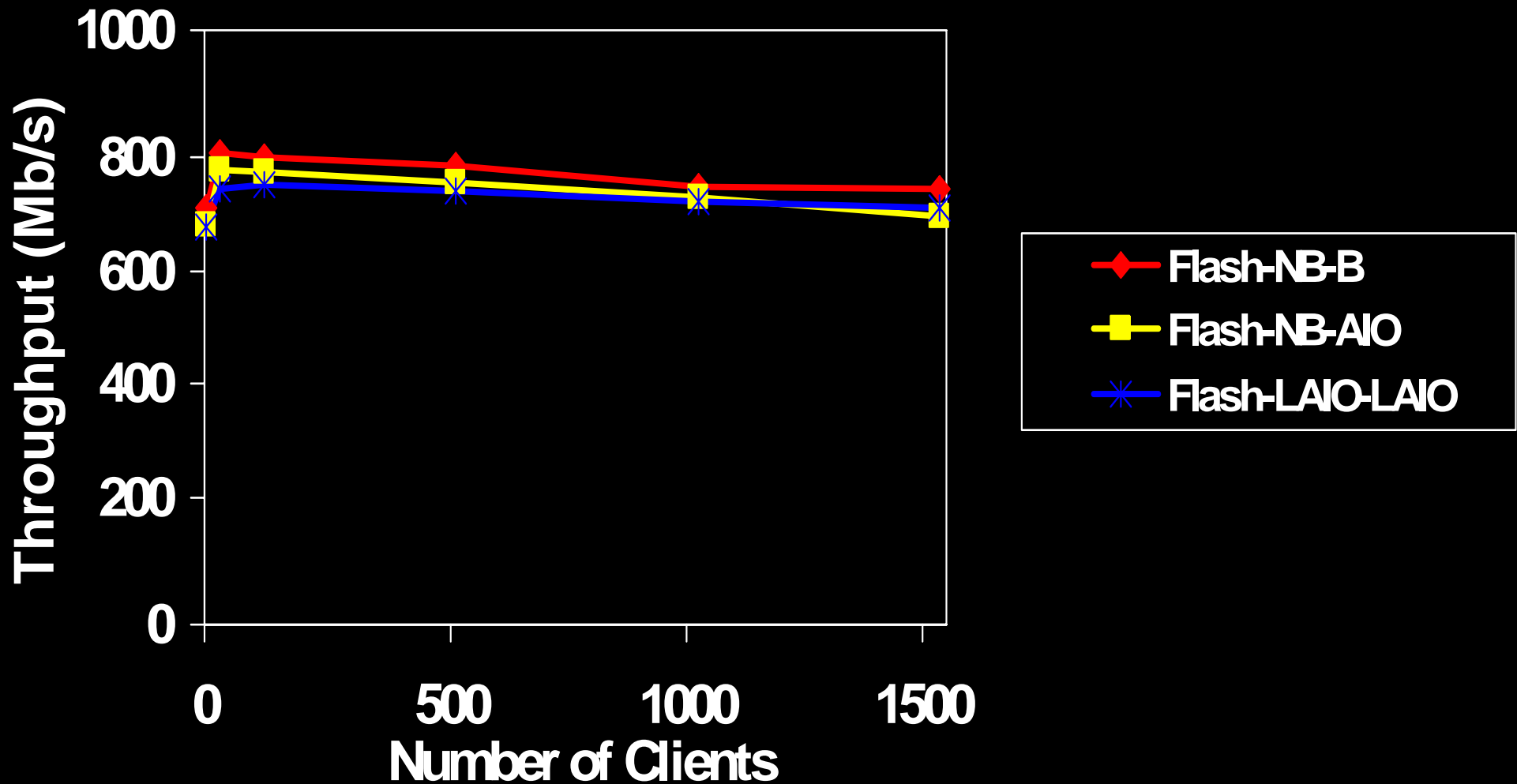
Performance: Large Workload

Berkeley Workload (warm cache)



