Improving IPC by Kernel Design

Jochen Liedtke
German National Research for Computer Science
Presented by Emalayan

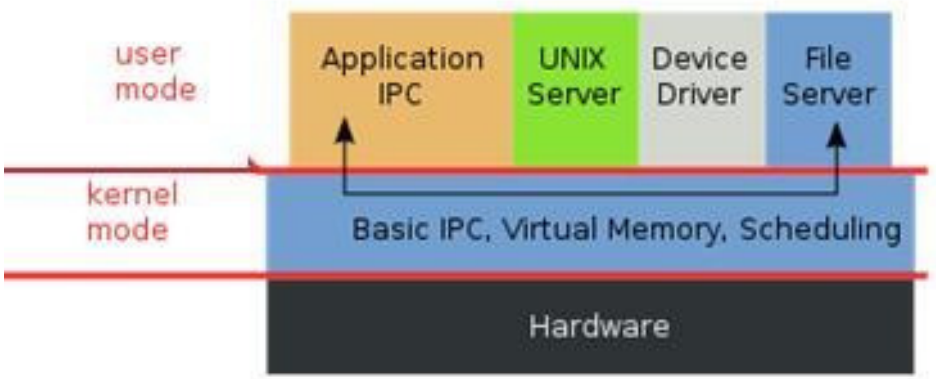
Agenda

- Introduction to Micro Kernal
- Design Objectives
- L3 & Mach Architecture
- Design
  - Architectural Level
  - Algorithamatic Level
  - Interface Level
  - Coding Level
- Resutls
- Conclusion

Micro Kernels

- Micro kernels
  - Microkernel architectures introduce a heavy reliance on IPC, particularly in modular systems
  - Mach pioneered an approach to highly modular and configurable systems, but had poor IPC performance
  - Poor performance leads people to avoid microkernels entirely, or architect their design to reduce IPC
  - Paper examines a performance oriented design process and specific optimizations that achieve good performance
- Popular Micro kernels

  - L3 & Mach



- The rationale behind its design is to separate mechanism from policy allowing many policy decisions to be made at user level

Design Objectives

- IPC performance is the primary objective
- Design discussions before implementation
- Poor performance replacements
- Top to Bottom detailed design (From architecture to coding level)
- Consider synergetic effects
- Design with concrete basis
- Design with concrete performance

L3 & Mach Architecture

- Similarities
  - Tasks, threads
  - Virtual memory subsystem with external pager interface
  - IPC via messages
    - L3 - Synchronous IPC via threads
    - Mach – Asynchronous IPC via ports
- Difference
  - Ports
- L3 is used as the workbench

Performance Objective

- Finding out the best possible case
  - NULL message transfer using IPC
- Goal
  - 350 cycles (7 us) per short message transfer

Approach Needed

- Synergetic approach in Design and Implementation guided by IPC requirements
  - Architectural Level
  - Algorithm Level
  - Interface Level
  - Coding Level

Architectural Level Design

a) System Calls

- System calls are expensive (2 us)

- IPC implementation with minimum system calls.

- New system calls introduced.
    - call()
    - reply & receive next()

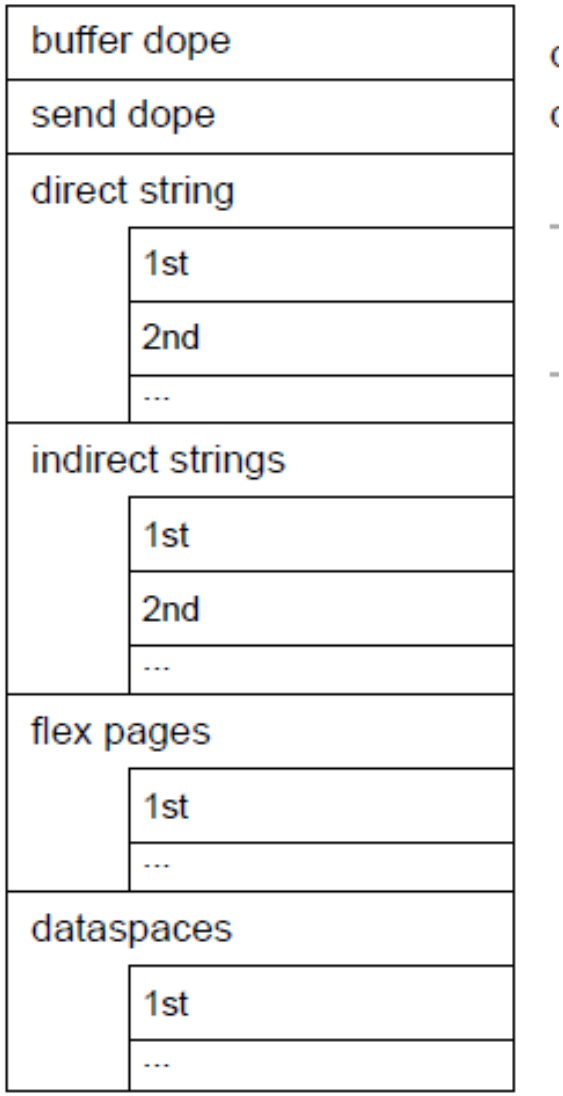
- Totally around 4 us saved by reducing two system calls (instead of send(), receive(), call(), reply())

