

# CRITIQUE OF MICROKERNEL ARCHITECTURES

# 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*

# 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?

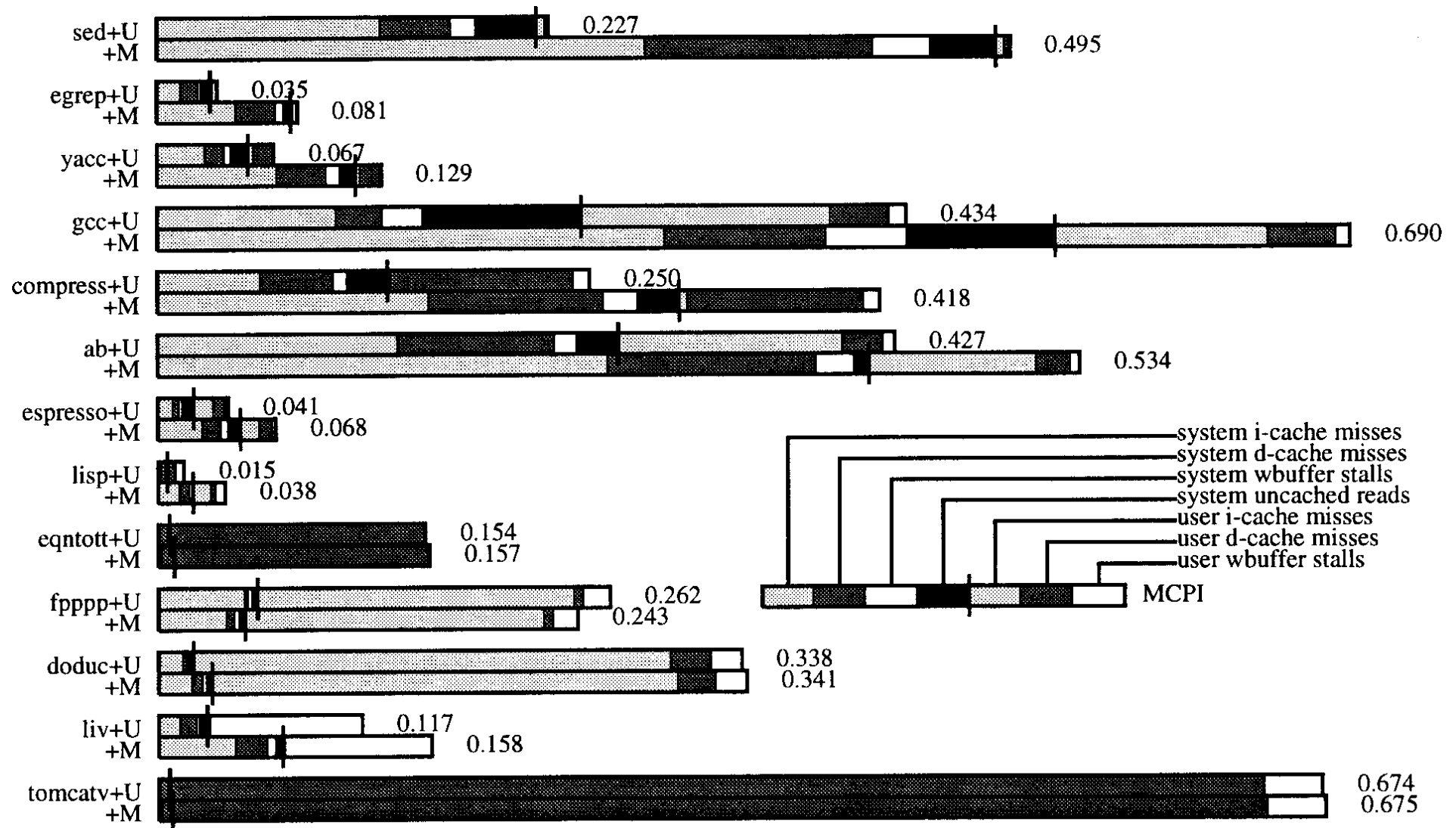
# MICROKERNEL PERFORMANCE

- First generation  $\mu$ -kernel systems exhibited poor performance when compared to monolithic UNIX implementations.
  - particularly Mach, the best-known example

# MICROKERNEL PERFORMANCE

- First generation  $\mu$ -kernel systems exhibited poor performance when compared to monolithic UNIX implementations.
  - particularly Mach, the best-known example
- Reasons are investigated by [Chen & Bershad 93]:
  - 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).

