Microkernels
In a Bit More Depth

Gernot Heiser

COMP9242 2005/S2 Week 5

MOTIVATION

- Early operating systems had very little structure.
- A strictly layered approach was promoted by [Dijkstra 1968].
- Later OS (more or less) followed that approach (e.g., Unix).
- Such systems are known as *monolithic kernels*

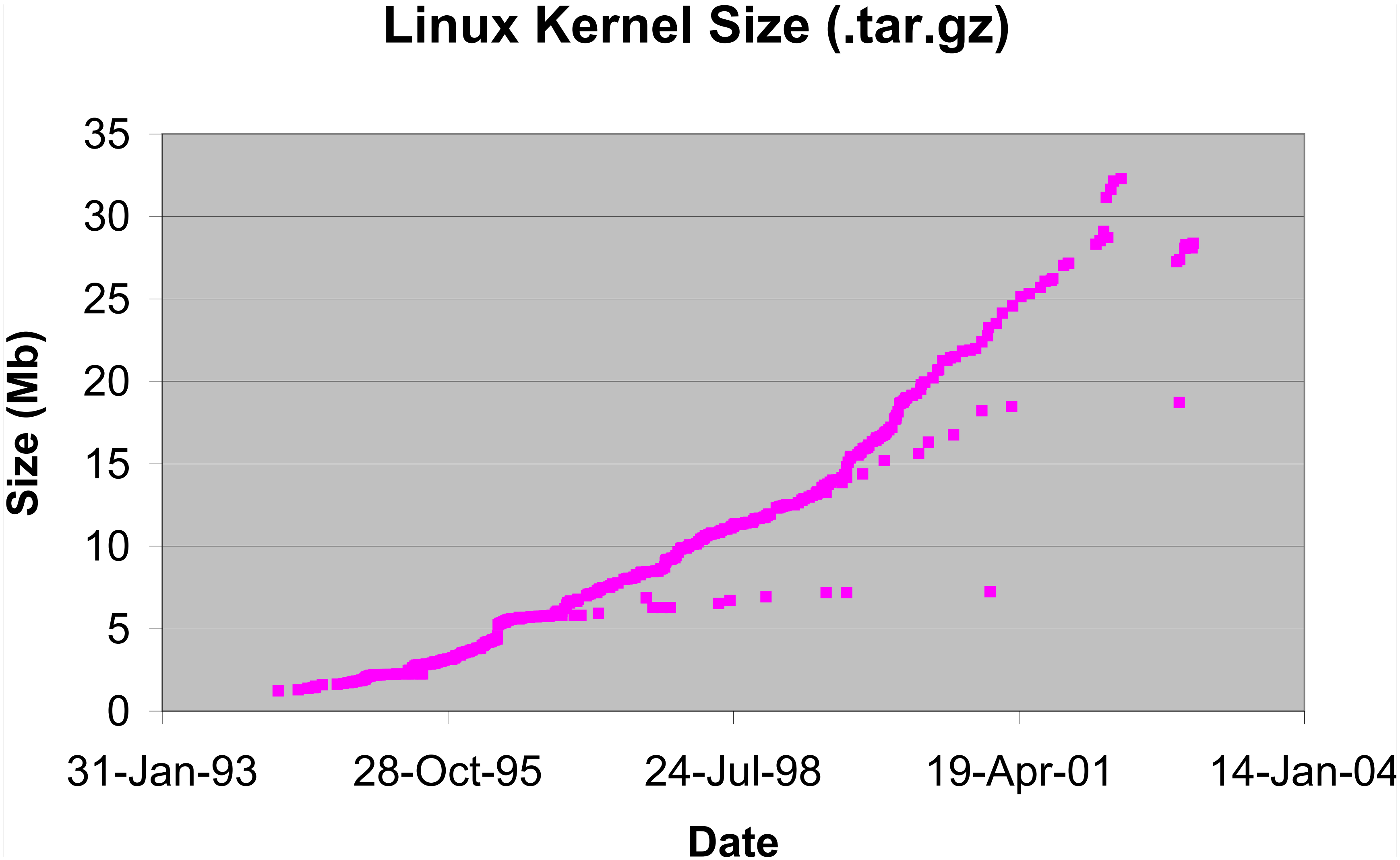
ADVANTAGES OF MONOLITHIC KERNELS

- Kernel has access to everything:
  - ➜ all optimisations possible
  - ➜ all techniques/mechanisms/concepts implementable
- Kernel can be extended by adding more code, e.g. for:
  - ➜ new services
  - ➜ support for new hardare

PROBLEMS WITH LAYERED APPROACH

- Widening range of services and applications
  $\Rightarrow$ OS bigger, more complex, slower, more error prone.
- Need to support same OS on different hardware.
- Like to support various OS environments.
- Distribution
  $\Rightarrow$ impossible to provide all services from same (local) kernel.

EVOLUTION OF THE LINUX KERNEL



Linux 2.4.18: 2.7M LoC

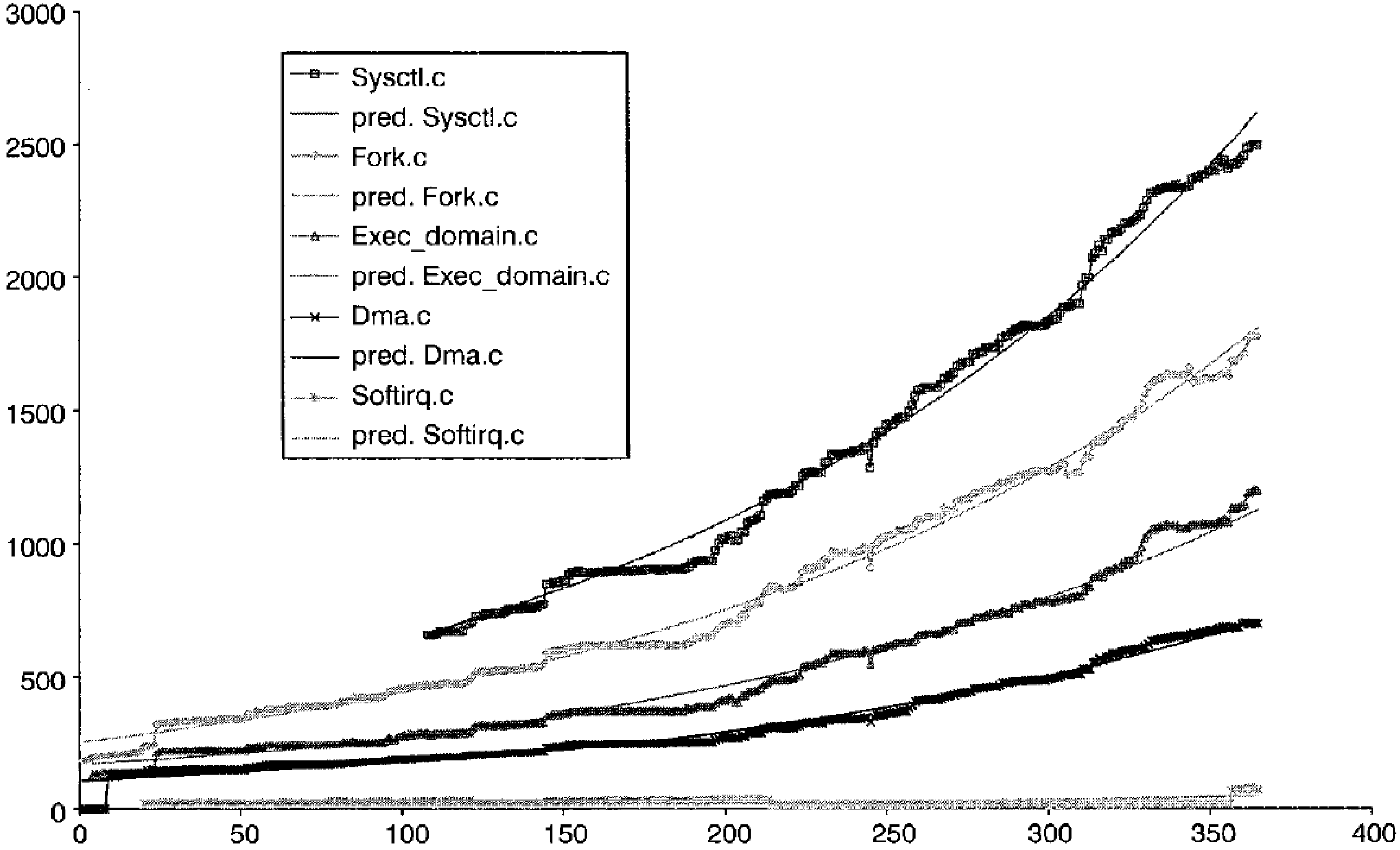
APPROACHES TO TACKLING COMPLEXITY

- Classical software-engineering approach: modularity
  - ➜ (relatively) small, mostly self-contained components
  - ➜ well-defined interfaces between them
  - ➜ enforcement of interfaces
  - ➜ containment of faults to few modules