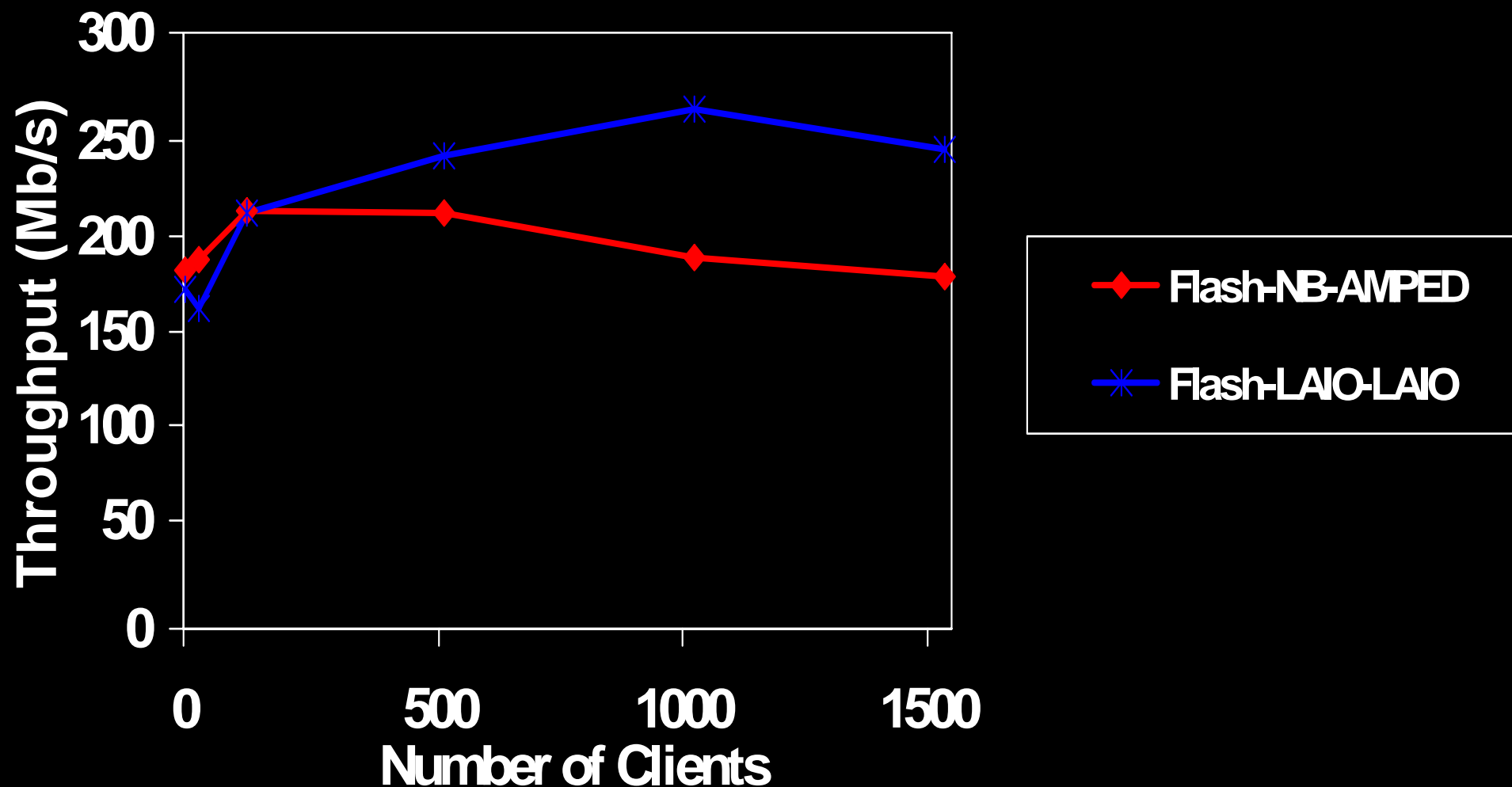
Performance: Small Workload

Rice Workload (warm cache)