# 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.



# 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 [Ber92].

# 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

**1 OS has less instruction and data locality than user code.**

→ System code has higher cache and TLB miss rates.

→ Particularly bad for instructions.

**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.

# 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.

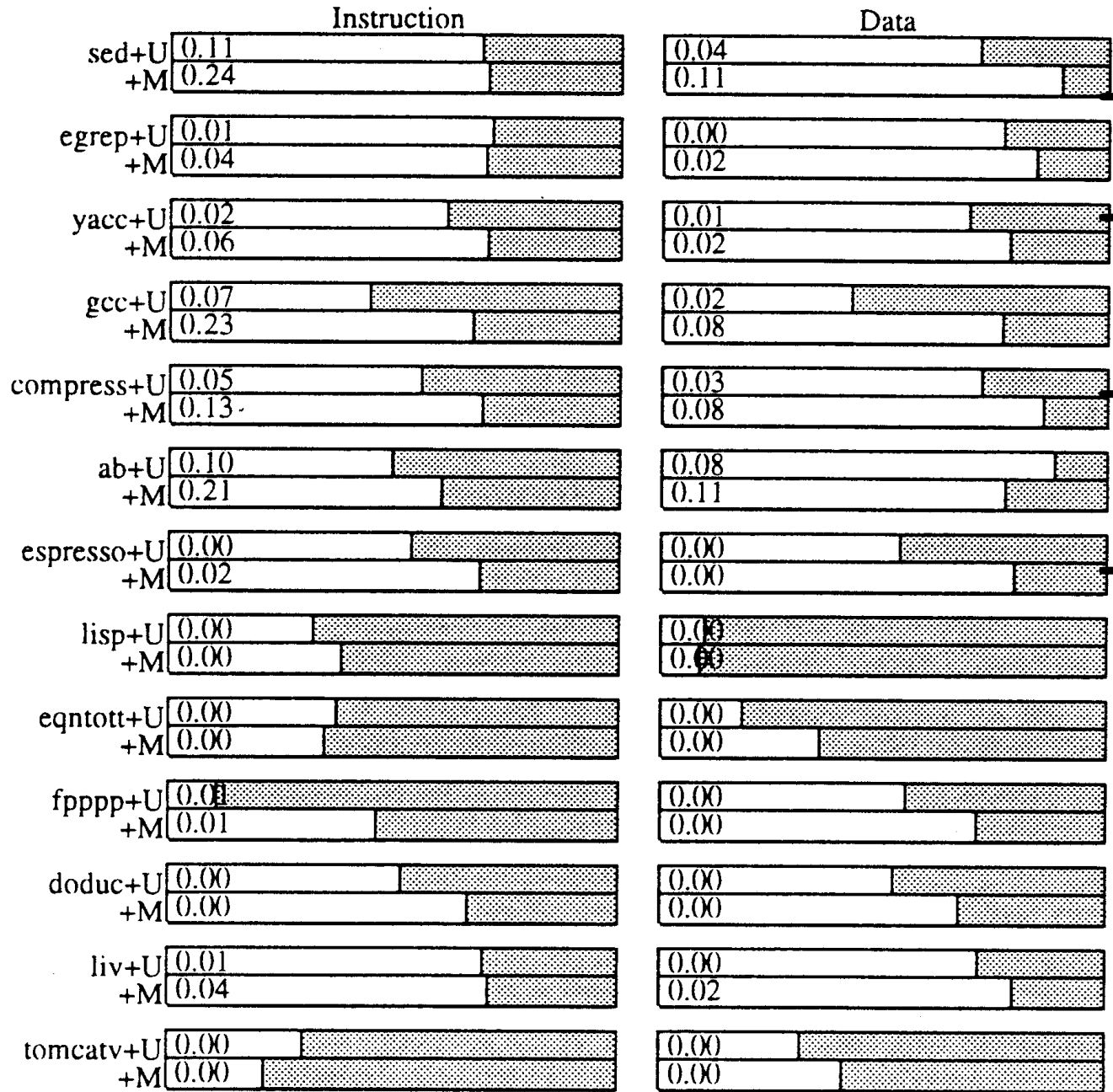
## 2 System execution is more dependent on instruction cache behaviour than is user execution

- MCPIs dominated by system i-cache misses.  
Note: most benchmarks were small, i.e. user code fits in cache.