APPROACHES TO TACKLING COMPLEXITY

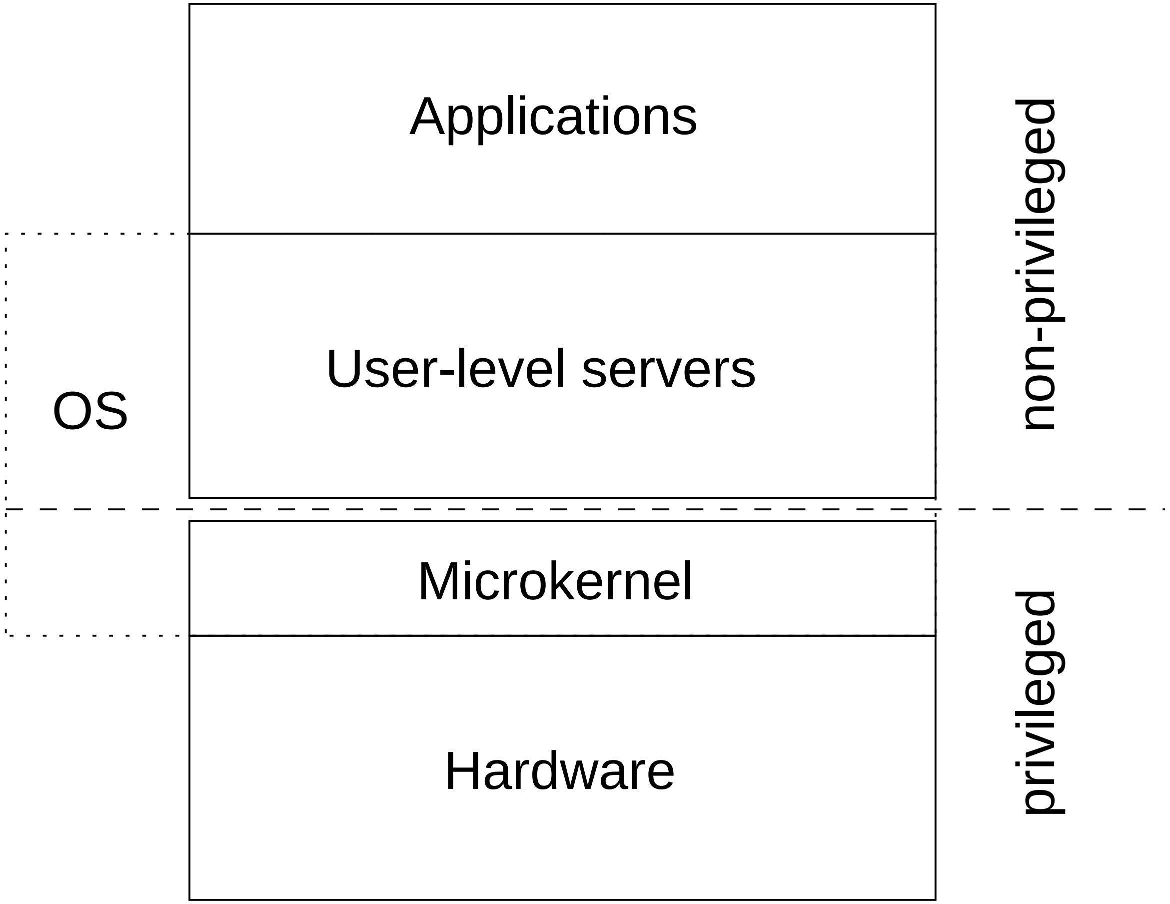
- Classical software-engineering approach: modularity
  - ➜ (relatively) small, mostly self-contained components
  - ➜ well-defined interfaces between them
  - ➜ enforcement of interfaces
  - ➜ containment of faults to few modules
- Doesn’t work with monolithic kernels:
  - ➜ all kernel code executes in privileged mode
    - ➜ faults aren’t contained
    - ➜ interfaces cannot be enforced
    - ➜ performance takes priority over structure

EVOLUTION OF THE LINUX KERNEL — PART 2



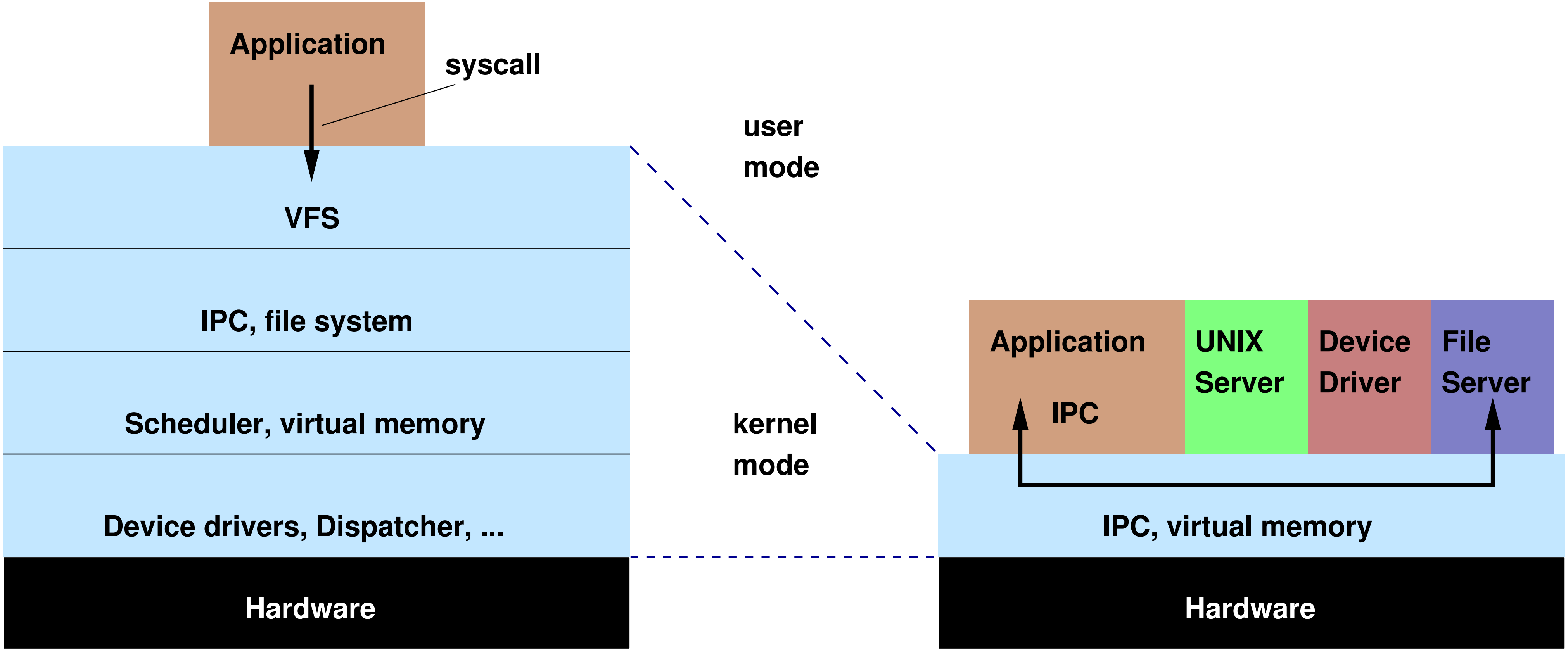
Dependencies between kernel modules by global variables ("common coupling") as a function of version number [SJW$^+$02]: *Exponential growth!*

MICROKERNEL IDEA: BREAK UP THE OS



Based on the ideas of Brinch Hansen's "Nucleus" [BH70]

MONOLITHIC VS. MICROKERNEL OS STRUCTURE



MICROKERNEL OS

- Kernel:
  - ⋆ contains code which *must* run in supervisor mode;
  - ⋆ isolates hardware dependence from higher levels;
  - ⋆ is small and fast
  - ⇒ extensible system;

MICROKERNEL OS

- Kernel:
  - ⋆ contains code which *must* run in supervisor mode;
  - ⋆ isolates hardware dependence from higher levels;
  - ⋆ is small and fast
  - ⇒ extensible system;
  - ⋆ provides *mechanisms*.

MICROKERNEL OS

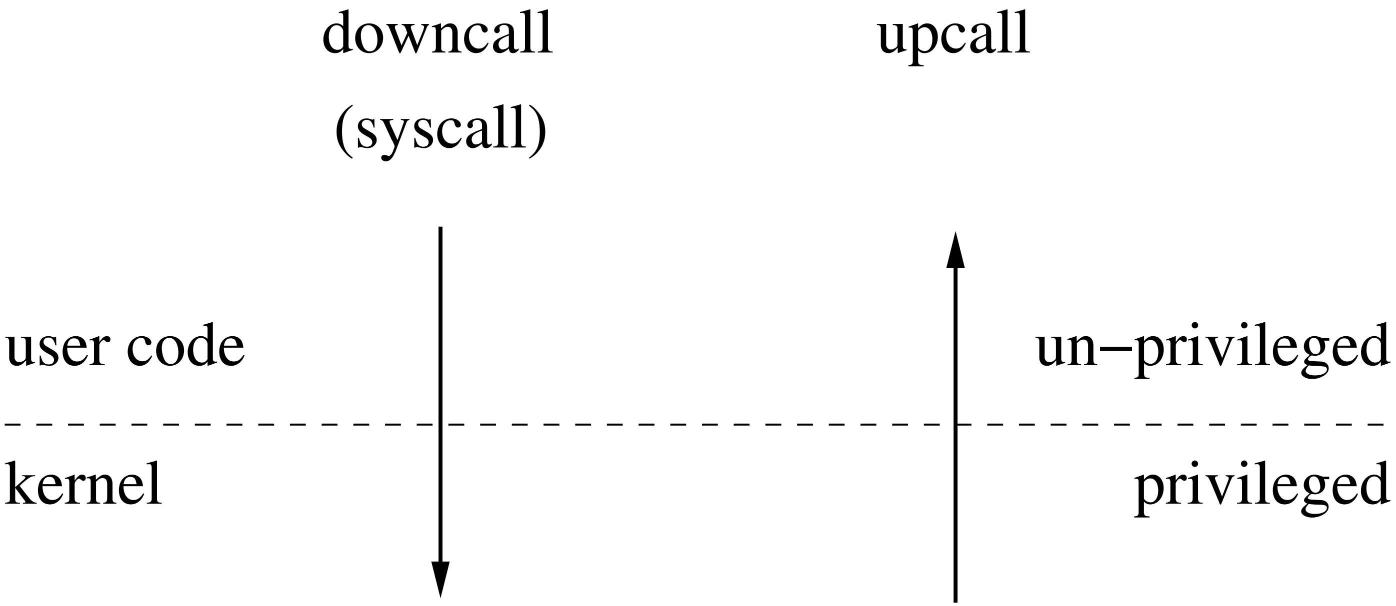
- Kernel:
  - ⋆ contains code which *must* run in supervisor mode;
  - ⋆ isolates hardware dependence from higher levels;
  - ⋆ is small and fast
  - ⇒ extensible system;
  - ⋆ provides *mechanisms*.

- User-level servers:
  - ⋆ are hardware independent/portable,
  - ⋆ provide “OS environment”/”OS personality” (maybe several),
  - ⋆ may be invoked:
    - ➜ from **application** (via message-passing IPC)
    - ➜ from **kernel** (upcalls);

MICROKERNEL OS

- Kernel:
  - ⋆ contains code which *must* run in supervisor mode;
  - ⋆ isolates hardware dependence from higher levels;
  - ⋆ is small and fast
  - ⇒ extensible system;
  - ⋆ provides *mechanisms*.
- User-level servers:
  - ⋆ are hardware independent/portable,
  - ⋆ provide “OS environment”/”OS personality” (maybe several),
  - ⋆ may be invoked:
    - ➜ from **application** (via message-passing IPC)
    - ➜ from **kernel** (upcalls);
  - ⋆ implement *policies* [BH70].

