Slide 1

The Performance of μ-Kernel-Based Systems

Hermann Härtig
Michael Hohmuth
Sebastian Schönberg
Jean Wolter

Dresden University of Technology
Department of Computer Science

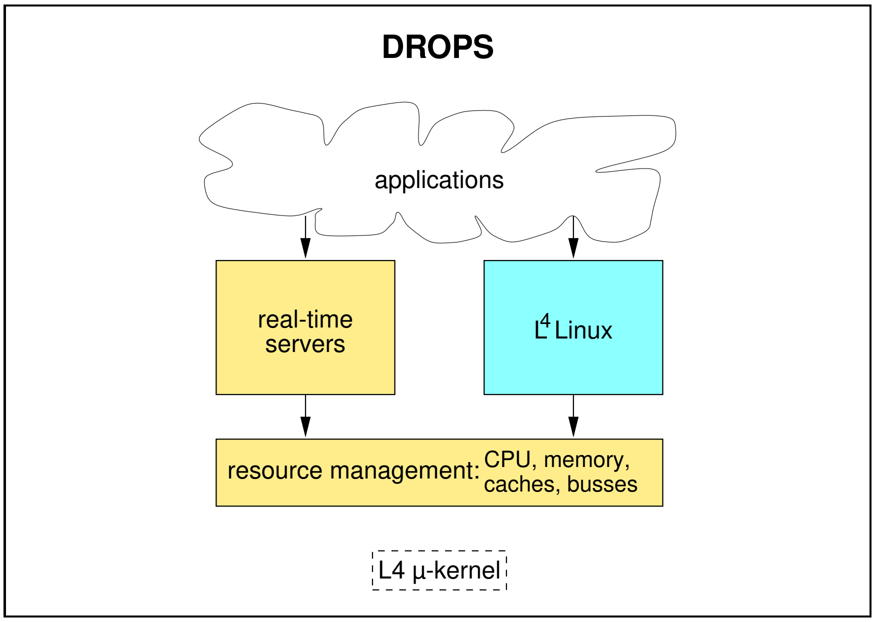
Jochen Liedtke

IBM T. J. Watson Research Center
Hawthorne, NY

This is a joint work with

- Sebastian Schönberg and Jean Wolter,
- and our PhD supervisor Hermann Härtig.
- Jochen Liedtke from IBM Watson Research Center,
- (I am Michael Hohmuth.)

Slide 2



First some notes on our project's background:

- our group in Dresden focuses on µK-based real-time systems, for example for continuous media presentation; DROPS stands for "Dresden Real-Time Operating System"
- vision: RT components get all the resources and processing power they need, and the rest can be burned for non-RT tasks
- this is enforced by a system-wide resource management for all system resources: not only CPU time slices and main memory, but also for caches and I/O buses, for example
- we needed platform to do development and other "real" work on
- that's why, we ported Linux to the L4 µK
  - port promises fast results
  - repetition of earlier similar experiments, but more straight-forward and with fewer compromises

Main question answered in our paper is: What performance can we expect from an L4-based Unix system?

In the following, I'll mainly concentrate on L^4Linux.

- I'll cover a few design aspects of our port of the monolithic Linux kernel to the L4 µK
- then we'll take a look at the performance of the resulting µK-based Linux system.

Objectives for L^4Linux

Slide 3

- best possible performance
- constraints:
  - binary compatible with Linux/x86
  - minimal modifications