## 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.

# 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.

# 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



# 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) are to be avoided.

## OTHER EXPERIENCE WITH $\mu$ -KERNEL 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 [Ous90].
  - IPC (involving system calls and context switches) is inherently expensive.

**So, WHAT'S WRONG?**

## SO, WHAT'S WRONG?

- $\mu$ -kernels heavily depend on IPC
- IPC is expensive

# SO, WHAT'S WRONG?

- $\mu$ -kernels heavily depend on IPC
- IPC is expensive
  - ★ Is the  $\mu$ -kernel idea flawed?
  - ★ Should some code never leave the kernel?
  - ★ Do we have to buy flexibility with performance?

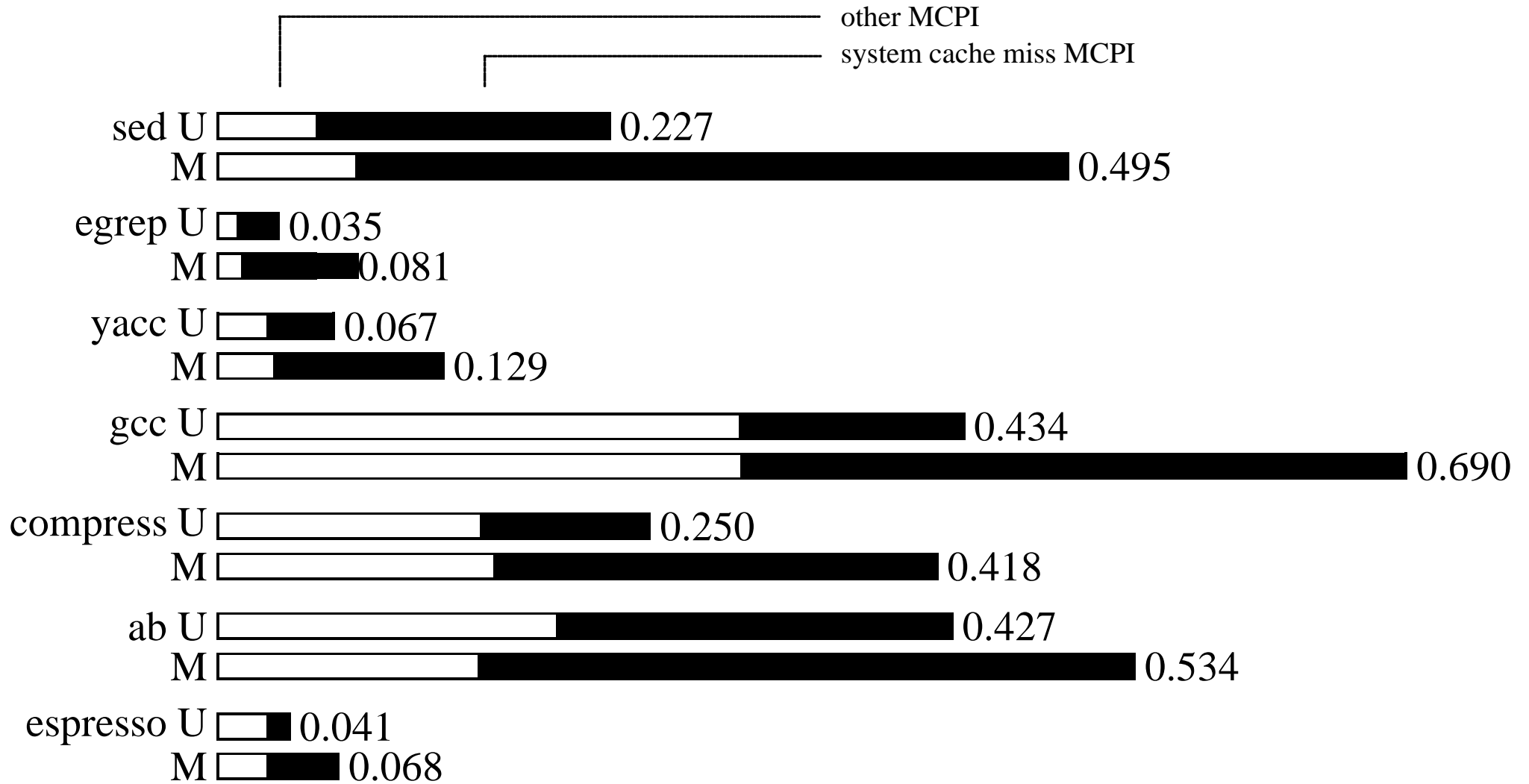
# A CRITIQUE OF THE CRITIQUE

Data presented earlier:

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

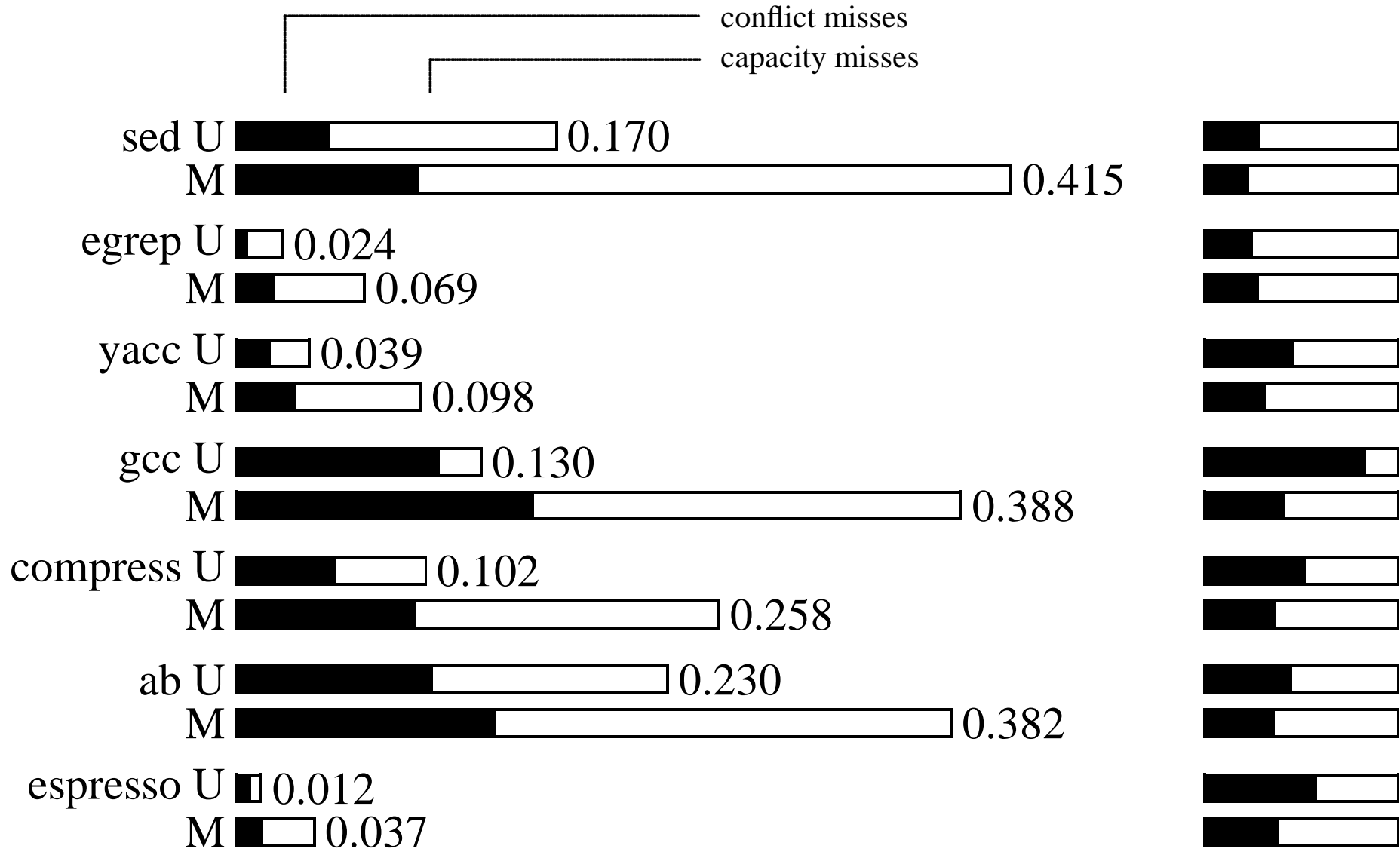
⇒ Cannot trust the conclusions [Lie95].