DOWNCALL VS. UPCALL



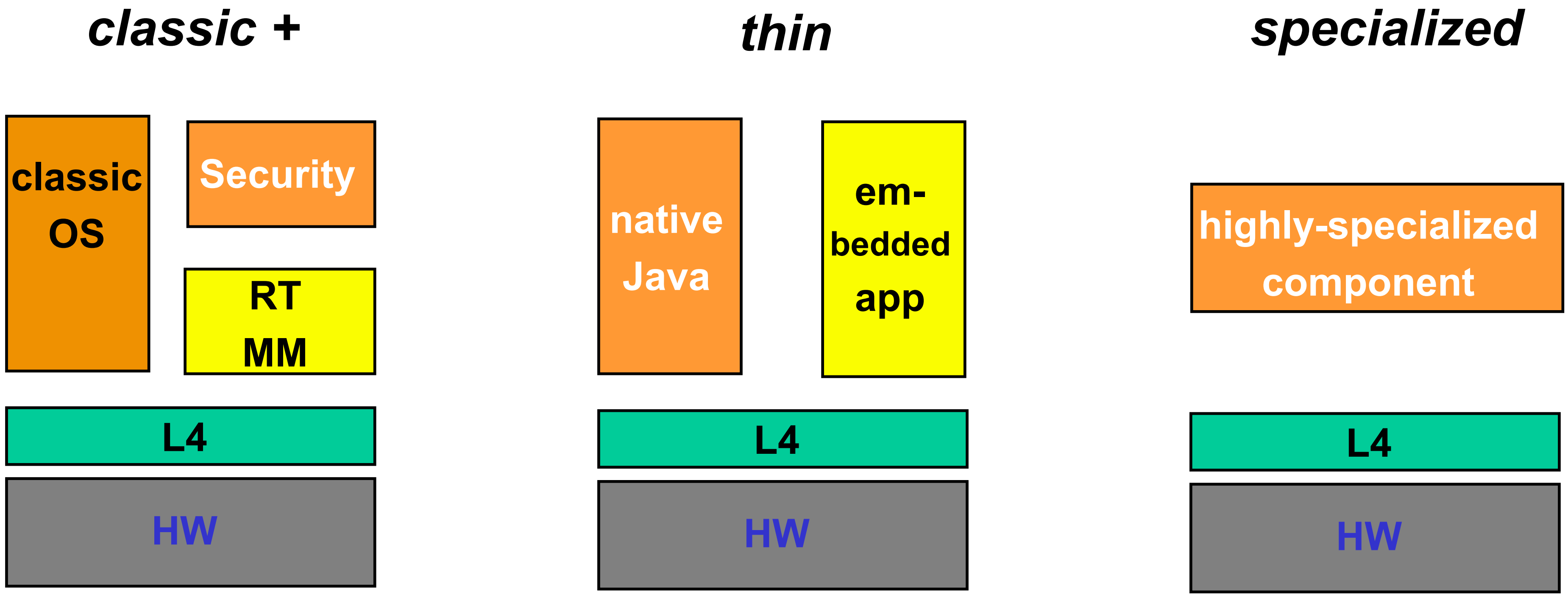
unprivileged code
enters kernel mode

implemented via trap

privileged code
enters user mode

implemented via IPC

MICROKERNEL-BASED SYSTEMS



EARLY EXAMPLE: HYDRA

- Separation of mechanism from policy
  - ➜ e.g. protection vs. security
- No hierarchical layering of kernel.
- Protection, even within OS.
  - ➜ Uses (segregated) *capabilities*.
- Objects, encapsuation, units of protection.
- Unique object *name*, no ownership.
- Object persistence based on reference counting [WCC$^+$74].

HYDRA ...

- Can be considered the first *object-oriented* OS;
- Has been called the first *microkernel* OS (by people who ignored Brinch Hansen)
- Has had enormous influence on later operating systems research;
- Was never widely used even at CMU because of
  - ➜ poor performance,
  - ➜ lack of a complete environment.

POPULAR EXAMPLE: MACH

- Developed at CMU by Rashid and others [RTY$^+$88] from 1984
- successor of Accent [FR86] and RIG [Ras88].

POPULAR EXAMPLE: MACH

- Developed at CMU by Rashid and others [RTY$^+$88] from 1984
- successor of Accent [FR86] and RIG [Ras88].

GOALS:

- *Tailorability:* support different OS interfaces
- *Portability:* almost all code H/W independent
- *Real-time capability*
- *Multiprocessor and distribution* support
- *Security*

POPULAR EXAMPLE: MACH

- Developed at CMU by Rashid and others [RTY$^+$88] from 1984
- successor of Accent [FR86] and RIG [Ras88].

GOALS:

- *Tailorability:* support different OS interfaces
- *Portability:* almost all code H/W independent
- *Real-time capability*
- *Multiprocessor and distribution* support
- *Security*

Coined term *microkernel*.

BASIC FEATURES OF MACH KERNEL

- Task and thread management
- Interprocess communication (asynchronous message-passing)
- Memory object management
- System call redirection
- Device support
- Multicomputer support

MACH TASKS AND THREADS

- Task consists of one or more threads
- Task provides *address space* and other environment
- Thread is active entity (basic unit of CPU utilisation)
- Threads have own stacks, are kernel scheduled
- Threads may run in parallel on multiprocessor
- “Privileged user-state program” may be used to control scheduling
- Task created from “blueprint” with empty or inherited address space
- Activated by creating a thread in it

MACH IPC: PORTS

- Addressing based on ports:
  - ➜ port is a mailbox, allocated/destroyed via a system call
  - ➜ has a fixed-size message queue associated with it
  - ➜ is protected by (segregated) capabilities
  - ➜ has exactly *one receiver*, but possibly *many senders*
  - ➜ can have "send-once" capability to a port

MACH IPC: PORTS

- Addressing based on ports:
  - ➜ port is a mailbox, allocated/destroyed via a system call
  - ➜ has a fixed-size message queue associated with it
  - ➜ is protected by (segregated) capabilities
  - ➜ has exactly *one receiver*, but possibly *many senders*
  - ➜ can have “send-once” capability to a port
- Can pass the *receive capability* for a port to another process
  - ➜ give up read access to the port

MACH IPC: PORTS

- Addressing based on ports:
  - ➜ port is a mailbox, allocated/destroyed via a system call
  - ➜ has a fixed-size message queue associated with it
  - ➜ is protected by (segregated) capabilities
  - ➜ has exactly *one receiver*, but possibly *many senders*
  - ➜ can have “send-once” capability to a port
- Can pass the *receive capability* for a port to another process
  - ➜ give up read access to the port
- Kernel detects ports without senders or receiver
- Processes may have many ports (UNIX server has 2000!)
- Ports can be grouped into *port sets*
  - ➜ Allows listening to many ports (like `select()`)

MACH IPC: PORTS

- Addressing based on ports:
  - ➜ port is a mailbox, allocated/destroyed via a system call
  - ➜ has a fixed-size message queue associated with it
  - ➜ is protected by (segregated) capabilities
  - ➜ has exactly *one receiver*, but possibly *many senders*
  - ➜ can have “send-once” capability to a port
- Can pass the *receive capability* for a port to another process
  - ➜ give up read access to the port
- Kernel detects ports without senders or receiver
- Processes may have many ports (UNIX server has 2000!)
- Ports can be grouped into *port sets*
  - ➜ Allows listening to many ports (like `select()`)
- Send blocks if queue is full
  - ➜ except with send-once cap (used for server replies)

MACH IPC: MESSAGES

- Segregated capabilities:
  - ➜ threads refer to them via local indices
  - ➜ kernel marshalls capabilities in messages
  - ➜ message format must identify caps

MACH IPC: MESSAGES

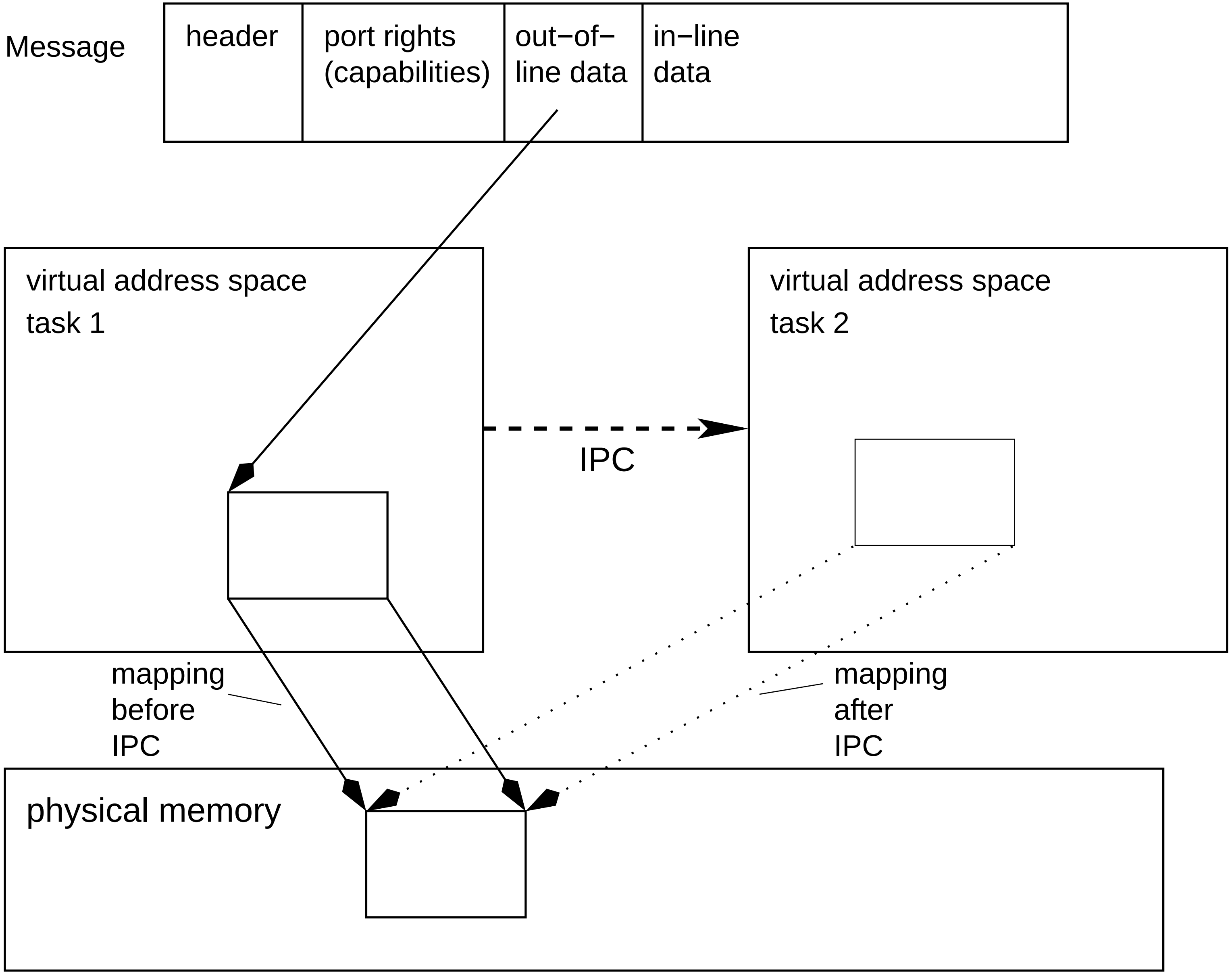
- Segregated capabilities:
  - ➜ threads refer to them via local indices
  - ➜ kernel marshalls capabilities in messages
  - ➜ message format must identify caps