Architectural Level Design …

b) Messages

- A message is a collection of objects passed to the kernel
- Kernel is responsible for delivering the message
- A message can contain direct strings, indirect strings, flex pages, data spaces.
- Buffer dope and Send dope are compulsory for messages

Sending Complex messages reduces system calls and IPC because of reduction in address space crossing



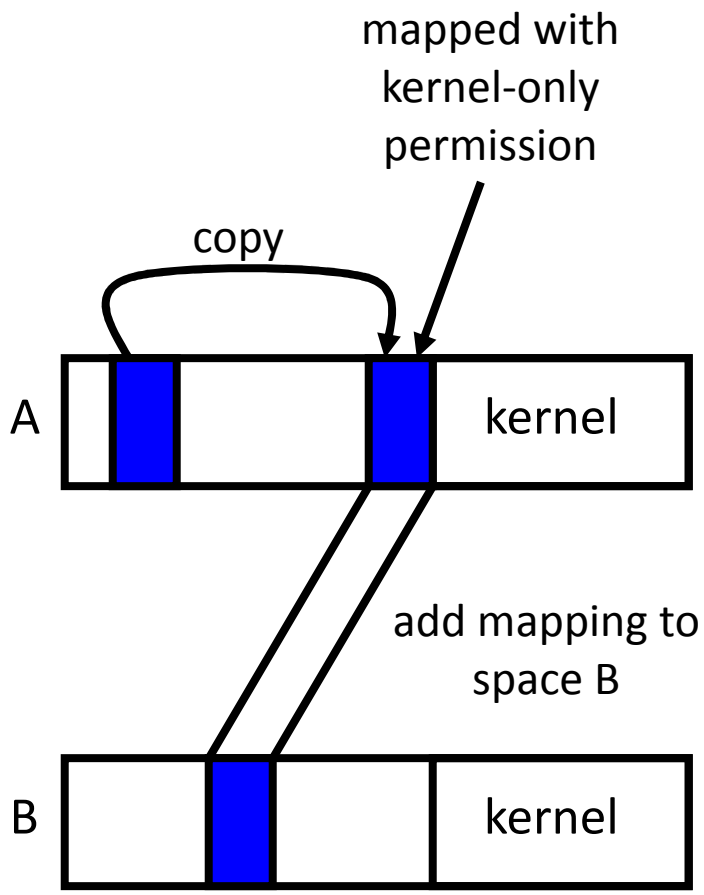
Architectural Level Design …

c) Direct Transfer by temporary mapping

- Two copy message transfer costs 20 + 0.75n cycles

- L3 copies data once to a special communication window in kernel space

- Communication Window is mapped to the receiver for the duration of the call (page directory entry)



Architectural Level Design …

d) Thread Control Blocks

- Hold kernel and hardware level thread specific data
- Every operation on a thread requires lookup, and possibly modification, of that thread's TCB
- Provide faster access by means of array offsets
- It saves 3 TLB misses per IPC

Algorithamic Level

a) Thread Identifiers
b) Handling virtual queues
c) Timeouts and Wakeups
d) Direct Process Switch
   - Scheduler is not invoked between process context switch
   - No sender may dominate the receiver
   - Polling threads are queued, kernel does not buffer messages
e) Short Messages via Registers
   - direct transfers of messages
   - gain of 2.4 us (48%) achieved

Algorithamic Level …

f) Lazy Scheduling

- Scheduler maintains several queues to keep track relevant thread-state information
  - **Ready queue** stores threads that are able to run
  - **Wakeup queues** store threads that are blocked waiting for an IPC operation to complete or timeout (organized by region)
  - **Polling-me queue** stores threads waiting to send to some thread

