Critique Of Microkernel Architectures

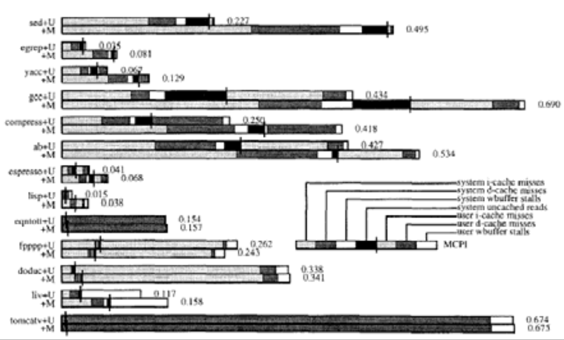
I'm not interested in making devices look like user-level. They aren't, they shouldn't, and microkernels are just stupid.
Linus Torvalds

Is Linus right?

Microkernel Performance

- First generation µ-kernel systems exhibited poor performance when compared to monolithic UNIX implementations.
  - particularly Mach, the best-known example
- Reasons are investigated by [Chen & Bershad, 1993]:
  - instrumented user and system code to collect execution traces
  - run on DECstation 5000/200 (25MHz R3000)
  - run under Ultrix and Mach with Unix server
  - traces fed to memory system simulator
  - analyse MCPI (memory cycles per instruction)
  - baseline MCPI (i.e. excluding idle loops)

ULTRIX VS. MACH MCPI



Interpretation

Observations:

- Mach memory penalty (i.e. cache missess or write stalls) higher
- Mach VM system executes more instructions than Ultrix (but has more functionality).

Claim:

- Degraded performance is (intrinsic?) result of OS structure.
- IPC cost (known to be high in Mach) is not a major factor [Bershad, 1992].

Assertions

1. **OS has less instruction and data locality than user code**.
   - System code has higher cache and TLB miss rates.
   - Particularly bad for instructions.

Assertions

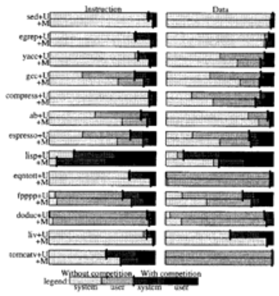
2. **System execution is more dependent on instruction cache behaviour than is user execution**
   - MCPIs dominated by system i-cache misses.
   - Note: most benchmarks were small, i.e. user code fits in cache.

workload	instruction cache				data cache			
	Ultrix		Mach		Ultrix		Mach	
	sys	user	sys	user	sys	user	sys	user
sed	0.129	0.005	0.283	0.005	0.041	0.001	0.132	0.003
egrep	0.014	0.001	0.046	0.001	0.010	0.000	0.023	0.000
yacc	0.028	0.004	0.069	0.003	0.011	0.011	0.029	0.012
gcc	0.103	0.145	0.294	0.123	0.027	0.034	0.094	0.039
compress	0.060	0.002	0.157	0.005	0.042	0.106	0.101	0.102
ab	0.139	0.130	0.261	0.098	0.091	0.024	0.121	0.020
espresso	0.009	0.012	0.026	0.011	0.003	0.007	0.011	0.008
lisp	0.002	0.001	0.013	0.011	0.003	0.004	0.006	0.003
eqntott	0.001	0.000	0.003	0.000	0.005	0.147	0.006	0.147
fpppp	0.050	0.184	0.040	0.173	0.002	0.005	0.005	0.005
doduc	0.014	0.277	0.020	0.270	0.002	0.023	0.006	0.022
liv	0.013	0.000	0.045	0.000	0.010	0.001	0.018	0.000
tomcatv	0.000	0.000	0.002	0.000	0.005	0.634	0.005	0.634

Assertions

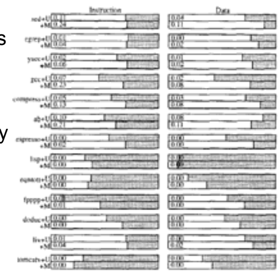
3. **Competition between user and system code is not a problem**
   - Few conflicts between user and system caching.
   - TLB misses are not a relevant factor
   - Note: the hardware used has direct-mapped physical caches.

   ⇒ Split system/user caches wouldn't help.



Self-Interference

- Only examine system cache misses.
- Shaded: System cache misses removed by associativity.
- MCPI for system-only, using R3000 direct-mapped cache.
- Reductions due to associativity were obtained by running system on a simulator and using a two-way associative cache of the same size.



Assertions…

4. **Self-interference is a problem in system instruction reference streams.**
   - High internal conflicts in system code.
   - System would benefit from higher cache associativity.
5. **System block memory operations are responsible for a large percentage of memory system reference costs.**
   - Particularly true for I/O system calls.
6. **Write buffers are less effective for system references.**
   - write buffer allows limited asynch. writes on cache misses
7. **Virtual to physical mapping strategy can have significant impact on cache performance**
   - Unfortunate mapping may increase conflict misses.
   - "Random" mappings (Mach) less likely to exhibit consistently poor performance.