# 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!

# CONCLUSION

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

# 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  $\mu$ -kernel working set is too big

# 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  $\mu$ -kernel working set is too big

**Can we build  $\mu$ -kernels which avoid these problems?**

## REQUIREMENTS FOR $\mu$ -KERNELS:

- Fast (system call costs, IPC costs)
  - Small (big  $\Rightarrow$  slow)
- $\Rightarrow$  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

# 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.

# 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).



# 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  $\mu$ -kernel concept!

# $\mu$ -KERNEL DESIGN PRINCIPLES (LIEDTKE)

**Minimality:** If it doesn't *have to be* in the kernel, it *shouldn't* 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

⇒ different trade-offs

## WHAT *must* A $\mu$ -KERNEL PROVIDE?

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

## WHAT *must* A $\mu$ -KERNEL PROVIDE?

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

## $\mu$ -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*

## 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

## 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)



## 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

## 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

# 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)

# 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

# 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)

# 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

# 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

# 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

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

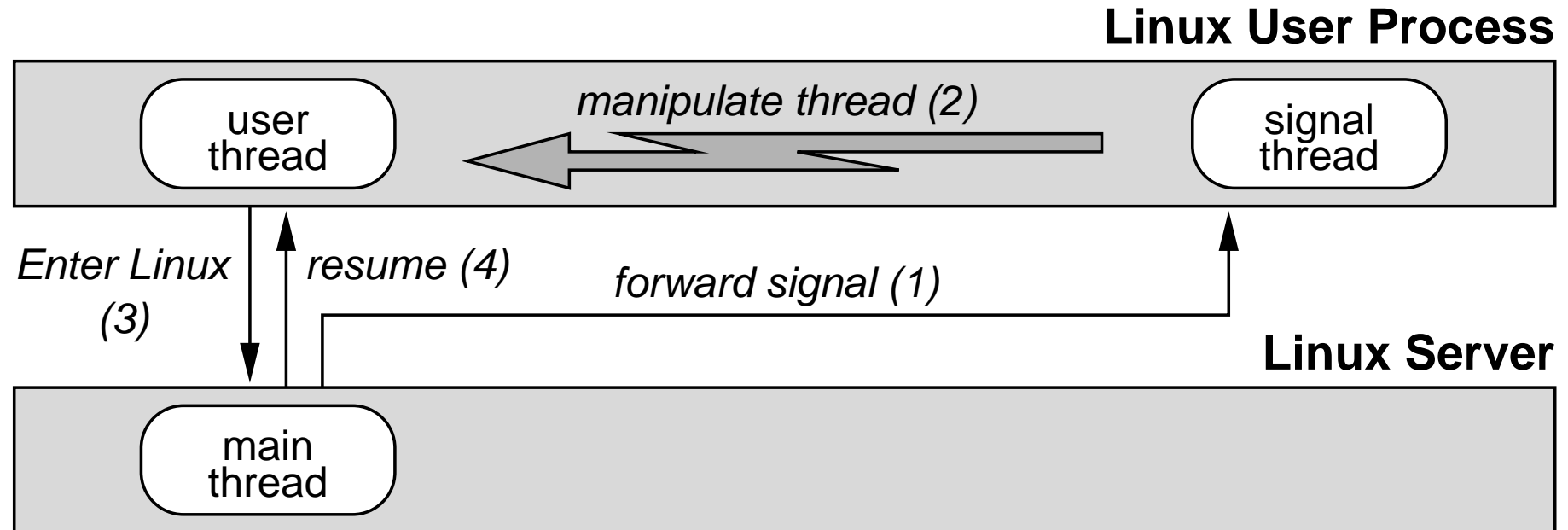
# CASE IN POINT: L<sup>4</sup>LINUX [HÄRTIG *et al.* 97]

- 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

# CASE IN POINT: L<sup>4</sup>LINUX [HÄRTIG *et al.* 97]

- 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

# SIGNAL DELIVERY IN L<sup>4</sup> 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

# L<sup>4</sup> LINUX PERFORMANCE

## MICROBENCHMARKS:

<i>System</i>	<i>Time [<math>\mu</math>s]</i>	<i>Cycles</i>
Linux	1.68	223

getpid( ) on 133MHz Pentium

# L<sup>4</sup>LINUX PERFORMANCE

## MICROBENCHMARKS:

<i>System</i>	<i>Time [<math>\mu</math>s]</i>	<i>Cycles</i>
Linux	1.68	223
L <sup>4</sup> Linux	3.95	526
L <sup>4</sup> Linux (trampoline)	5.66	753

getpid( ) on 133MHz Pentium

# L<sup>4</sup>LINUX PERFORMANCE

## MICROBENCHMARKS:

<i>System</i>	<i>Time [<math>\mu</math>s]</i>	<i>Cycles</i>
Linux	1.68	223
L <sup>4</sup> Linux	3.95	526
L <sup>4</sup> Linux (trampoline)	5.66	753
MkLinux in-kernel	15.66	2050
MkLinux server	110.60	14710

getpid( ) on 133MHz Pentium

## CYCLE BREAKDOWN:

<i>Client</i>	<i>Cycles</i>	<i>Server</i>
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



