Improving IPC by Kernel Design

by Jochen Liedtke
presented by Matt Hoffman

“The IPC Dilemma”

- Microkernels are good: better security, modularity, etc.
  - also “the key” to distributed systems– somewhat tangential to this paper though.
- ... but, IPC is slow (or at least Mach’s was)
  - 50–500 (or 100) μs for a small message
  - compare to target of 5–7 μs

A brief aside on “security”

- “ When they talk about security, do they mean prevention from malicious attacks or unintentional errors?”
- “The author has suggested numerous techniques to improve IPC performance, however, I wonder what collective impact of all these techniques have on security?”
- [What is the tradeoff?]

Why is IPC important?

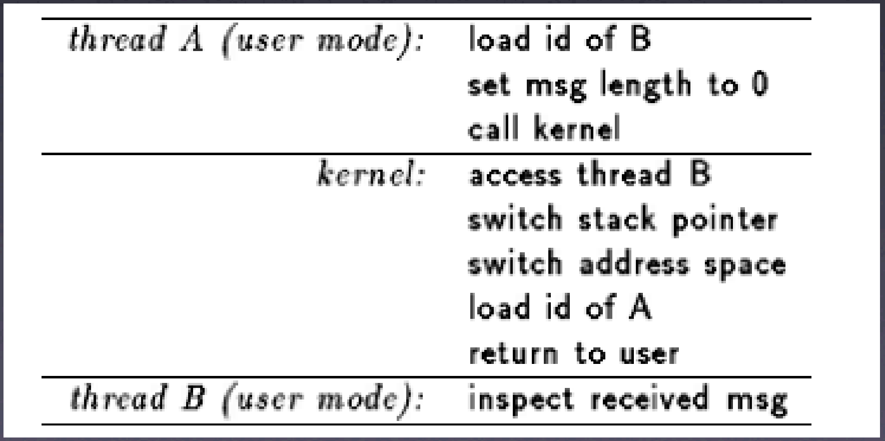
- “Do you have any insight as to why the author targeted a µ-kernel architecture for IPC improvement [and not a monolithic kernel]?”
- “... he never mentions how his optimizations affected the overall performance of the system...”
- “Is it worth it to modify that much of the system?”

The L3 microkernel

- Mach’s IPC is complex; esp. buffering IPC
  - Led some developers to move drivers/etc. into the kernel.
- L3 is a response to this. No buffering; very light-weight.
  - predecessor to L4 (L4 family); recently used in embedded devices.

Making IPC faster...

- Look at the most basic case
  - (this takes 172 cycles, 3.5 µs)
  - Try and get as close to this as possible
  - We want under T=7 µs
  - (they make it to around 5 µs)



thread A (user mode):	load id of B
	set msg length to 0
	call kernel
kernel:	access thread B
	switch stack pointer
	switch address space
	load id of A
	return to user
thread B (user mode):	inspect received msg

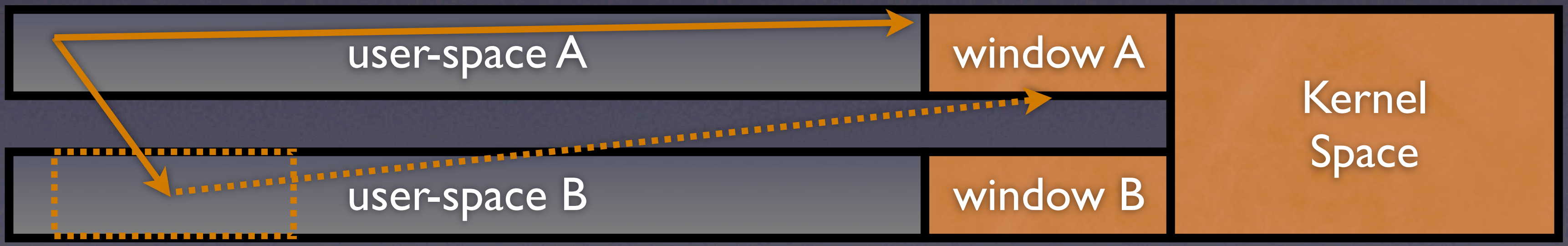
The architectural level

Avoid system calls

- Stay out of the kernel as much as possible
- Allow synchronous/blocking IPC calls
  - *call* and *reply & receive*
- Allow the sending of complex messages:
  - memory objects,
  - multiple messages, etc...