- Message contents:
  - ⋆ Send capability to destination port (mandatory)
    - ➜ used by kernel to validate operation
  - ⋆ optional send capability to reply port
    - ➜ for use by receiver to send reply
  - ⋆ possibly other capabilities

MACH IPC: MESSAGES

- Segregated capabilities:
  - ➜ threads refer to them via local indices
  - ➜ kernel marshalls capabilities in messages
  - ➜ message format must identify caps
- Message contents:
  - ⋆ Send capability to destination port (mandatory)
    - ➜ used by kernel to validate operation
  - ⋆ optional send capability to reply port
    - ➜ for use by receiver to send reply
  - ⋆ possibly other capabilities
  - ⋆ “in-line” (by-value) data
  - ⋆ “out-of-line” (by reference) data, using copy-on-write,
    - ➜ may contain whole address spaces

MACH IPC


Message
header
port rights (capabilities)
out−of−line data
in−line data
virtual address space task 1
virtual address space task 2
IPC
mapping before IPC
mapping after IPC
physical memory

MACH VIRTUAL MEMORY MANAGEMENT

- Address space constructed from *memory regions*
  - initially empty

MACH VIRTUAL MEMORY MANAGEMENT

- Address space constructed from *memory regions*
  - ⋆ initially empty
  - ⋆ populated by:
    - ➜ explicit allocation
    - ➜ explicitly mapping a *memory object*
    - ➜ inheriting from “blueprint” (as in Linux clone()),
      - ➜ inheritance: *not*, *shared* or *copied*
    - ➜ allocated automatically by kernel during IPC
      - ➜ when passing by-reference parameters
    - ⇒ sparse virtual memory use (unlike UNIX)

MACH VIRTUAL MEMORY MANAGEMENT

- Address space constructed from *memory regions*
  - ⋆ initially empty
  - ⋆ populated by:
    - ➜ explicit allocation
    - ➜ explicitly mapping a *memory object*
    - ➜ inheriting from “blueprint” (as in Linux clone()),
      - ➜ inheritance: *not*, *shared* or *copied*
    - ➜ allocated automatically by kernel during IPC
      - ➜ when passing by-reference parameters
    - ⇒ sparse virtual memory use (unlike UNIX)
  - ⋆ 3 page states:
    - ➜ unallocated,
    - ➜ allocated & unreferenced,
    - ➜ allocated & initialised

COPY-ON-WRITE IN MACH

- When data is copied (“blueprint” or passed by-reference):
  - ➜ source and destination share single copy,
  - ➜ both virtual pages are mapped to the same frame
- Marked as read-only
- When one copy is modified, a fault occurs
- Handling by kernel involves making a physical copy is made,
  - ➜ VM mapping is changed to refer to the new copy

COPY-ON-WRITE IN MACH

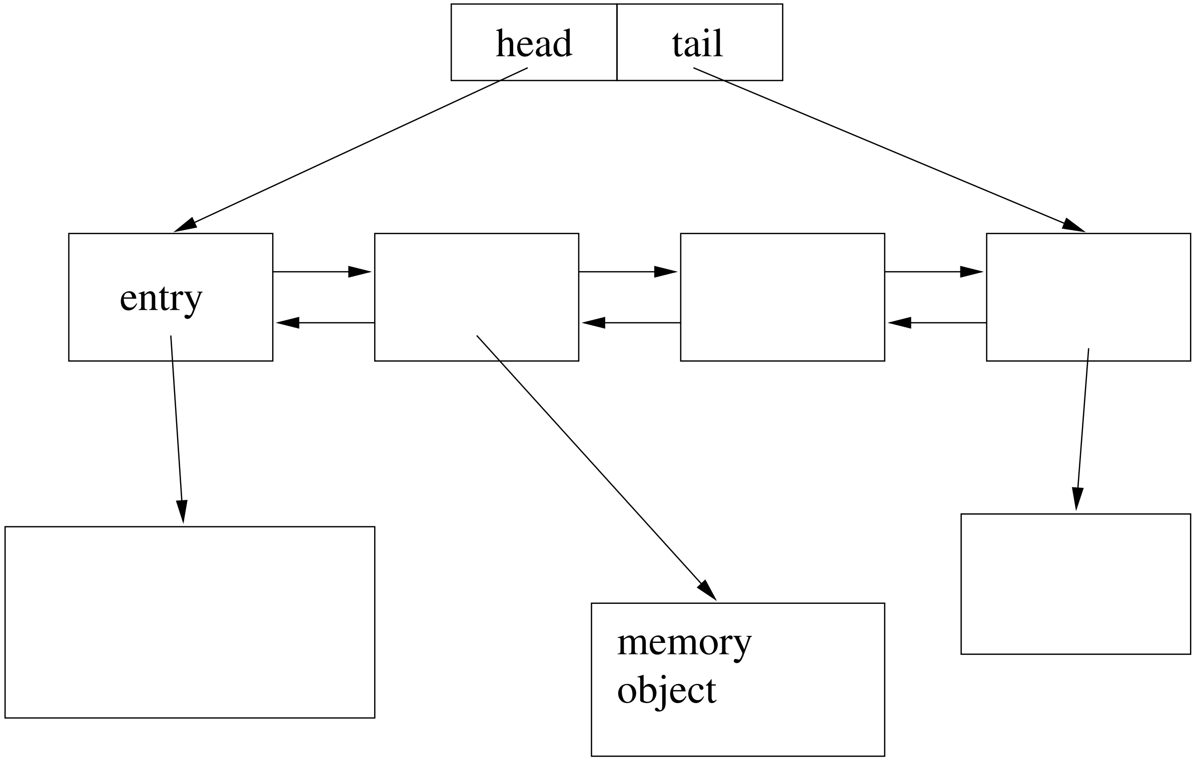
- When data is copied (“blueprint” or passed by-reference):
  - ➜ source and destination share single copy,
  - ➜ both virtual pages are mapped to the same frame
- Marked as read-only
- When one copy is modified, a fault occurs
- Handling by kernel involves making a physical copy is made,
  - ➜ VM mapping is changed to refer to the new copy
- Advantage:
  - ➜ efficient way of sharing/passing large amounts of data

COPY-ON-WRITE IN MACH

- When data is copied ("blueprint" or passed by-reference):
  - ➜ source and destination share single copy,
  - ➜ both virtual pages are mapped to the same frame
- Marked as read-only
- When one copy is modified, a fault occurs
- Handling by kernel involves making a physical copy is made,
  - ➜ VM mapping is changed to refer to the new copy
- Advantage:
  - ➜ efficient way of sharing/passing large amounts of data
- Drawbacks:
  - ➜ expensive for small amounts of data (page table manipulations)
  - ➜ data must be properly aligned

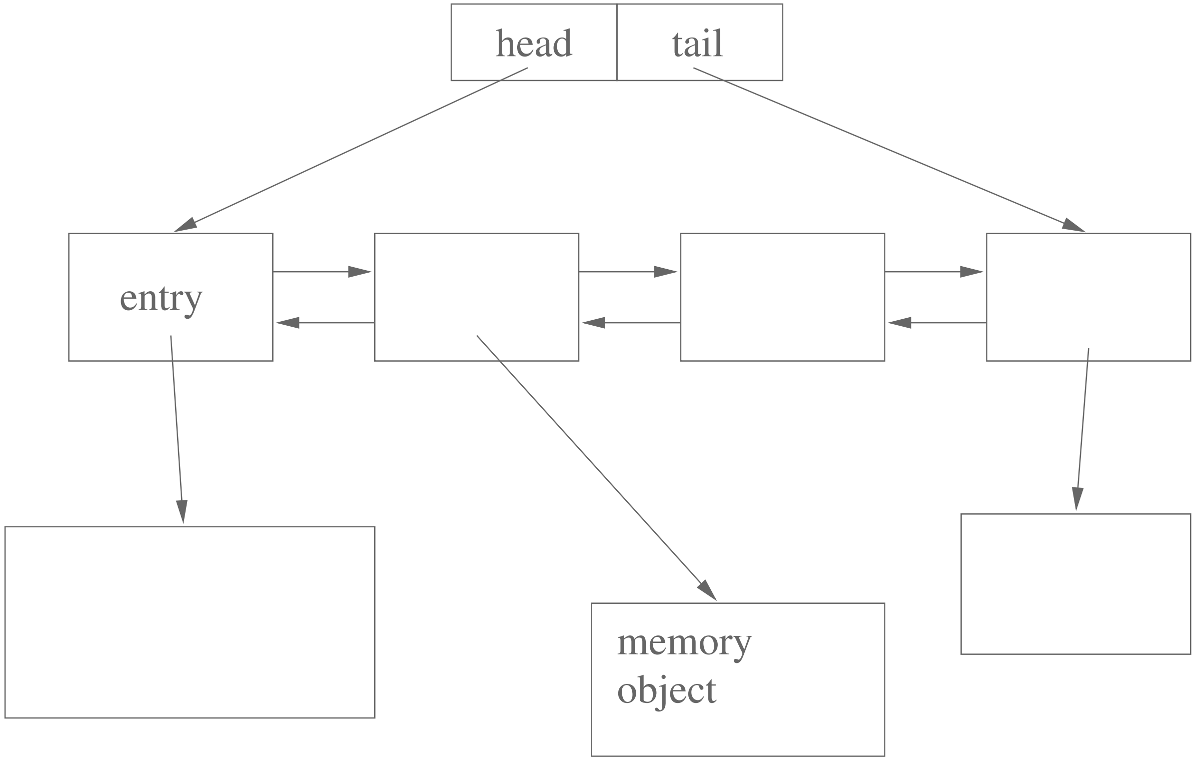
MACH ADDRESS MAPS

- Address spaces represented as *address maps*:



MACH ADDRESS MAPS

- Address spaces represented as *address maps*:



- Any part of AS can be mapped to (part of) a memory object
- Compact representation of *sparse* address spaces