# MACROBENCHMARKS: LMBENCH

write /dev/null [lat]

null process [lat]

simple process [lat]

/bin/sh process [lat]

mmap [lat]

2-proc context switch [lat]

8-proc context switch [lat]

pipe [lat]

UDP [lat]

RPC/UDP [lat]

TCP [lat]

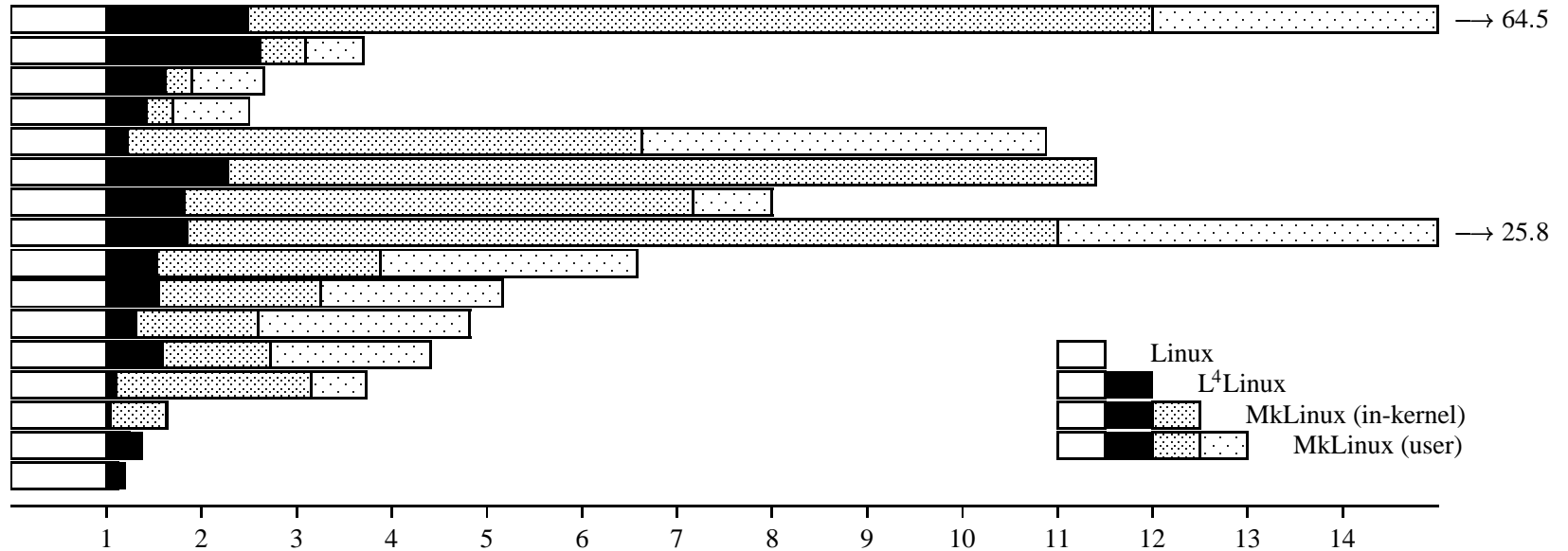
RPC/TCP [lat]

pipe [bw<sup>-1</sup>]