Complex messages

- Basically a copy of some region of memory
- Each process has a *communication window* in user-space; only "kernel accessible"
- Copy data to window and temporarily map
- Mapping requires one-word p.table change



Complex messages contd.

- “RPC and LRPC both use shared message buffers ... [do] they also use shared message buffers?”
- Also, there’s that TLB weirdness. Just flush it on process switches. (and for threads)
  - Also you’d want more hardware support if that’s not for free. (or multiprocessors)

Strict process orientation

- Each process has a kernel stack attached
- Is combined with its *thread-control block (tcb)*
- Each process runs exactly the same between user- and kernel mode.

Thread control blocks

- Thread (process) specific information
  - such as: registers, kernel stack, etc.
- Accessed as an array, but each block has next/prev pointers for various queues
- Can lock a thread by deleting it from a queue/etc.

The algorithmic level

Timeouts/queues

- Already talked about queues ready/waiting
- “Queues” for IPC timeouts; Utilizing *n* lists
  - thread is put into unordered list (τ mod *n*) for a timeout of τ
  - for *k* threads, inspect *k*/*n* entries per cycle on average

Lazy scheduling

- using the *call* or *reply* & *receive* IPC calls require “expensive” queue add/remove
- So, obviously, don’t do them
  - ready queue contains **at least** all ready threads
  - wakeup queue contains **at least** all threads waiting in this class (i.e. τ mod *n*)
  - delete when parsing; add on send, timeslice end, and hardware interrupt.

Lazy scheduling contd.

- “The author says [lazy scheduling] performs better and better with increasing IPC rate, why?”
  - Answer (I think): less time spent on adding/removing from the queues.
  - Especially makes sense with the later ping/pong test

Direct process-switch

- For *call* and *reply & receive* calls, transfer control directly to called thread.
  - donates the current timeslice
- BUT if A sends to B, B replies, and C is waiting to send to B: B runs again with C's message.
- sounds unfair if there's still time left in A's timeslice; depends on timing granularity

Short messages via reg.

- Many RPCs have short messages
  - e.g. ack/error replies
- In L3 many were ≤8 bytes (plus 8 byte id)
- Transfer directly using registers.
- “ What suffers when we reallocate registers to be used for short IPC transfer?”

The interface level

- Need good user/kernel interfaces.
  - light-weight RPC stubs, etc.
  - IPC calls for complex messages must use structure (dope vectors); grouped by type
- Avoid unnecessary copies
  - compiler support; don't copy on send/reply when using same variables
  - use registers where possible