Other Experience With Microkernel Performance

- System call costs are (inherently?) high.
  - Typically hundreds of cycles, 900 for Mach/i486.
- Context (address-space) switching costs are (inherently?) high.
  - Getting worse (in terms of cycles) with increasing CPU/memory speed ratios [Ousterhout, 1990].
  - IPC (involving system calls and context switches) is inherently expensive.

So, What's Wrong?

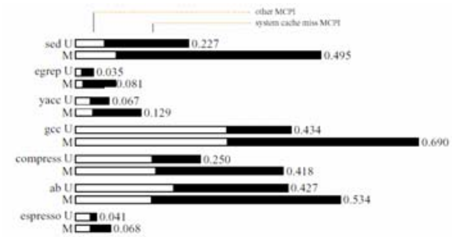
- The MCPI for Mach is significantly higher than Ultrix
- μ-kernels heavily depend on IPC
- IPC is expensive
  - Is the μ-kernel idea flawed?
  - Should some code never leave the kernel?
  - Do we have to buy flexibility with performance?

A Critique Of The Critique

- Data presented earlier:
  - are specific to one (or a few) system,
  - results cannot be generalised without thorough analysis,
  - no such analysis has been done.

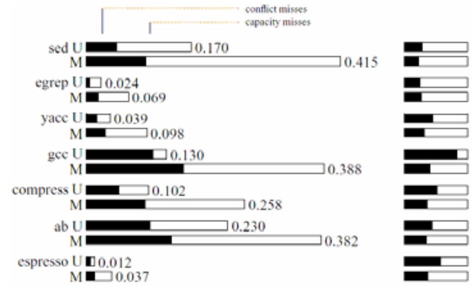
⇒ Cannot trust the conclusions [Liedkte, 1995].

Re-analysis Of Chen & Bershad's Data



MCPI for Ultrix and Mach

Re-analysis Of Chen & Bershad's Data…



MCPI caused by cache misses: conflict (black) vs capacity (white)

Conclusion

- Mach system (kernel + UNIX server + emulation library) is too big!
- UNIX server is essentially same.
- Emulation library is irrelevant (according to Chan & Bershad).

⇒ Mach μ-kernel working set is too big

Can we build μ-kernels which avoid these problems?

Requirements For Microkernels:

- Fast (system call costs, IPC costs)
- Small (big ⇒ slow)

⇒ Must be well designed, providing a minimal set of operations.

Can this be done?

Are High System Costs Essential?

- Example: kernel call cost on i486
  - Mach kernel call: 900 cycles
  - Inherent (hardware-dictated cost): 107 cycles.

    ⇒ 800 cycles kernel overhead.
  - L4 kernel call: 123–180 cycles (15–73 cycles overhead).

  ⇒ Mach's performance is a result of design and implementation **not** the μ-kernel concept!

Microkernel Design Principles (Liedtke)

- **Minimality:** If it doesn't *have to be* in the kernel, it *shouldn't* be in the kernel
  - Security is the only case for *must be in the kernel*
- **Appropriate abstractions** which can be made fast **and** allow efficient implementation of services
- **Well written:** It pays to shave a few cycles off TLB refill handler or the IPC path
- **Unportable:** must be targeted to specific hardware
  - no problem if it's small, and higher layers are portable
  - Example: Liedtke reports significant rewrite of memory management when porting from 486 to Pentium

  ⇒ "abstract hardware layer" is too costly

NON-PORTABILITY EXAMPLE: I486 VS PENTIUM:

- Size and associativity of TLB
- Size and organisation of cache (larger line size - restructured IPC)
- Segment regs in Pentium used to simulate tagged TLB

$\Rightarrow$ different trade-offs

WHAT *must* A μ-KERNEL PROVIDE?

- Virtual memory/address spaces
- threads,
- *fast* IPC,
- unique identifiers (for IPC addressing).

μ-KERNEL DOES *not* HAVE TO PROVIDE:

- file system
  - use user-level server (as in Mach)
- device drivers
  - user-level driver invoked via interrupt (= IPC)
- page-fault handler
  - use user-level pager

L4 Implementation Techniques

- Appropriate system calls to reduce number of kernel invocations
  - e.g., *reply & receive next*
- Rich message structure
  - value and reference parameters in message
- Copy message only once (i.e. **not** user!kernel!user)
- Short messages in registers
- As many syscall parameters in registers as possible
- One kernel stack (for interrupt handling) per thread (in TCB)
- TCBs in (mapped) VM, cache-friendly layout
- Thread UIDs (containing thread ID)
- "Hottest" kernel code is shortest
- Kernel IPC code on single page, critical data on single page
- Many H/W specific optimisations