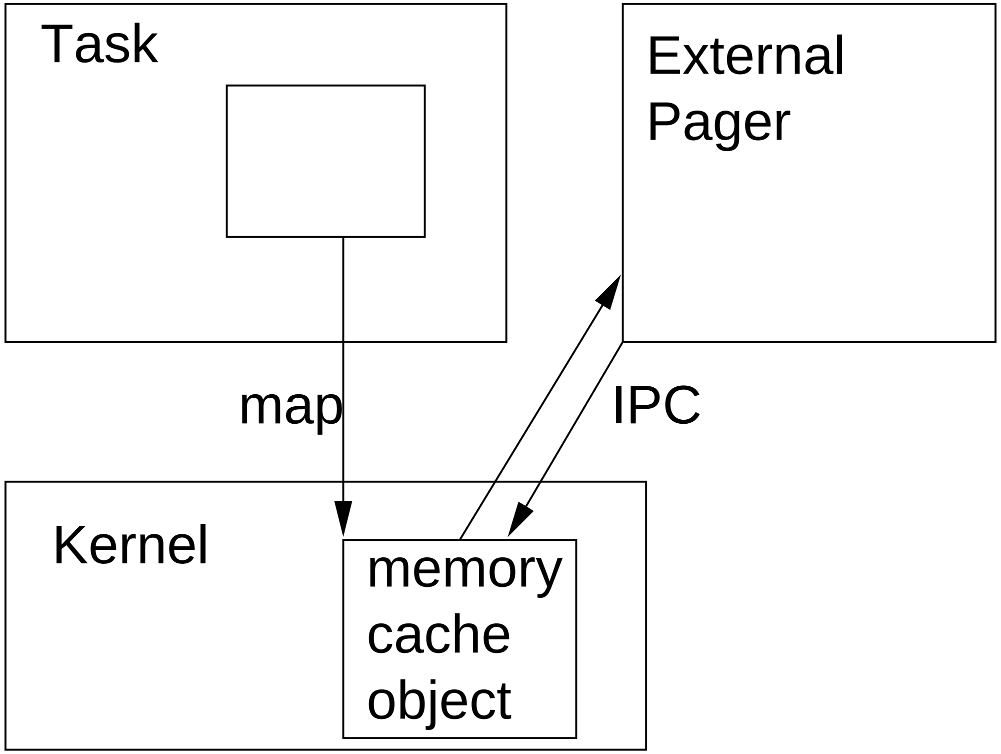
➜ Compare to multi-level page tables?

MEMORY OBJECTS

- Kernel doesn’t support file system
- Memory objects are an abstraction of secondary storage:
  - ⋆ can be mapped into virtual memory
  - ⋆ are cached by the kernel in physical memory
  - ⋆ pager invoked if uncached page is touched
    - ➜ used by file system server to provide data
- Support data sharing
  - ➜ by mapping objects into several address spaces
- Memory is only cache for memory objects

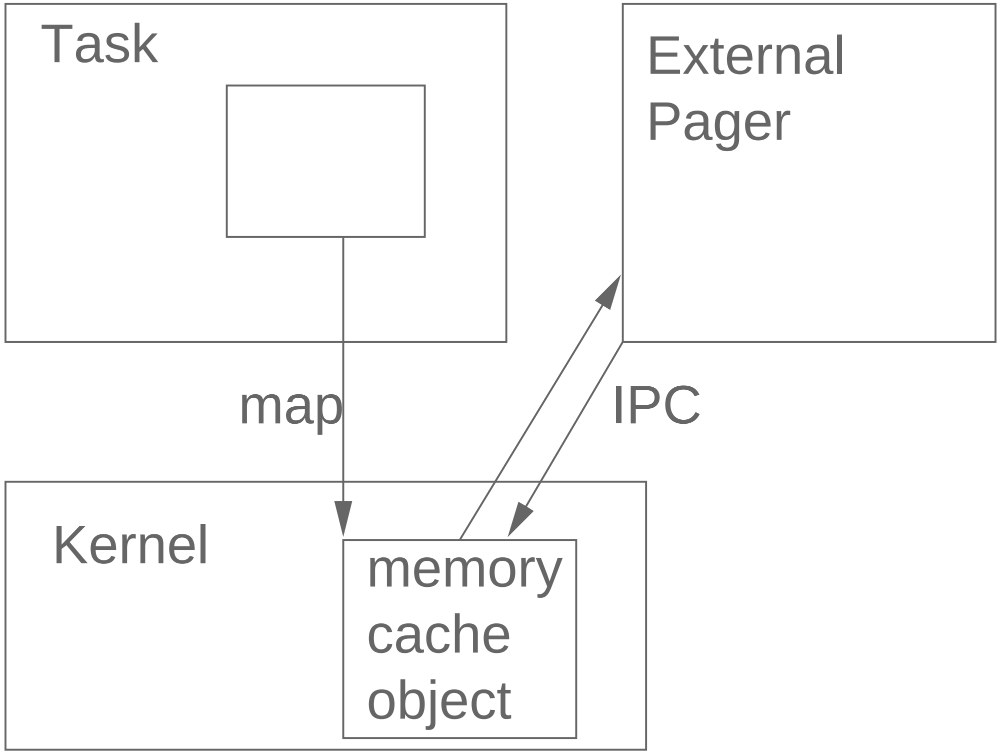
USER-LEVEL PAGE FAULT HANDLERS

- All actual I/O performed by *pager*; can be
  - ➜ default pager (provided by kernel), or
  - ➜ *external* pager, running at user-level.



USER-LEVEL PAGE FAULT HANDLERS

- All actual I/O performed by *pager*; can be
  - ➜ default pager (provided by kernel), or
  - ➜ *external* pager, running at user-level.



- Intrinsic page fault cost: 2 IPCs

HANDLING PAGE FAULTS

① Check protection & locate memory object
  ➜ uses address map

② Check cache, invoke pager if cache miss
  ➜ uses a hashed page table

③ Check copy-on-write
  ➜ perform physical copy if write fault

④ Enter new mapping into H/W page tables

REMOTE COMMUNICATION

- Client A sends message to server B on remote node

  ① A sends message to local *proxy port* for B's receive port

  ② User-level *network message server* receives from proxy port

  ③ NMS converts proxy port into (global) *network port*.

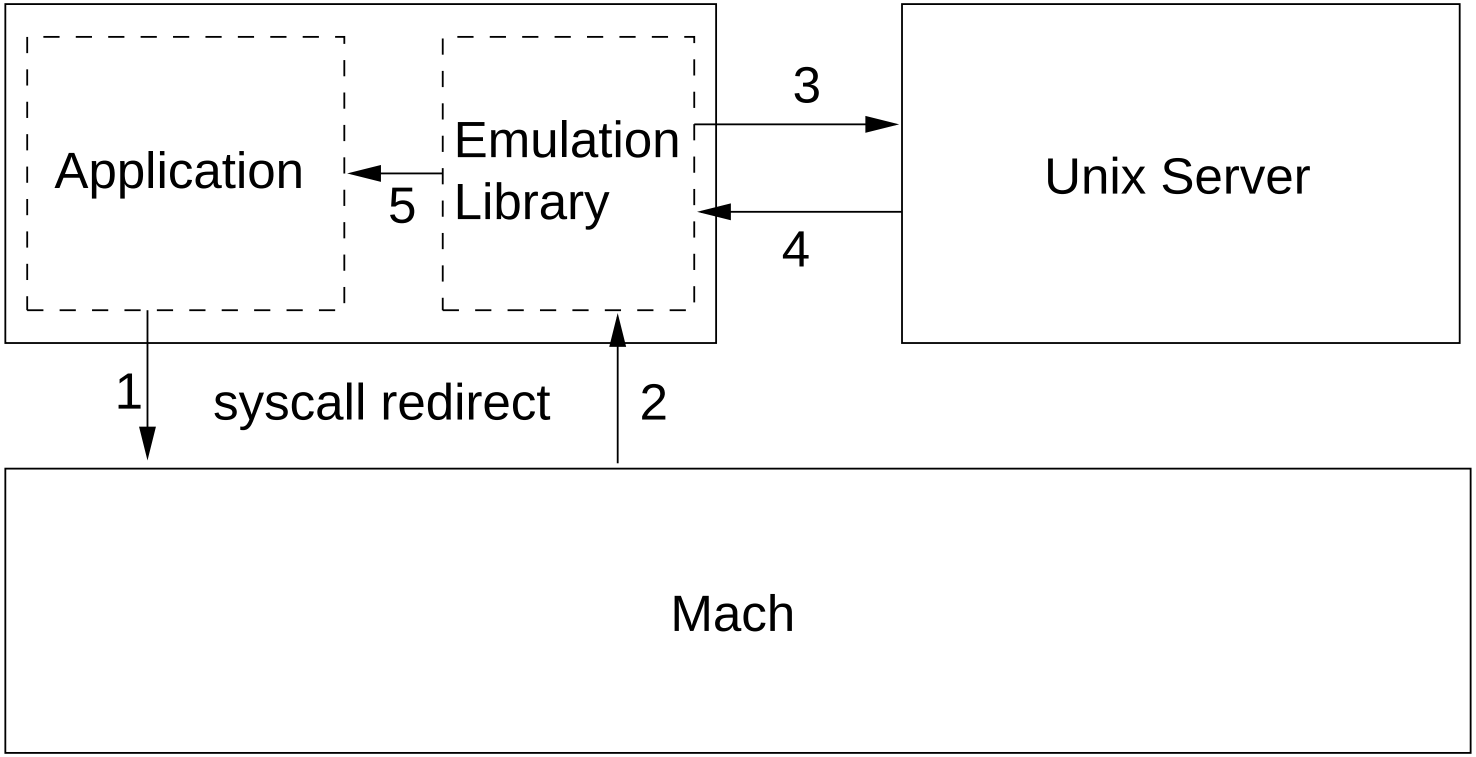
  ④ NMS sends message to NMS on B's node

  ➜ may need conversion (byte order...)

  ⑤ Remote NMS converts network port into local port (B's).

  ⑥ Remote NMS sends message to that port

MACH UNIX EMULATION



- Emulation library in user address space handles IPC
- Invoked by system call redirection (*trampoline mechanism*)
  ➜ supports binary compatibility

MACH = MICROKERNEL?

- Most OS services implemented at user level

➜ using memory objects and external pagers
➜ Provides mechanisms, not policies

- Mostly hardware independent

MACH = MICROKERNEL?

- Most OS services implemented at user level
  - ➜ using memory objects and external pagers
  - ➜ Provides mechanisms, not policies

- Mostly hardware independent

- Big!
  - ➜ 140 system calls
  - ➜ Size: 200k instructions

- Performance poor
  - ➜ tendency to move features into kernel
  - ➜ Darwin (base if MacOS X) has complete BSD kernel inside Mach

MACH = MICROKERNEL?

- Most OS services implemented at user level
  - ➜ using memory objects and external pagers
  - ➜ Provides mechanisms, not policies
- Mostly hardware independent
- Big!
  - ➜ 140 system calls
  - ➜ Size: 200k instructions
- Performance poor
  - ➜ tendency to move features into kernel
  - ➜ Darwin (base if MacOS X) has complete BSD kernel inside Mach

Further information on Mach: [YTR$^+$87, CDK94, Sin97]

OTHER CLIENT-SERVER SYSTEMS

- Lots!