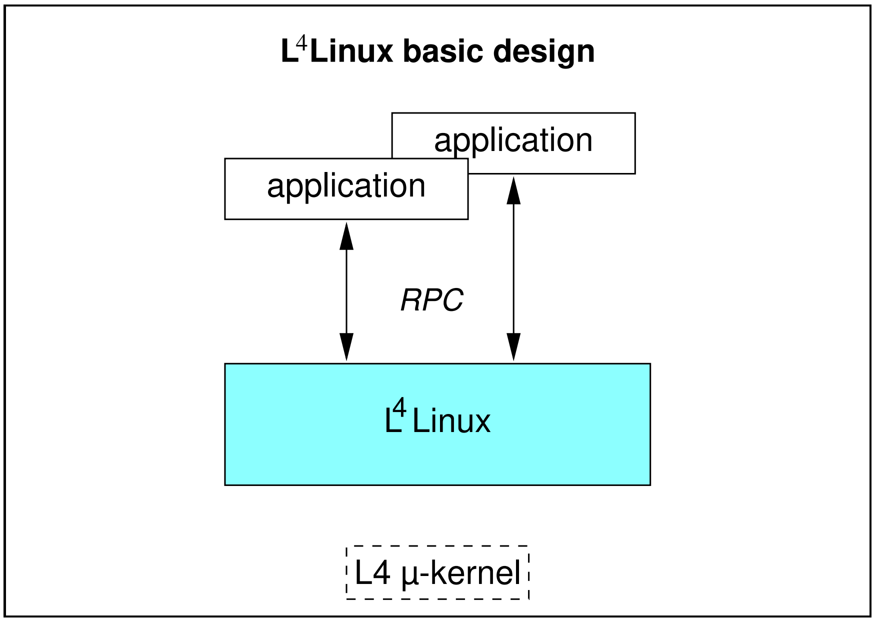
- so the main objective is to aim for the best possible performance, but with the following constraints:
- L4 µK runs on Intel 486 and Pentium processors, therefore we aimed at binary compatibility with Linux for these processors.

  our L^4Linux server should be usable with every standard Linux installation
- in order to save development time, we wanted to make only minimal modifications to Linux; we didn't intend to optimize it for the underlying L4 architecture in the way for instance the BSD single servers have been optimized for Mach

  this should also give us a fair understanding of the penalty of introducing a µK to an existing system.

The simplest possible design matching the last criterion. . .

Slide 4



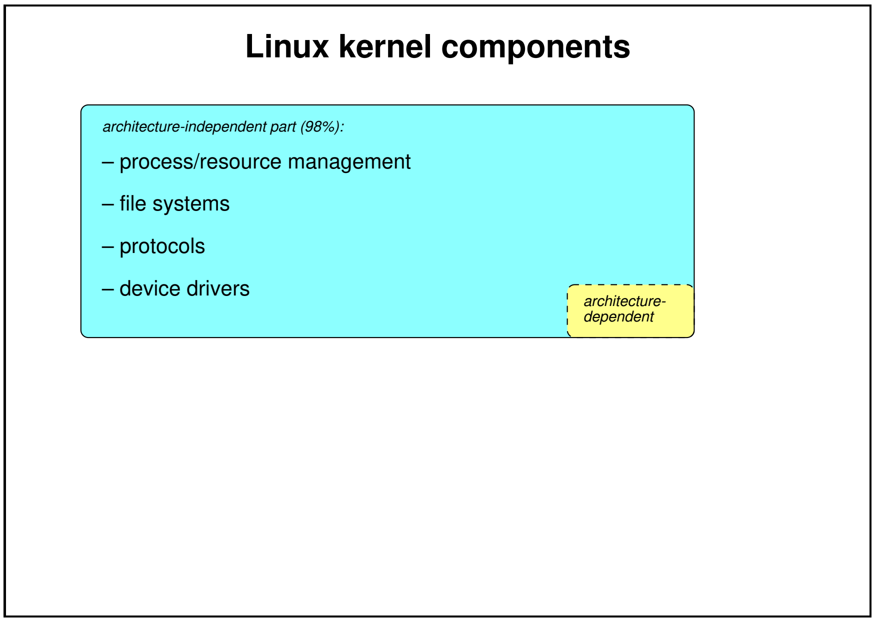
The simplest possible design matching the last criterion (minimal modification) is:

- a single server design;
- Linux server and
- user applications run in separate L4 tasks
- which communicate using µK-IPC-based remote procedure calls.

this is a trivial and very straight-forward design

How can we adapt the monolithic Linux kernel to a microkernel? Let's look the the Linux kernel's internals.