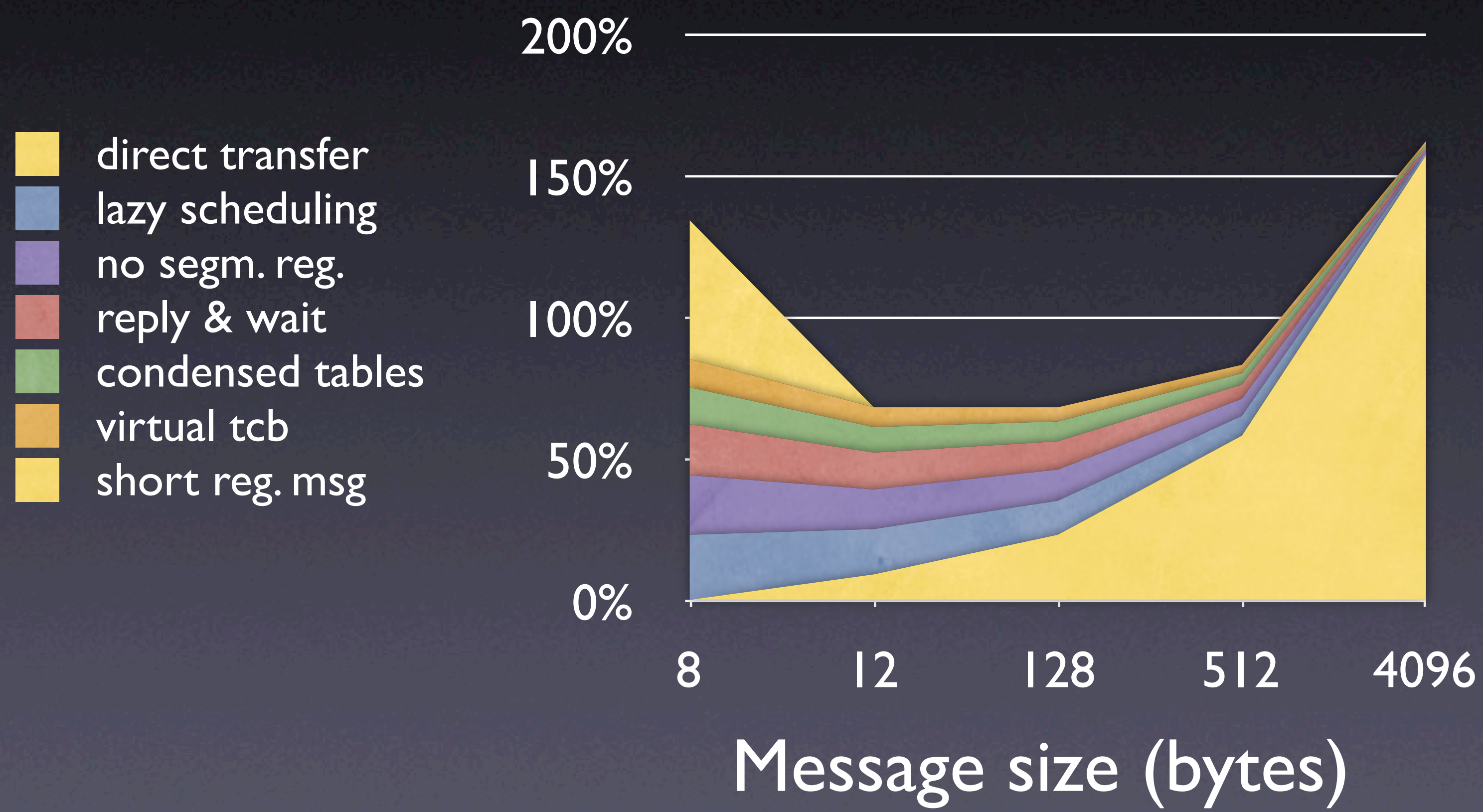
The (kernel) coding level

- ... keep the kernel as small as possible.
- Use the cache intelligently.
- Avoid loading the segment register except when required by user software (via a flag)
- Use special features of the hardware were possible (such as x86's register aliasing)
- Avoid jumps
- Avoid checking where possible