OTHER CLIENT-SERVER SYSTEMS

- Lots!
- Most notable systems:

Amoeba: FU Amsterdam, early 1980's [TM81, TM84, MT86]

Chorus: INRIA (France), from early 1980's [DA92, RAA$^+$90, RAA$^+$92]

➜ Commercialised by *Chorus Systèmes* in 1988
➜ Recently bought by Sun

Windows NT: Microsoft (early 1990's) [Cus93]

➜ Main servers run in kernel mode

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

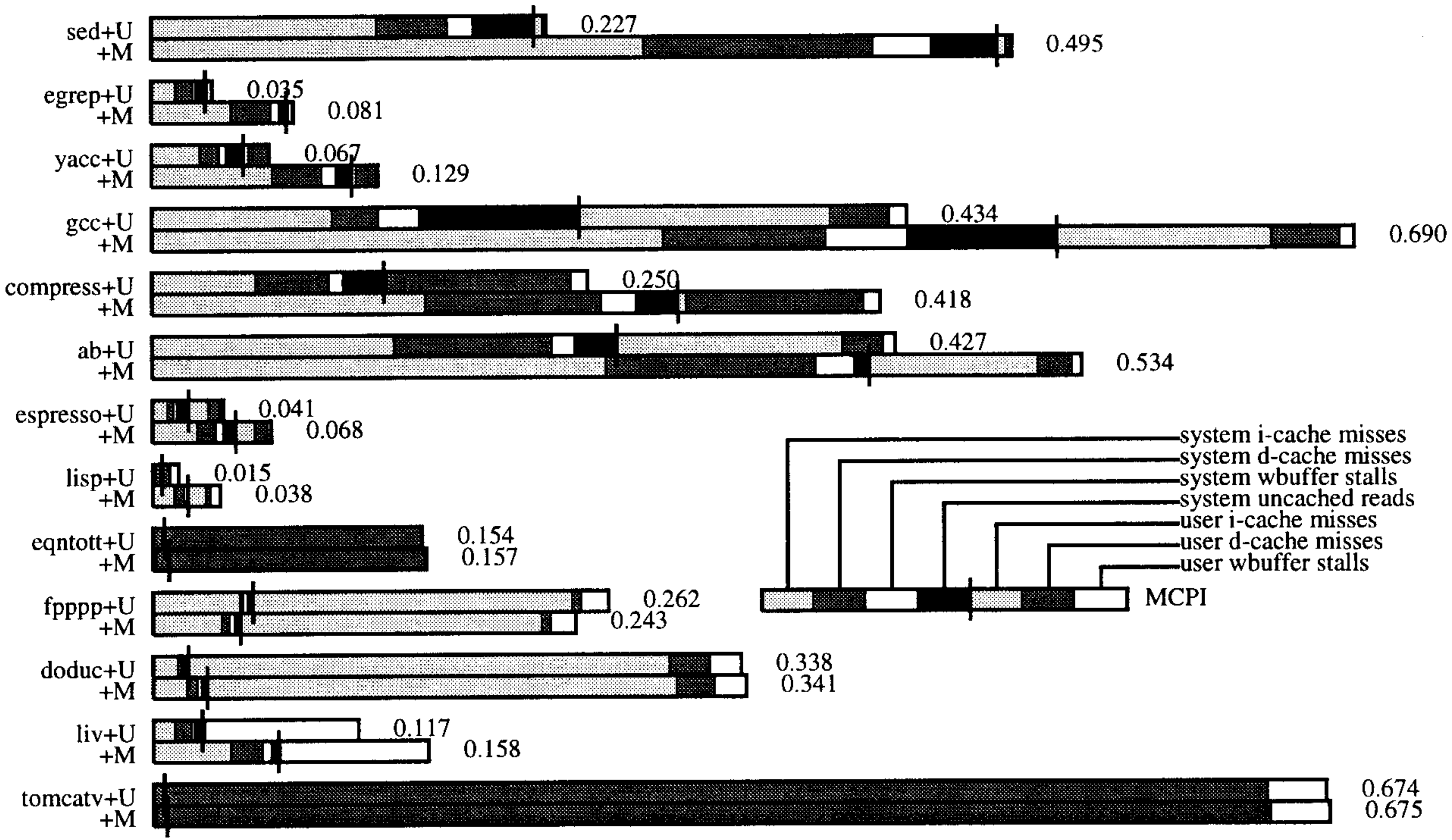
- First generation μ-kernel systems exhibited poor performance when compared to monolithic UNIX implementations

➜ particularly Mach, the best-known example

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 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

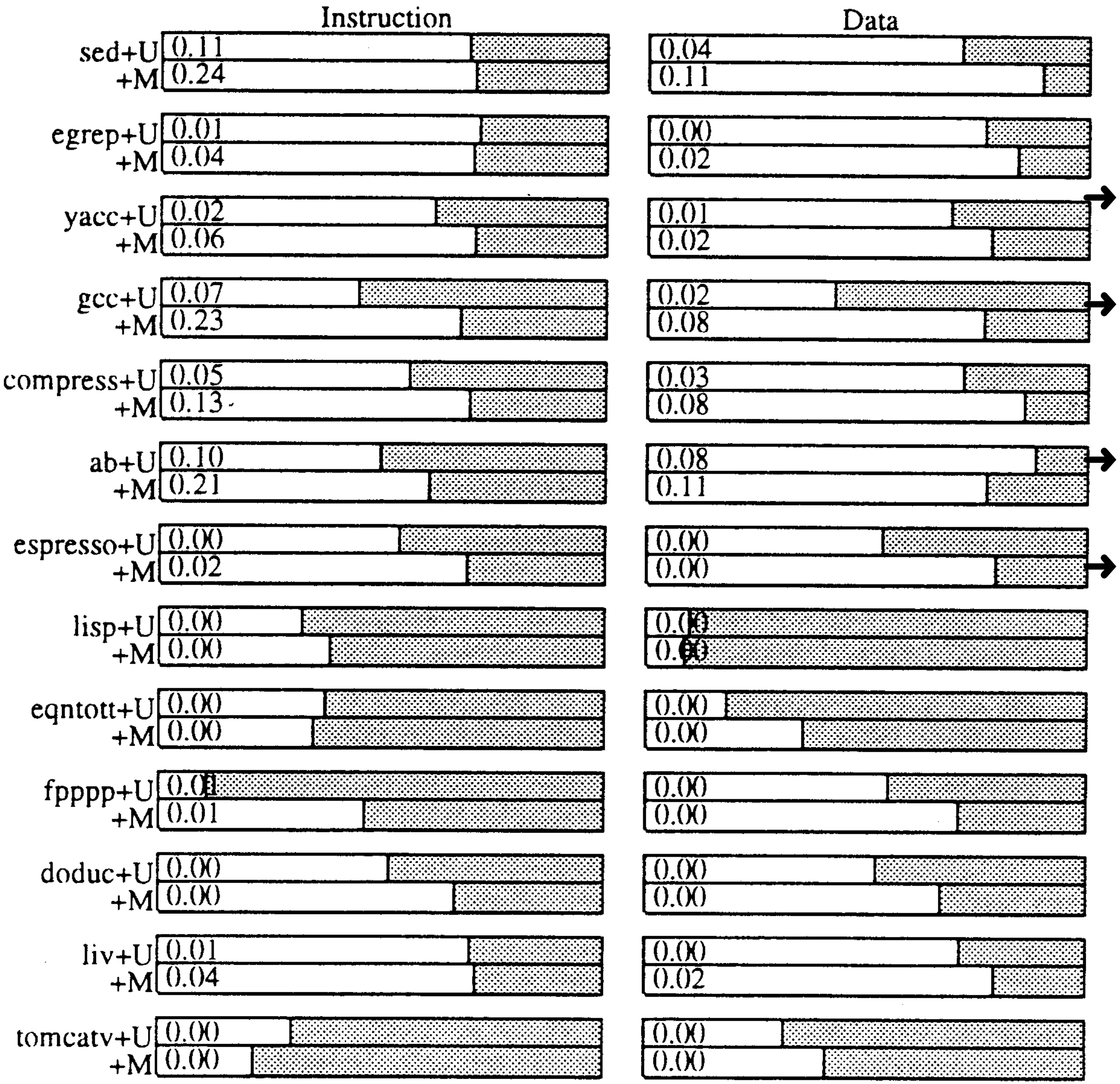
Note: most benchmarks were small, i.e. user code fits in cache

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 asynchronous 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 asynchronous 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 μ-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?

- μ-kernels heavily depend on IPC
- IPC is expensive

SO, WHAT'S WRONG?

- 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?

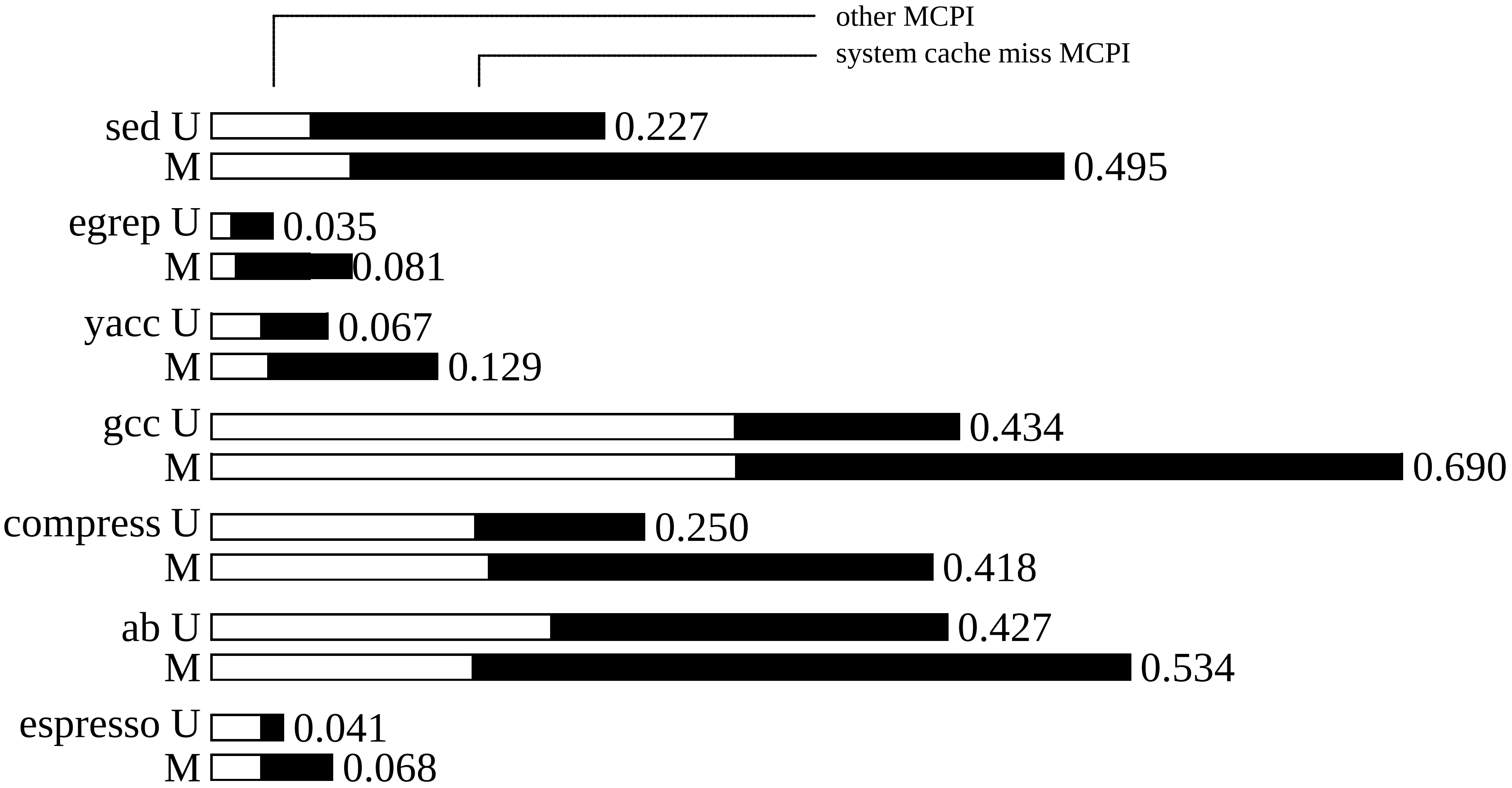
A CRITIQUE OF THE CRITIQUE

Data presented earlier:

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

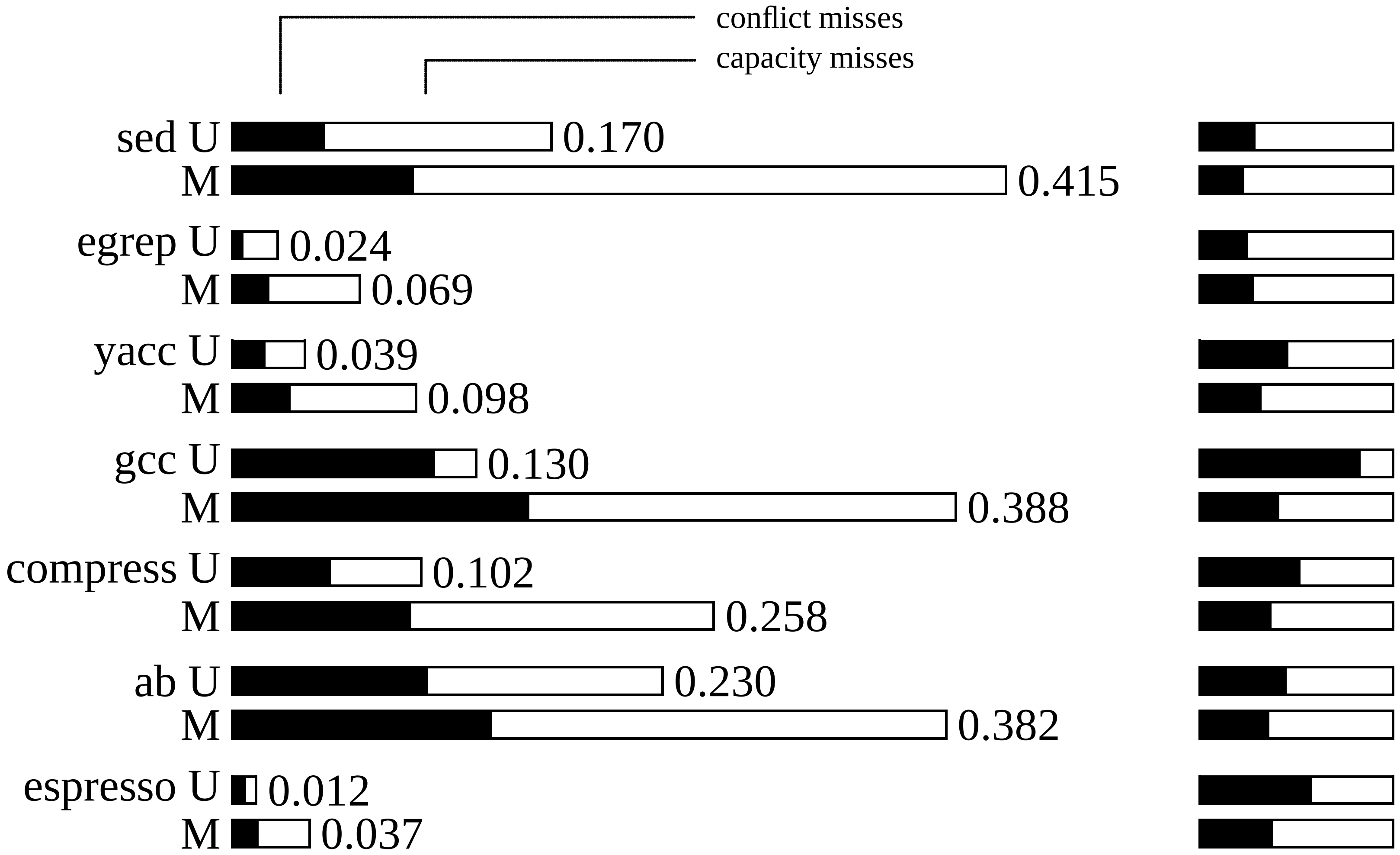
⇒ Cannot trust the conclusions [Lie95]

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)

$\Rightarrow$ Mach μ-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 μ-kernel working set is too big

Can we build μ-kernels which avoid these problems?

REQUIREMENTS FOR μ-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 μ-kernel concept!

μ-KERNEL DESIGN PRINCIPLES [LIE96]

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 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 (LIEDTKE)

- Appropriate system calls to reduce number of kernel invocations
  - ➜ e.g., *reply & receive next*
  - ➜ as many syscall args as possible in registers

L4 IMPLEMENTATION TECHNIQUES (LIEDTKE)

- Appropriate system calls to reduce number of kernel invocations

  ➜ e.g., *reply & receive next*
  ➜ as many syscall args as possible in registers

- Efficient IPC

  ➜ rich message structure
  ➜ value and reference parameters in message
  ➜ copy message only once (i.e. **not** user→kernel→user)

L4 IMPLEMENTATION TECHNIQUES (LIEDTKE)

- Appropriate system calls to reduce number of kernel invocations
  - ➜ e.g., *reply & receive next*
  - ➜ as many syscall args as possible in registers
- Efficient IPC
  - ➜ rich message structure
  - ➜ value and reference parameters in message
  - ➜ copy message only once (i.e. **not** user→kernel→user)
- Fast thread access
  - ➜ Thread UIDs (containing thread ID)
  - ➜ TCBs in (mapped) VM, cache-friendly layout
  - ➜ Separate kernel stack for each thread (for fast interrupt handling)

L4 IMPLEMENTATION TECHNIQUES (LIEDTKE)

- Appropriate system calls to reduce number of kernel invocations
  - ➜ e.g., *reply & receive next*
  - ➜ as many syscall args as possible in registers
- Efficient IPC
  - ➜ rich message structure
  - ➜ value and reference parameters in message
  - ➜ copy message only once (i.e. **not** user→kernel→user)
- Fast thread access
  - ➜ Thread UIDs (containing thread ID)
  - ➜ TCBs in (mapped) VM, cache-friendly layout
  - ➜ Separate kernel stack for each thread (for fast interrupt handling)
- General optimisations
  - ➜ “Hottest” kernel code is shortest
  - ➜ Kernel IPC code on single page, critical data on single page
  - ➜ Many H/W specific optimisations

MICROKERNEL 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

L4KA::PISTACHIO IPC PERFORMANCE

	port/	**C++**		**optimised**	
Architecture	**optimisation**	**intra AS**	**inter AS**	**intra AS**	**inter AS**
Pentium-3	UKa	180	367	113	305
Pentium-4	UKa	385	983	196	416
Itanium 2	UKa/NICTA	508	508	36	36
cross CPU	UKa	7419	7410	N/A	N/A
MIPS64	NICTA/UNSW	276	276	109	109
cross CPU	NICTA/UNSW	3238	3238	690	690
PowerPC-64	NICTA/UNSW	330	518	200‡	200‡
Alpha 21264	NICTA/UNSW	440	642	≈70†	≈70†
ARM/XScale	NICTA/UNSW	340	340	120–140‡	120–140‡
UltraSPARC	NICTA/UNSW			100‡	100‡

† “Version 2” assembler kernel

‡ Guestimate!

CASE IN POINT: L^4LINUX [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^4LINUX [HÄRTIG *et al*. 97]

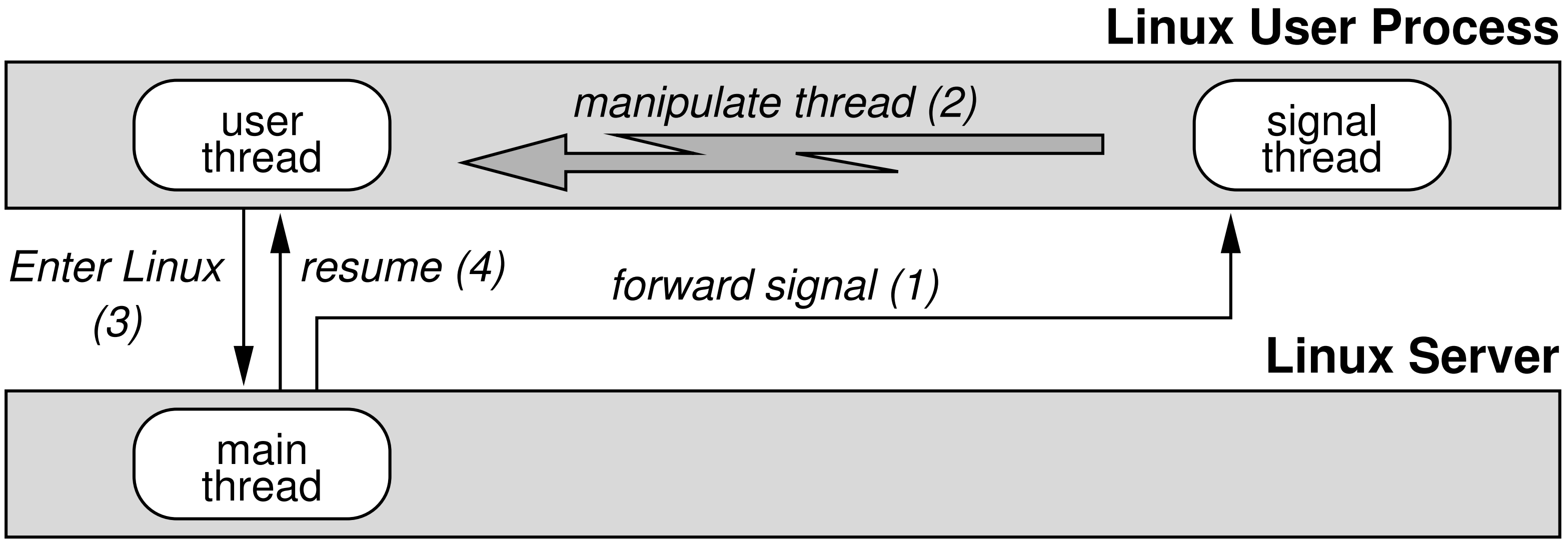
- Port of Linux kernel to L4 (like Mach Unix server)
  - ➜ single-threaded (for simplicity, **not** performance)
  - ➜ is pager of all Linux user processes
  - ➜ maps emulation library and signal-handling code into AS
  - ➜ server AS maps physical memory (& Linux runs within)
  - ➜ copying between user 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^{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