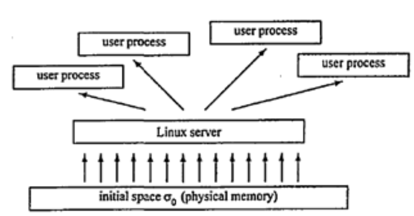
Performance

System	*CPU*	*MHz*	RPC μs	cyc/IPC	semantics
L4	R4600	100	1.7 μs	100	full
L4	Alpha	433	0.2 μs	45	full
L4	Pentium	166	1.5 μs	121	full
L4	486	50	10 μs	250	full
QNX	486	33	76 μs	1254	full
Mach	R2000	16.7	190 μs	1584	full
SCR RPC	CVAX	12.5	464 μs	2900	full
Mach	486	50	230 μs	5750	full
Amoeba	68020	15	800 μs	6000	full
Spin	Alpha 21064	133	102 μs	6783	full
Mach	Alpha 21064	133	104 μs	6916	full
Exo-tlrpc	R2000	116.7	6 μs	53	restricted
Spring	SparcV8	40	11 μs	220	restricted
DP-Mach	486	66	16 μs	528	restricted
LRPC	CVAX	12.5	157 μs	981	restricted

Case In Point: L⁴Linux [Härtig *et al.,* 1997]

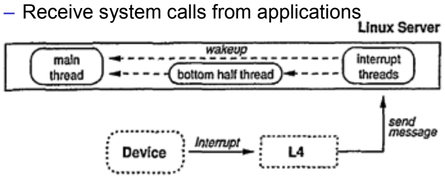
- Port of Linux kernel to L4 (like Mach Unix server)
  - single-threaded (for simplicity, **not** performance)
  - is pager of all Linux user processes
  - maps emulation library and signal-handling code into AS
  - server AS maps physical memory (& Linux runs within)
  - copying between user and server done on physical memory
  - use software lookup of page tables for address translation
- Changes to Linux restricted to architecture-dependent part
- Duplication of page tables (L4 and Linux server)
- Binary compatible to native Linux via trampoline mechanism
  - but also modified libc with RPC stubs

L⁴Linux Overview



Server Internals

- L4 threads used to
  - receive device interrupts
  - Emulated Linux's *bottom half* handling
  - Receive system calls from applications

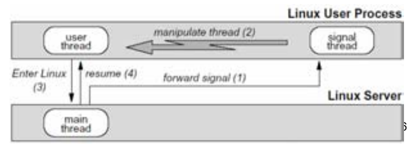


THE UNIVERSITY OF NEW SOUTH WALES

25

Signal Delivery In L^{4}Linux

- Separate signal-handler thread in each user process
  - server IPCs signal-handler thread
  - handler thread ex regs main user thread to save state
  - user thread IPCs Linux server
  - server does signal processing
  - server IPCs user thread to resume



THE UNIVERSITY OF NEW SOUTH WALES

L^{4}Linux Performance

Microbenchmarks:

System	*Time [μs]*	*Cycles*
Linux	1.68	223
L^{4}Linux	3.95	526
L^{4}Linux (trampoline)	5.66	753
MkLinux in-kernel	15.66	2050
MkLinux server	110.60	14710

`getpid()` on 133MHz Pentium

THE UNIVERSITY OF NEW SOUTH WALES

27

Cycle Breakdown

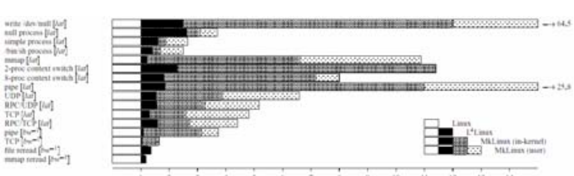
Client	*Cycles*	*Server*
enter emulation lib	20	
send syscall message	168	wait for msg
	131	Linux kernel
receive reply	188	send reply
leave emulation lib	19	

Hardware cost: 82 cycles

THE UNIVERSITY OF NEW SOUTH WALES

28

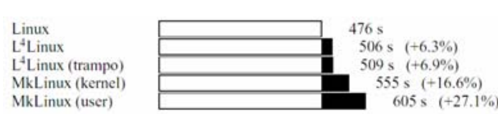
Macrobenchmarks: LMBENCH



THE UNIVERSITY OF NEW SOUTH WALES

29

Macrobenchmarks: Kernel Compile



THE UNIVERSITY OF NEW SOUTH WALES

30

cse

Conclusion

- Mach sux $\nRightarrow$ microkernels suck
- L4 shows that performance *might* be deliverable
  - L^{4}Linux gets close to monolithic kernel performance
  - need real multi-server system to evaluate μ-kernel potential
- Jury is still out!
- Mach has prejudiced community (see Linus...)
  - It'll be an uphill battle!

THE UNIVERSITY OF NEW SOUTH WALES

31

cse

Implementations

API	Kernel	Who	Language	CPU
V2	L4/x86	Liedtke	asm	x86
	L4/MIPS	UNSW	asm/C	R4k
	L4/Alpha	UNSW/Dres	PAL/C	21x64
	Fiasco	Dresden	C++	x86
X.0	L4/x86	Liedtke	asm	x86
	Hazelnut	Karlsruhe	C	x86, ARM
V4	Pistachio	Karlsruhe	C++	x86, IA-64
				PPC-32
		UNSW		MIPS, Alpha
				ARM, PPC-64
				SPARC (i.p.)