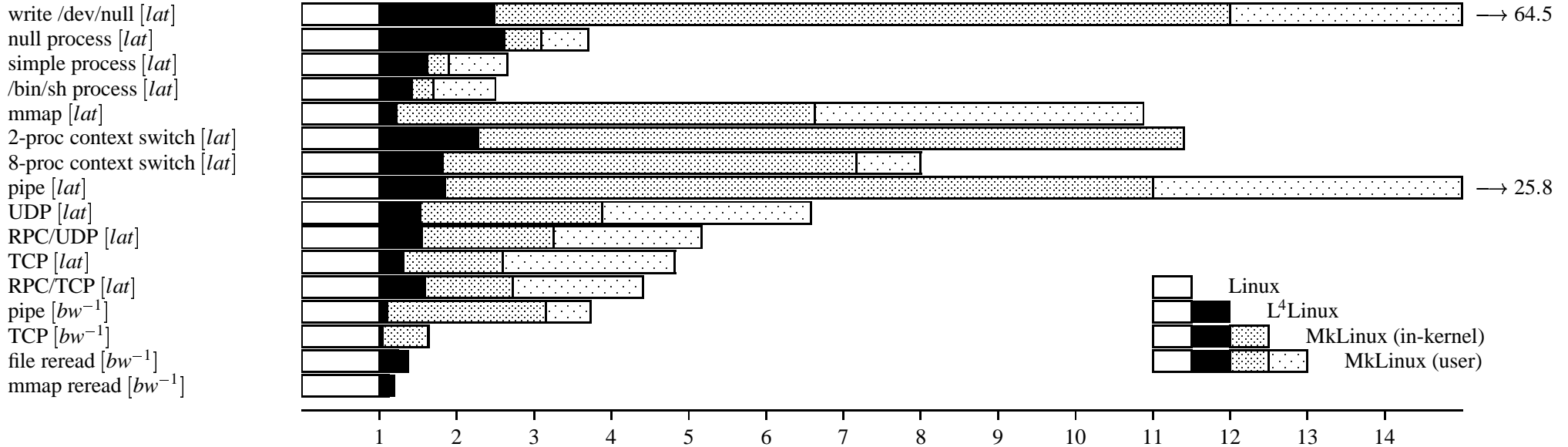
TCP [bw<sup>-1</sup>]

file reread [bw<sup>-1</sup>]

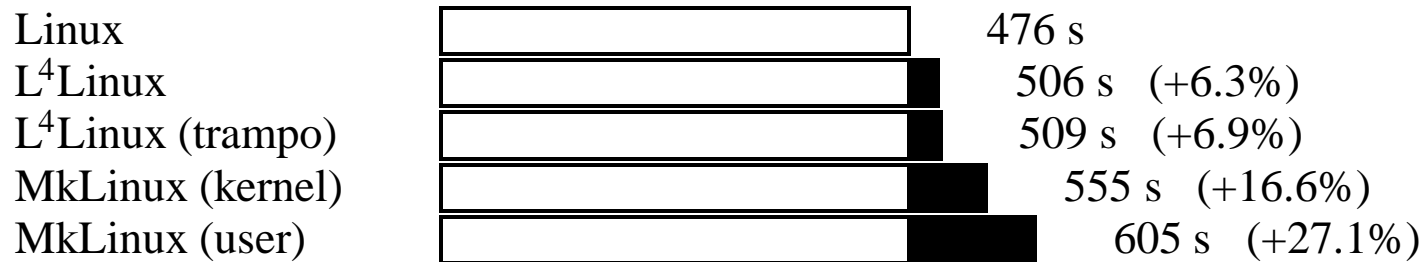
mmap reread [bw<sup>-1</sup>]



# MACROBENCHMARKS: LMBENCH



# MACROBENCHMARKS: KERNEL COMPILE



# CONCLUSIONS

- Mach sux  $\not\Rightarrow$  microkernels suck

# CONCLUSIONS

- Mach sux  $\nRightarrow$  microkernels suck
- L4 shows that performance *might* be deliverable
  - L<sup>4</sup>Linux gets close to monolithic kernel performance
  - need real multi-server system to evaluate  $\mu$ -kernel potential

# CONCLUSIONS

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

# L4 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  UNSW	C++	x86, IA-64 PPC-32 MIPS, Alpha ARM, PPC-64 SPARC (i.p.)

# References

- [Ber92] Brian N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proc. USENIX WS. Microkernels & other Kernel Arch.*, pages 205–211, Seattle, WA, USA, Apr 1992.
- [CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proc. 14th ACM SOSP*, pages 120–133, Asheville, NC, USA, Dec 1993.
- [HHL<sup>+</sup>97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of  $\mu$ -kernel-based systems. In *Proc. 16th ACM SOSP*, pages 66–77, St. Malo, France, Oct 1997.

- [Lie95] Jochen Liedtke. On  $\mu$ -kernel construction. In *Proc. 15th ACM SOSOP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.
- [Ous90] J.K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. 1990 Summer USENIX Techn. Conf.*, pages 247–56, Jun 1990.