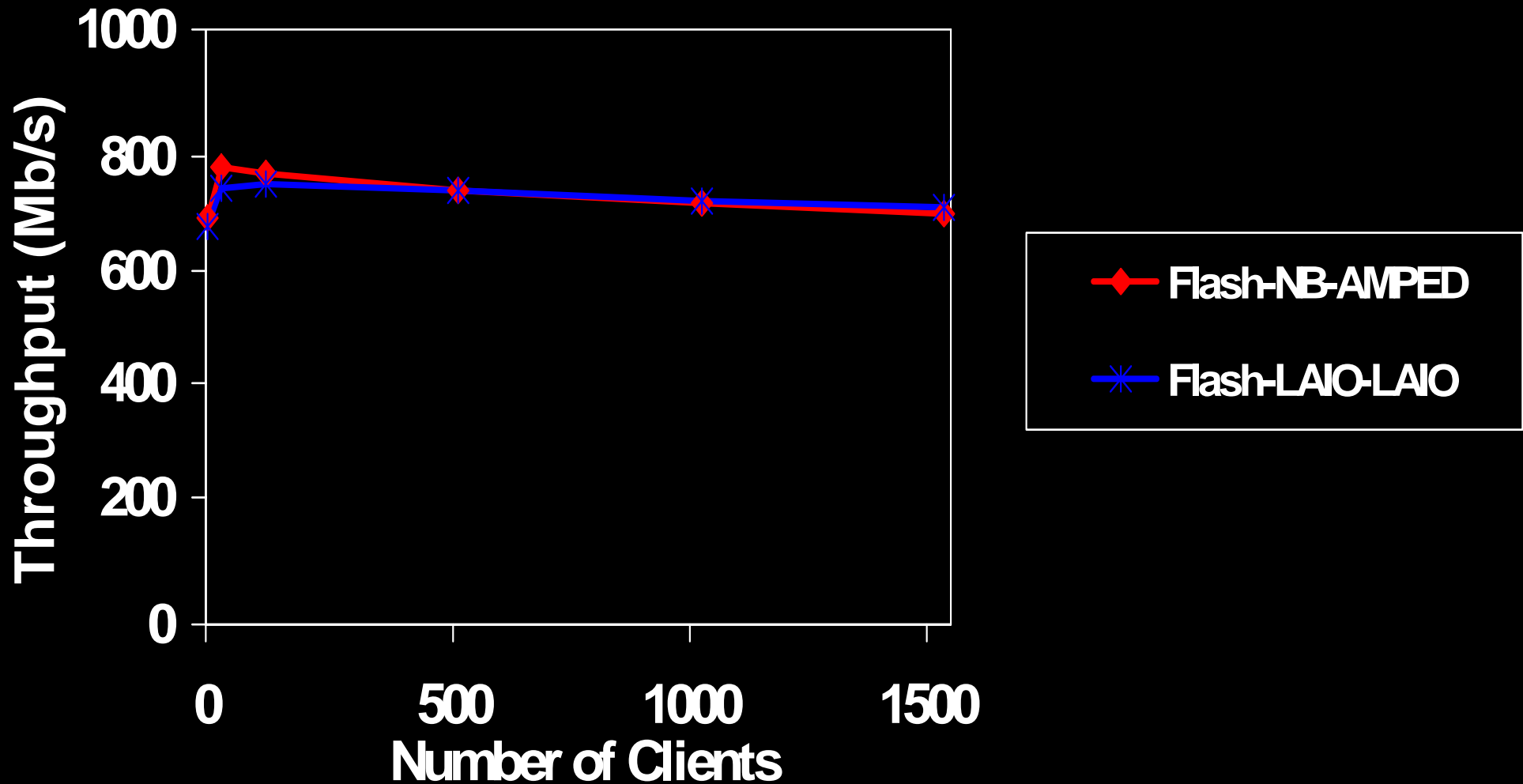
Performance of LAIO vs. AMPED

Berkeley Workload (warm cache)



Performance of LAIO vs. AMPED

Rice Workload (warm cache)



Critique

Strengths:

- Explains the problem well, with representative examples of the current solutions.
- Comparisons against many alternatives under a variety of conditions (big and small workloads, warm and cold cache).
- General solution, robust to changes in OS.

Weaknesses:

- Claim about superior disk utilization not explained.
- So many comparisons that it is hard to see the important results. Could these have been presented differently?
- Exaggerates claims about code complexity and confuses the issue by presenting quantitative measures that don't relate to code examples.

Discussion