- Efficient representation of data structures
  - Queues are stored as doubly-linked lists distributed across TCBs
  - Scheduling never causes page faults

Interface Level Design

a) Avoiding unnecessary copies
b) Parameter passing

   - Registers can be used

   - Input and /Output parameters in register give better chance for compilers

Coding Level Design

a) Cache aware design
   - Reducing Cache misses
   - Minimizing TLB misses

b) Architecture aware design
   - Segment Registers
   - General Registers
   - Avoiding jumps and checks

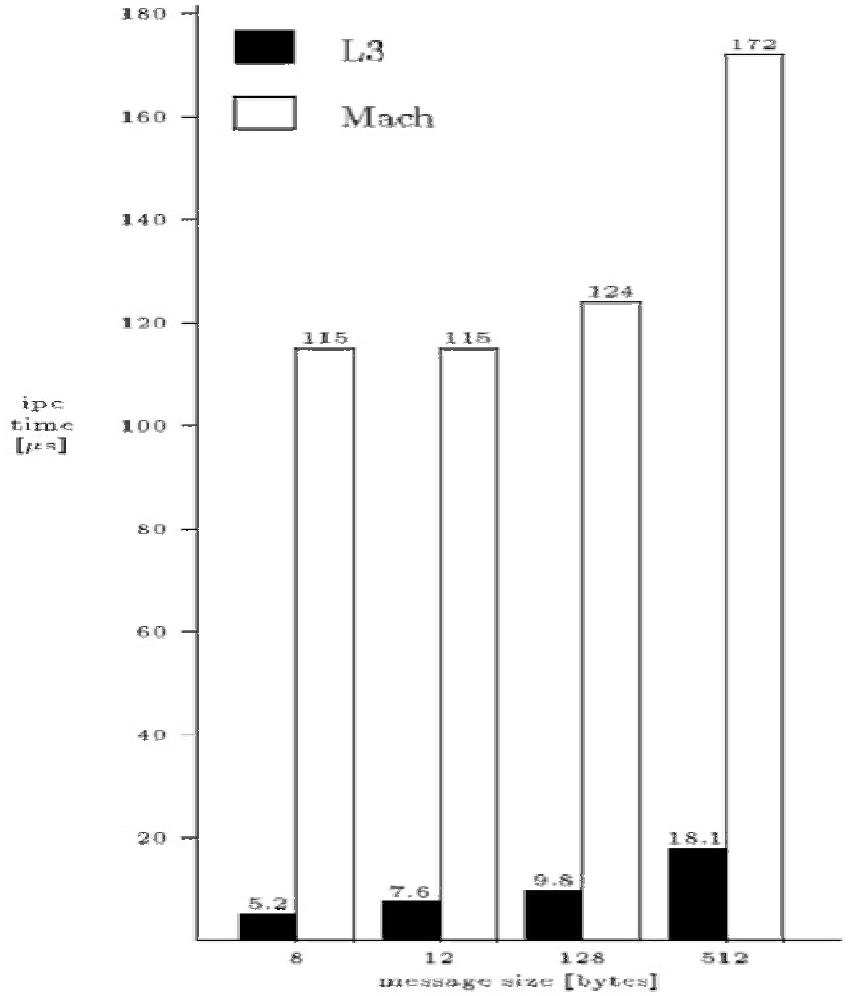
Summary of Techniques

removed optimization	time increase (n-byte ipc) 8	12	128	512	4096
short msg via reg	49%	–	–	–	–
direct transfer	–	9%	23%	58%	157%
lazy scheduling	23%	16%	12%	7%	1%
no segm reg	21%	14%	11%	6%	1%
reply & wait[a]	18%	13%	10%	5%	1%
condensed tables[b]	13%	9%	7%	4%	1%
virtual tcb[c]	10%	7%	5%	3%	1%

- Table completely ignores the synergetic effects.
- Direct message transfer dominates for large messages
- For short messages register transfer works well

