Improving IPC by Kernel Design

Jochen Liedtke
German National Research Center for Computer Science
SOSP 1993

Presented by Bryon Nevis
Summary

• L3 μ-kernel is 22X faster than Mach
  – Achieved by addressing performance of the whole system

• Performance optimizations are generally applicable
  – **Implementation** makes *all the difference*!
Implementation Platform

• L3 implemented on uniprocessor Intel 486-DX50

• Basic features
  – Predictable performance, 50 MHz clock
  – Segmentation, ring architecture
  – Virtual memory, 2 level index, 4K pages
  – 32-entry TLB, flushed by hardware
  – 8K cache, 128 bit cache lines
Figure 7: 486-DX50, L3 versus Mach Ipc Times

Figure 8: 486-DX50, L3 versus Mach Ipc Times
L3 vs Mach IPC times

Latency (clocks)

- Theoretical lower bound: 172
- L3 (achieved): 250
- L3 (goal): 350
- Mach (measured): 5760

Latency (μs)
17(19) Techniques for faster IPC

Four broad categories

• OS architecture (5)
• Internal algorithms (6)
• User-kernel interface (+2)
• Efficient coding & use of memory (6)
Analysis of improvements

- Optimizations in paper account for < 50% of actual L3 vs Mach performance difference.

- What else could be responsible?
  - Mach ports & security?
  - Excessive modularity?
  - Lack of locality?
  - Use of expensive machine instructions?
OPTIMIZATION #0: MACHINE INSTRUCTIONS
Performance vs Protection
Relative costs of control transfer on 80486

Cost of task switch (registers + address space) [Naive implementation]

Cost of ring transition (L3)

- CALL rel & Ret: 8 clocks
- CALL far & Ret: 38 clocks
- INT+IRET kernel trap: 107 clocks
- INT+IRET+TS: 687 clocks
Ideal/Best case Breakdown
(Send IPC: 172 cycles)

- Kernel entry, 71, 41%
- Kernel exit, 54, 31%
- Set source thread ID, 2, 1%
- Simulate address space switch, 14, 8%
- Access target TCB, 13, 8%
- Simulate stack switch, 4, 2%
- Read parameters END, 11, 7%
- Simulate null message, 1, 1%
- Set target thread, 2, 1%

"Return" to destination thread

Overwrite fields in 486 task state segment to effect context switch

Running on kernel stack in kernel mode

Start
  - Address empty message to caller
  - Kernel trap
  - Switch code & stack
  - Return to user
  - Read sent message
  - End
Achieved performance (250 cycles)

What’s missing? 78 cycles:

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Remain</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>68</td>
<td>5.5.3 - Check segment register validity (need to check CS,SS?); 4 or 5 segment registers @ 2 clocks each</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>5.3.1- Compute TCB from thread ID, verify thread ID in TCB</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Save/restore registers while in kernel mode? (Since all GPR’s are used up in table 6.)</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Check if FPU register or debug register used</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Demux system call?</td>
</tr>
</tbody>
</table>

The paper only accounts for only 17 of the remaining 78 cycles
OPTIMIZATION #1,2: ELIMINATE SYSTEM CALLS
5.2.1 Avoiding 2 system calls
5.3.5 Direct process switch

System V message queue

Client

```c
while (true) {
    msgsend(request)  \[1\]
    msgrcv(reply)  \[2\]
    /* compute */
}
```

Server

```c
while (true) {
    msgrcv(request)  \[3\]
    /* process */
    msgsend(reply)  \[4\]
}
```

4 system calls per IPC

Note: mach_msg() can both send and receive too
5.2.1 Avoiding 2 system calls
5.3.5 Direct process switch

Improved client
while (true) {
    buffer = request
    call(buffer)
    reply = buffer
}

Improved server
receive(buf)
request = buf
do {
    /* process */
    buf = reply
    reply_and
    receive(buf)
    request = buf
} while (true)

2 system calls per IPC (save 344 cyc)

5.3.5 Server does not block until all incomings are processed
Discussion

• Message queue or procedure call?
  – Data is delivered via memory page
  – Kernel delivers all incoming messages before returning to the caller
OPTIMIZATION #3,4: AVOIDコピーING DATA
Performance vs Protection
Traditional Data Transfer (Protection)

- 1\textsuperscript{st} copy: process A to kernel
- 2\textsuperscript{nd} copy: kernel to process B
SRC RPC / LRPC (Performance)

- Communicate via shared memory & synchronization

Problems
- Covert channels (not usable for MLS secure systems)
- Confused deputy problems (TOCTOU race conditions)
- Pairwise communication buffers (hard to use, eats memory)
- Requires extensive pointer manipulation
Middle ground: temporary mapping