L^4LINUX PERFORMANCE

MICROBENCHMARKS: `GETPID()`

System	*Time [μs]*	*Cycles*
Linux	1.68	223

L^{4}LINUX PERFORMANCE

MICROBENCHMARKS: `GETPID()`

System	*Time [μs]*	*Cycles*
Linux	1.68	223
L^{4}Linux	3.95	526
L^{4}Linux (trampoline)	5.66	753

L^{4}LINUX PERFORMANCE

MICROBENCHMARKS: `GETPID()`

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

L^{4}LINUX PERFORMANCE

MICROBENCHMARKS: `GETPID()`

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

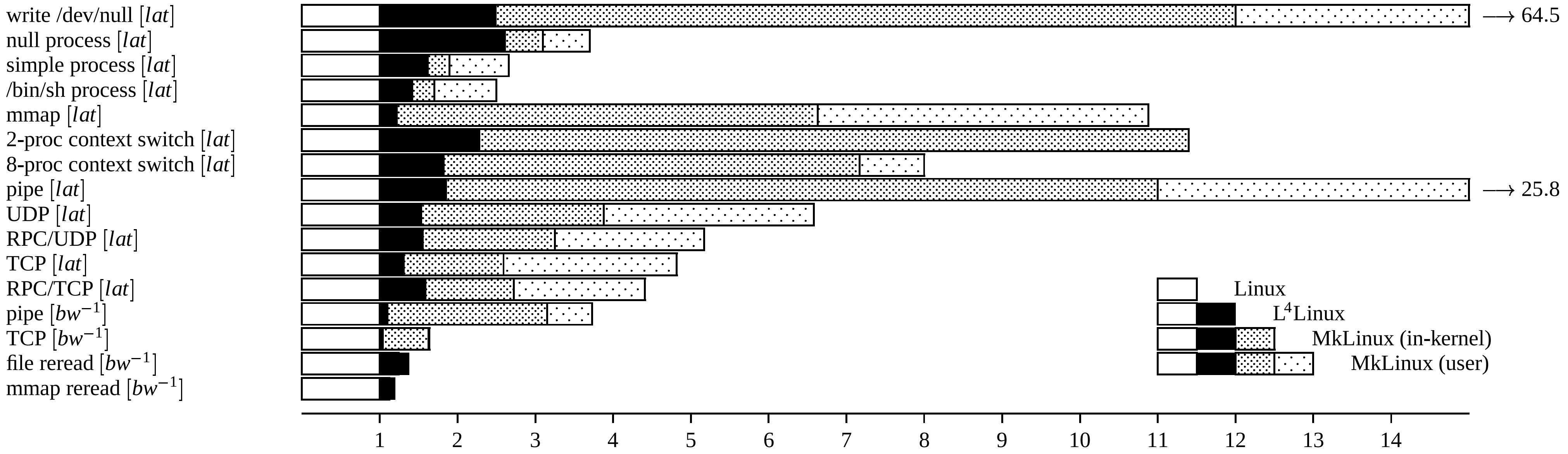
CYCLE BREAKDOWN:

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 (133MHz Pentium)

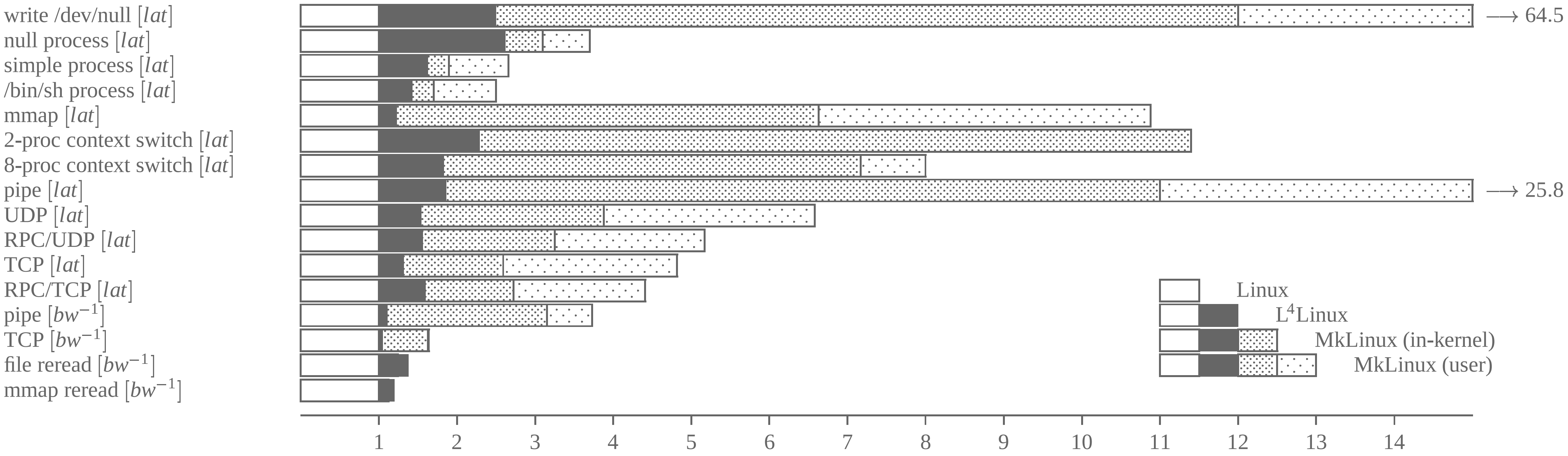
L^{4}LINUX PERFORMANCE

MICROBENCHMARKS: LMBENCH

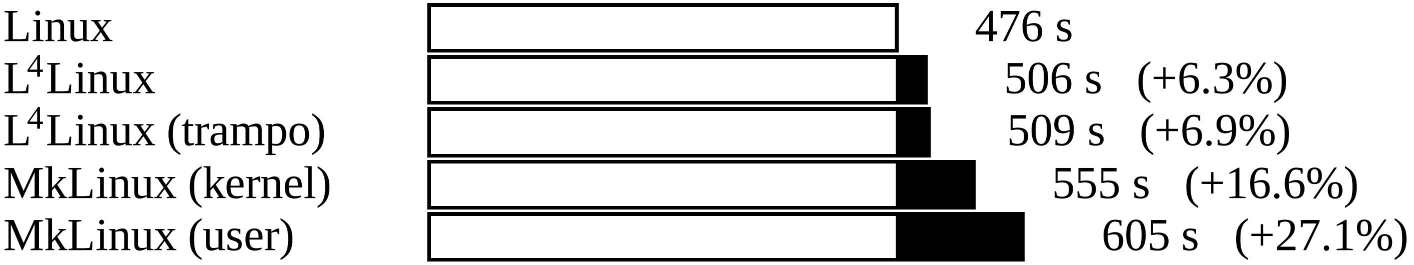


L^{4}LINUX PERFORMANCE

MICROBENCHMARKS: LMBENCH



MACROBENCHMARKS: KERNEL COMPILE



CONCLUSIONS

- Mach sux $\not\Rightarrow$ microkernels suck

CONCLUSIONS

- Mach sux $\not\Rightarrow$ microkernels suck

- L4 shows that performance *might* be deliverable

➜ L^4Linux gets close to monolithic kernel performance
➜ need real multi-server system to evaluate μ-kernel potential

CONCLUSIONS

- Mach sux $\not\Rightarrow$ microkernels suck
- L4 shows that performance *might* be deliverable
  - ➜ L^4Linux 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's still an uphill battle!

REFERENCES

[Ber92] Brian N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *USENIX WS Microkernels & other Kernel Arch.*, pages 205–211, Seattle, WA, USA, Apr 1992.

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming operating system. *CACM*, 13:238–250, 1970.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *14th SOSP*, pages 120–133, Asheville, NC, USA, Dec 1993.

[CDK94] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2nd edition, 1994.

[Cus93] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

[DA92] Randall W. Dean and Francois Armand. Data movement in kernelized systems. In *USENIX WS Microkernels & other Kernel Arch.*, pages 243–261, Seattle, WA, USA, Apr 1992.

[Dij68] Edsger W. Dijkstra. The structure of the "THE" multiprogramming system. *CACM*, 11:341–346, 1968.

[FR86] Robert Fitzgerald and Richard Rashid. The integration of virtual memory management and interprocess communication in Accent. *Trans. Comp. Syst.*, 4:147–177, 1986.

[HHL$^+$97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *16th SOSP*, pages 66–77, St. Malo, France, Oct 1997.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[Lie96] Jochen Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The design of a capability-based distributed operating system. *The Comp. J.*, 29:289–299, 1986.

[Ous90] J.K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *1990 Summer USENIX Techn. Conf.*, pages 247–56, Jun 1990.

[RAA$^+$90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the CHORUS distributed operating system. Technical report CS/TR-90-25, Chorus systeèmes, Montigny-le-Bretonneux (France), Apr 1990.

[RAA$^+$92] M. Rozier, V. Abrossimov, F. Armand, L. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *USENIX WS Microkernels & other Kernel Arch.*, pages 39–69, Seattle, WA, USA, Apr 1992.

[Ras88] Richard F. Rashid. From RIG to Accent to Mach: The evolution of a network operating system. In Bernardo A. Huberman, editor, *The Ecology of Computation*, Studies in Computer Science and Artifical Intelligence, pages 207–230. North-Holland, Amsterdam, 1988.

[RTY$^+$88] Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *Trans. Computers*, C-37:896–908, 1988.

[Sin97] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design*. Comp. Soc. Press, 1997.

[SJW+02] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the Linux kernel. *IEE Proc.: Softw.*, 149:18–23, 2002.

[TM81] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *Operat. Syst. Rev.*, 15(3):51–64, 1981.

[TM84] Andrew S. Tanenbaum and Sape Mullender. The design of a capability-based distributed operating system. Technical Report IR-88, Vrije Universiteit, Nov 1984.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *CACM*, 17:337–345, 1974.

[YTR⁺87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *11th SOSP*, pages 63–76, 1987.