Slide 5



Linux consists of two parts:

1. architecture-independent part — makes up about 98% of all kernel sources

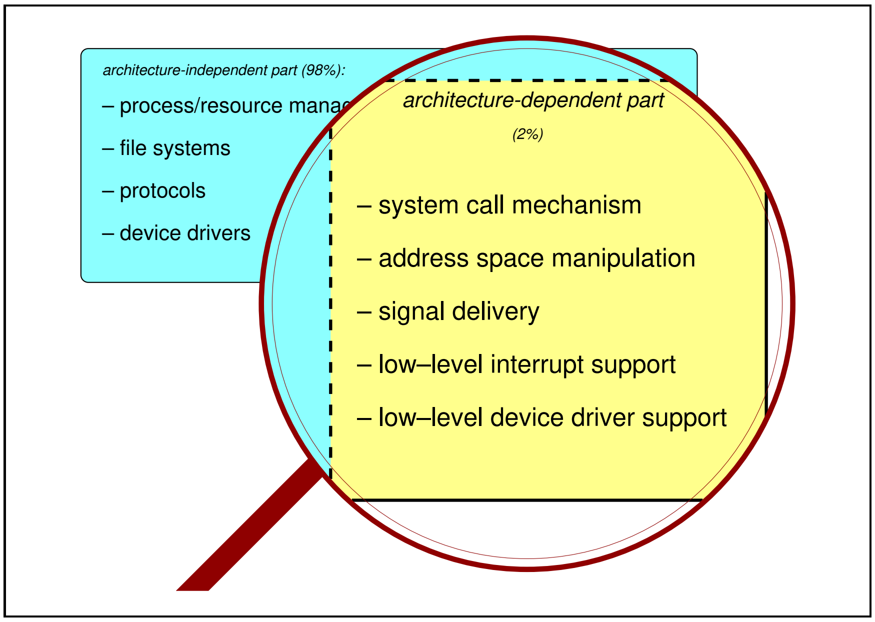
   contains most of the traditional Unix functionality

2. a small architecture-specific part which needs to be adopted when porting the Linux kernel to a new architecture

   encapsulates underlying architecture

We didn't want to modify or redesign the arch.-independent part of Linux. Instead, we wanted to constrain our changes to the arch.-specific part — lets zoom into this part.