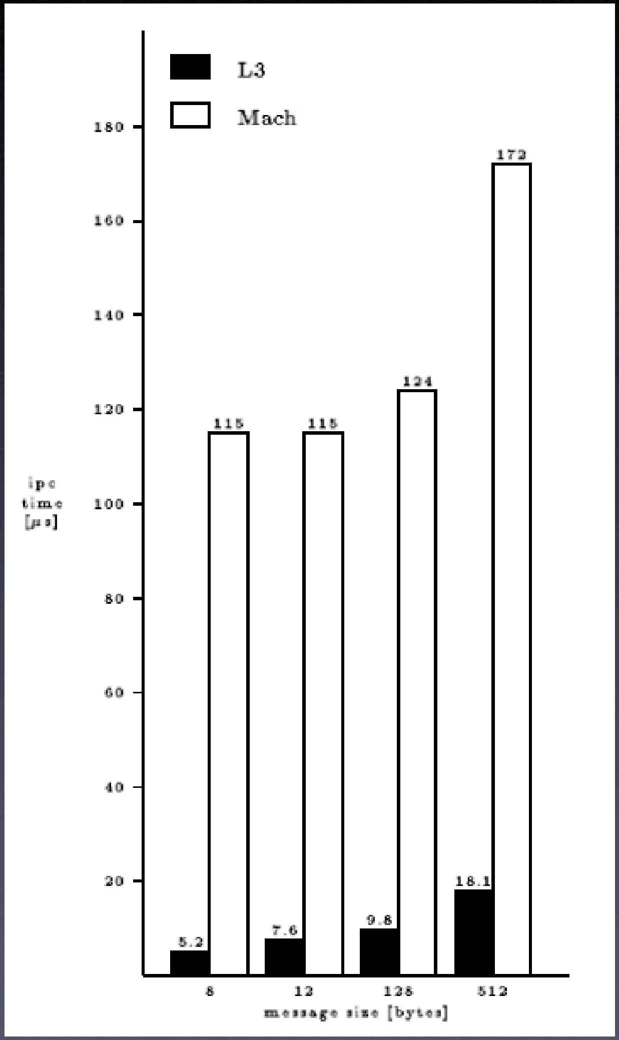
Results

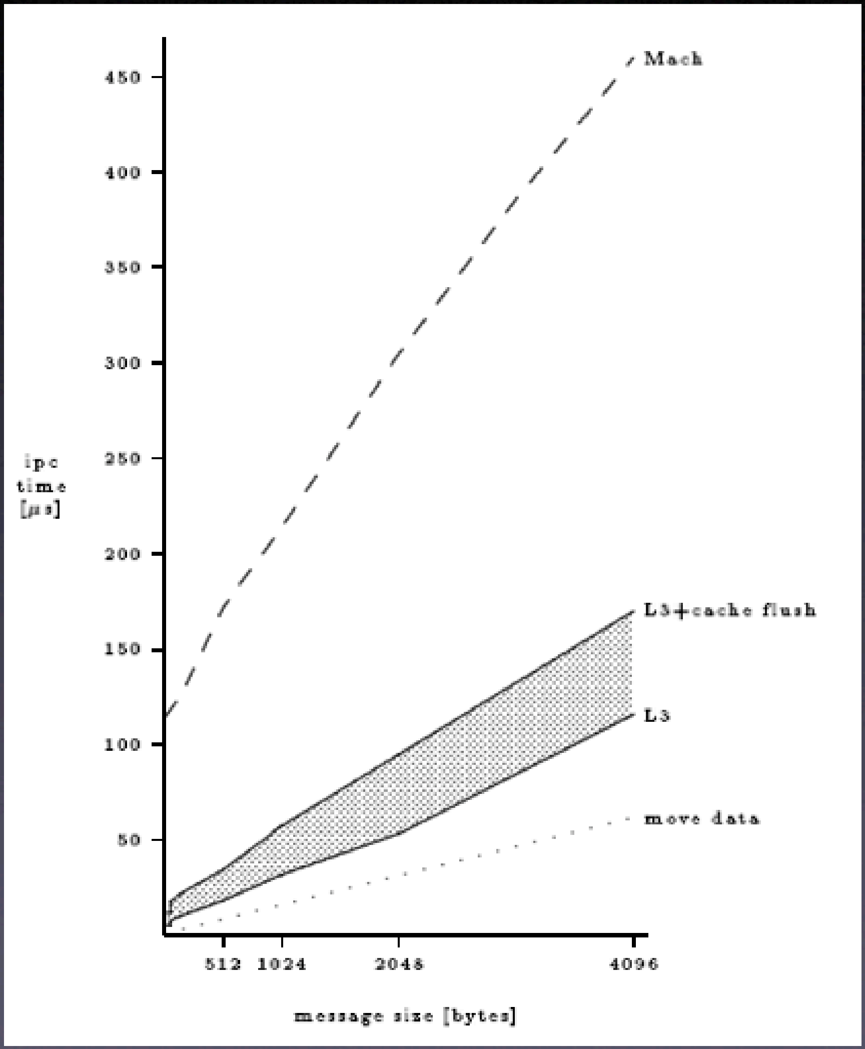
Easy to test effects...

Slowdown for removing each optimization



More results





Remarks/extras

Mach-style ports

- port tables added and briefly tested
- accessible by the kernel
  - checks offloaded onto the page-fault handler
  - still have to know which ports to use

Etc.

"This paper was written using LaTeX on top of L3"

A few unrelated questions

- “How is the performance compared with, e.g., the UNIX kernel?”

- "Author talk something about Multiprocessor, but the whole experiment is based on single processor system. How about use IPC on multiprocessor system?"
- "... going over a network or is L3 not even able to do IPC over a network?"

- “What is the reason that micro-kernels (such as L3) couldn't be the mainstream in OS design?”

- “How does ‘the persistence of data and threads’ and its ‘Clans & Chiefs model’ make ipc more efficient? (section 3)”
    - the *Clans & Chiefs* model is for security– messages stay within a “clan” or are redirected to the “chief”.
    - the “persistance..” model basically makes everything (data, processes, etc.) stick around indefinitely.