• Observation
  – Fast and secure if copy message into target address space and sender cannot modify message after sending it
5.2.3 Direct transfer by temporary mapping

• Performance tricks
  – 1 PDE=4MB
  – Can flush all TLB or one 4K page

  – TLB “window clean” algorithm
    • Flush and re-establish mapping after timers, page fault, interrupt; invalidate 4M of pages after thread switches (address space switches always flush TLB)
5.3.6 Short messages via registers

- 60% of IPCs transfer \( \leq 32 \) bytes\(^1\)
- L3: 80% of IPCs transfer 8 bytes

<table>
<thead>
<tr>
<th>input</th>
<th>register</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive buffer addr</td>
<td>EAX</td>
<td>result code</td>
</tr>
<tr>
<td>send timeout</td>
<td>EBX</td>
<td>bytes 0...3 rcved msg</td>
</tr>
<tr>
<td>send message addr</td>
<td>ECX</td>
<td>bytes 4...7 rcved msg</td>
</tr>
<tr>
<td>destination thread id</td>
<td>EDX+ESI</td>
<td>source thread id</td>
</tr>
<tr>
<td>receive timeout</td>
<td>EDI</td>
<td>— scratch —</td>
</tr>
<tr>
<td>stack pointer</td>
<td>EBP</td>
<td>— scratch —</td>
</tr>
<tr>
<td></td>
<td>ESP</td>
<td>stack pointer</td>
</tr>
</tbody>
</table>

Table 6: Register Usage for IPC Parameters (486)

Note: This table accounts for all of the GPRs on x86 CPU’s

120 cycles saved per IPC
Typical scheduler flow

- **Costs: 58 cycles**
  - Cost includes 4 TLB misses (if memory ops hit separate pages)
  - 7 memory ops to insert
  - 4 memory ops to remove
Observation

• It only takes 2 memory ops instead of 11 memory ops to change a flag in the TCB
Sub-optimization 1

• Scheduling queue is just a hint; only costs one additional memory op to double-check the TCB state
  – Note other optimizations guarantee that there won’t be a page fault for this access
  – Not fatal to performance if the queue contains a few extra entries
Sub-optimization 2

• Removing from a linked list is fast
• Combine queue cleanup with queue parsing for other reasons
5.3.4 IPC cost would **double** w/o lazy scheduling optimization

**OLD WAY**
- 4 queue ops per ipc

**NEW WAY**
- 2-5 ipcs per queue op
- (50 at extreme)

At 2:1 ratio
58 x 2 = **116** cycles per IPC savings

At 5:1 ratio
58 x 5 = **290** cycles per IPC savings
OPTIMIZATION #6,7
5.5.2 Minimizing TLB misses

- Fit into as few 4K pages as possible:
  - IPC-related kernel code
  - GDT, IDT, and TSS (486-specific)
  - System clock
  - Other important system tables
  - TCB array, Kernel stacks

100 cycles saved per IPC
Are compact data structures always Good?

What is LOCALITY?
What assumptions are being made?
5.5.3 Segment registers

- Segreg loading is expensive
  - Part of the protection system
  - Check (1 clock compare, 1 clock jump) for correct segment register value vs 9 clocks for unconditional load (segment descriptor is actually 64-bits wide)

**66 cycles saved per IPC**
BACKUP
5.3.2 Handing virtual queues

- Ensure that processing thread message queues does not lead to page faults, since TCBs are mapped into virtual memory.

Potentially fatal to performance; no specific number given in paper.
5.5.5 Branch prediction

• Branch not taken: 1 cycle
• Branch taken: 3 cycles!
## Most impactful optimizations

<table>
<thead>
<tr>
<th>Section</th>
<th>Cycles</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>344</td>
<td>2 system calls instead of 4</td>
</tr>
<tr>
<td>5.2.3</td>
<td>26-3092?</td>
<td>Copy message only once</td>
</tr>
<tr>
<td>5.3.2</td>
<td>10000’s?</td>
<td>Unknown cost of page fault while processing TCB’s</td>
</tr>
<tr>
<td>5.3.4</td>
<td>290</td>
<td>Lazy scheduler queue management</td>
</tr>
<tr>
<td>5.3.5</td>
<td>172?</td>
<td>172 defer context switch on reply</td>
</tr>
<tr>
<td>5.3.6</td>
<td>120</td>
<td>Use register messages</td>
</tr>
<tr>
<td>5.5.2</td>
<td>100</td>
<td>Avoid 11 TLB misses</td>
</tr>
</tbody>
</table>

*Note: For 7 of the 17 listed improvements, the actual improvement was not specifically quantified*