Slide 6

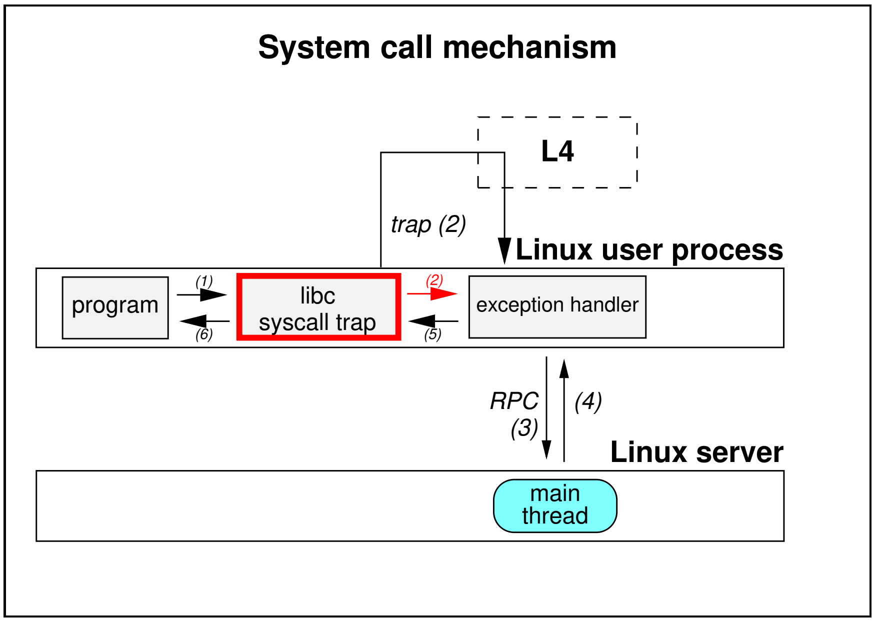


- system call mechanism
- address space manipulation, signal delivery, and low-level interrupt handling — we'll come to that in a minute
- low-level device driver support, e.g. for DMA — we copied this part unmodified from Linux/x86 because our Linux server runs on the same hardware architecture

to minimize our changes to Linux, we constrained changes to this small architecture-dependent part of Linux

now lets take a closer look at the remaining parts: system call mechanism, address space manipulation, signal delivery, and low-level interrupt handling

Slide 7



To initiate a system call: Linux:

- a user program calls a system call stub in the C library which then executes a system call trap.
- the trap is directly caught by the Linux kernel and handled appropriately.

Under L^{4}Linux, things aren't so easy, because the Linux kernel is running in a different task, and exceptions cannot be propagated automatically between tasks.

What happens is this:

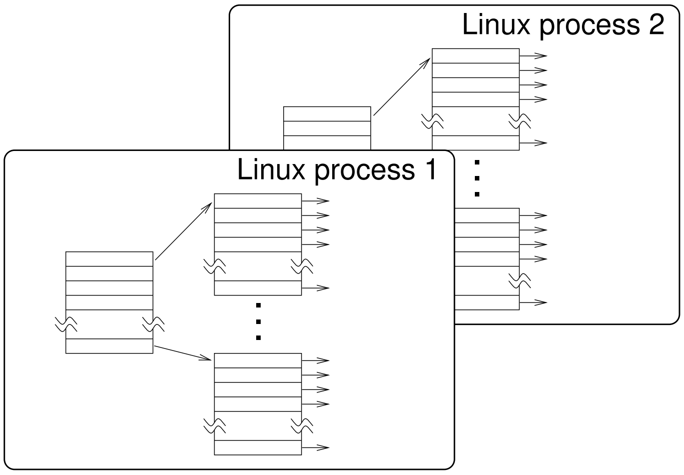
- When a system call trap exception is raised, the L4 kernel reflects it via a trampoline mechanism to a predefined handler in the raising thread. (A similar trampoline mechanism can be found in the Mach µK.)
- under L^{4}Linux, that's an exception handler mapped into each user task's address space

  apart from converting exceptions into RPCs for binary compatibility, the small exception handler library doesn't do any sys-call emulation
- handler transforms the exception into an IPC message sent to the Linux kernel, and then waits for a reply message
- then, the system call stub in the C library is resumed.

The trampoline mechanism has a certain cost: there's an additional switch to kernel mode, and it uses a non-optimized path within the L4 µK; fortunately it's possible to circumvent this cost for all applications linked dynamically against Libc simply by replacing it with a different version that doesn't do system call traps but directly invokes the respective handler.