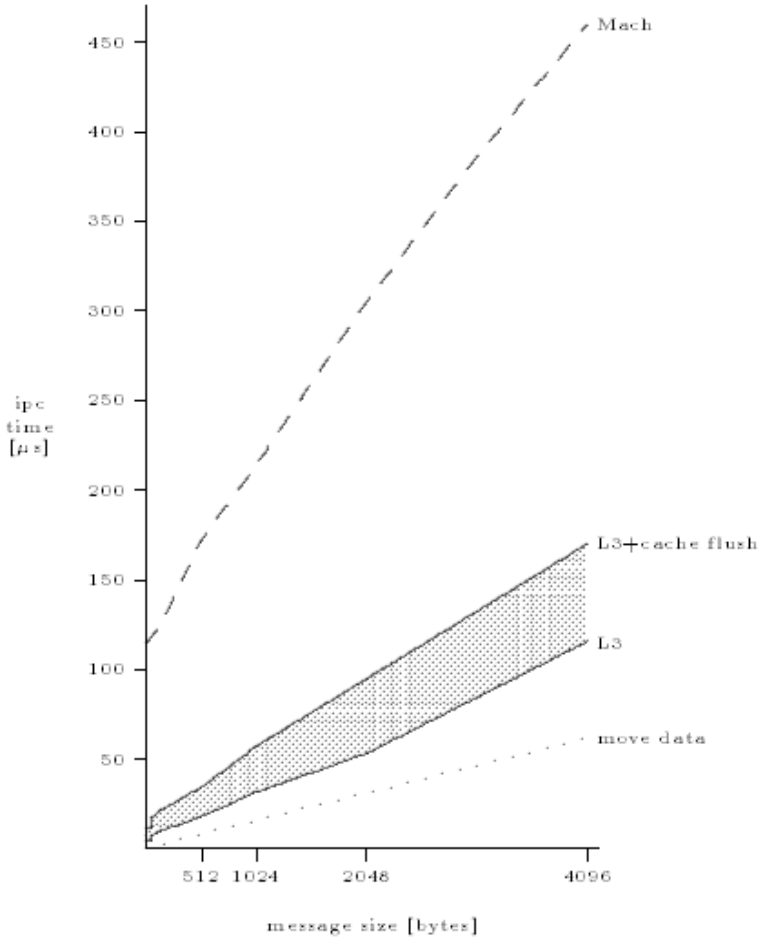
Performance Comparison

- Measured using pingpong micro-benchmark that makes use of unified send/receive calls
- For an n-byte message, the cost is 7 + 0.02n µs in L3



Performance Comparison

- Same benchmark with larger messages.
- For n-byte messages larger than 2k, cache misses increase and the IPC time is 10 + 0.04n µs
  - Slightly higher base cost
  - Higher per-byte cost
- By comparison, Mach takes 120 + 0.08n µs



Conclusion

- Efficient and effective IPC is mandatory for micro kernel design , which was a major limitation of Mach
- L3 demonstrates that good performance (22 times faster) by means of above techniques.
- Techniques demonstrated in the paper can be employed in any system, even if the specific optimizations cannot

THANK YOU

Questions

- Monolithic Kernels perform better than L3 or even L4 (written in assembly!). Why should I even bother with micro kernels and their ever-increasing IPC performance. The whole idea of IPC message passing seems to be plagued by performance problems.
- It would be interesting to know if some of the assembly-level hacks (discussed in this paper) been implemented in other production OS?

Questions

- The paper talks about an IPC message having a "direct string", some number of "indirect strings", and some number of "memory objects". Then they later discuss the idea of optimizing through registers the case of
an IPC with less than 8 bytes of payload. How exactly does this work? Is there a separate system call for "small IPC"? It's not obvious how two registers containing arbitrary integer values map to the structured message concept involving strings and memory objects.

Questions

- Majority of the performance gain in L3 is by pushing out ports right checking and message validity checking.How does this pushing of checking code into user space affect the performance(assuming similar checking is done in user space) and security?

Questions

- In section 5.6, the authors discuss how Mach's "mach_thread_self“ call took nine times more time than the bare machine time for a user/kernel/user call sequence. However, this system call is complete unrelated to IPC. In fact, it shows that Mach's system call mechanism is somewhat inefficient in general. Can we blame this for the difference between Mach's and L3's IPC performance, rather than blaming the implementation of IPC itself?

Questions

- Can the techniques proposed here in the paper be utilized in state-of-the-art mutli-core platforms, will these optimizations totally valid or should there still be some subtle problems that we may need to take care of?
- Can the techniques proposed be adopted into a regular kernel based system?

Questions

- I do not think using register as a mechanism to pass data, is a good idea. To use that don't we have to have intelligent scheduler for that to make sure the correct thread access the register to read the data ?
- Doing so many modifications to the OS, wouldn't it have an impact on the other operations in the OS ? I mean to which extent IPC counts towards the overall performance of the OS ?