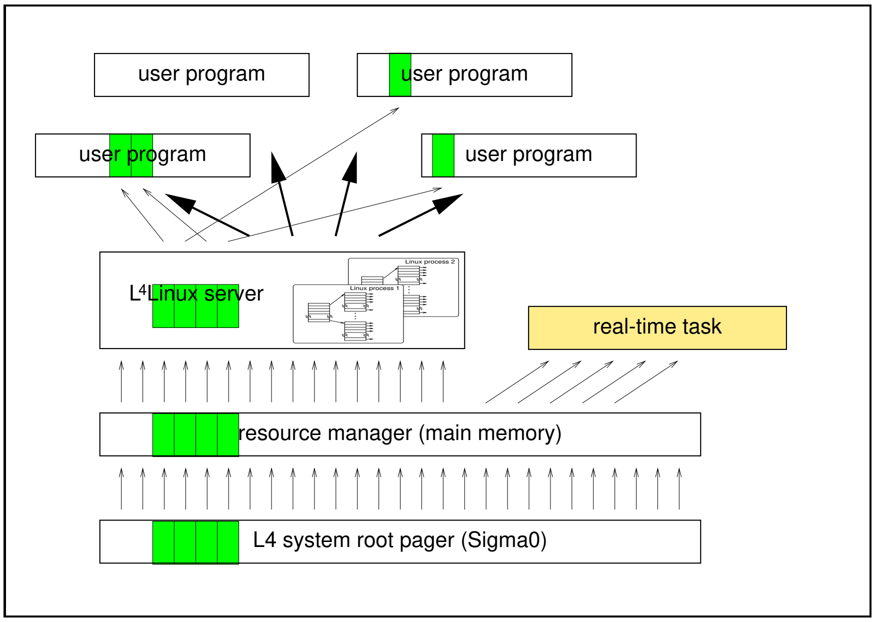
Slide 8

Virtual address space management in Linux



- Linux uses logical pages table to manages the user processes' address spaces
- the architecture adaption then maps operations on these page tables to address space manipulations using the underlying architecture
- Under monolithic Linux for the x86 architecture, this adaption layer hasn't much to do: it simply uses the logical page table as the CPU's physical page table
- because of (obvious) security reasons, the real page tables of the system are kept within the L4 µK, and these must be manipulated using L4's mapping primitives
- however, the logical page tables are still kept for bookkeeping within the L^4Linux server

Slide 9

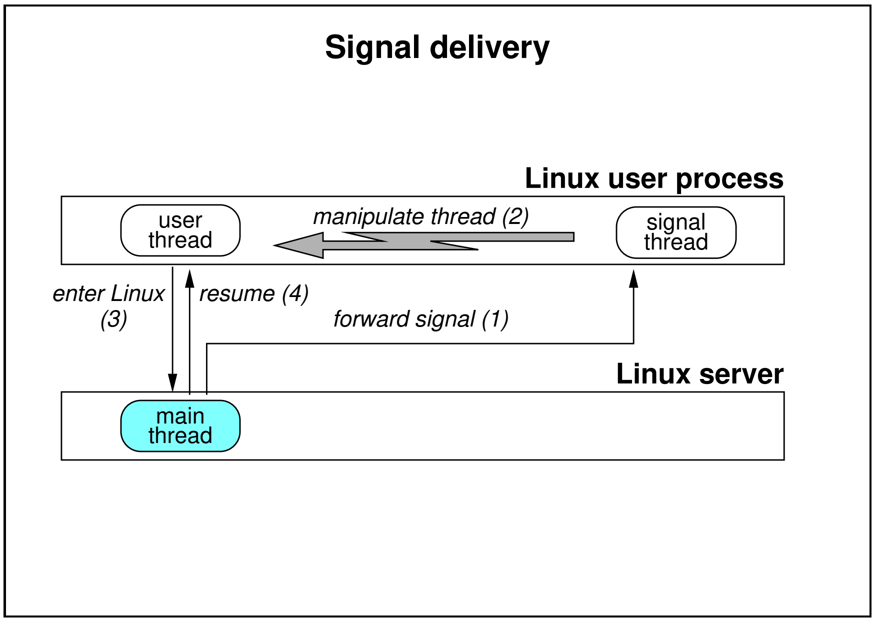


- the L4 tasks' virtual address spaces are constructed recursively between tasks
- Linux server maps physical memory one-to-one from its pager, normally the L4 system pager called sigma-zero
- Linux server manages virtual address spaces of Linux clients and acts as an external memory manager (pager) for them
- virtual address spaces are managed in the Linux server using a logical three-level page table structure

and BTW, we could construct a second sub-hierarchy on the same machine, for instance for a real-time network driver and its clients

this is what we eventually want to do for our real-time project

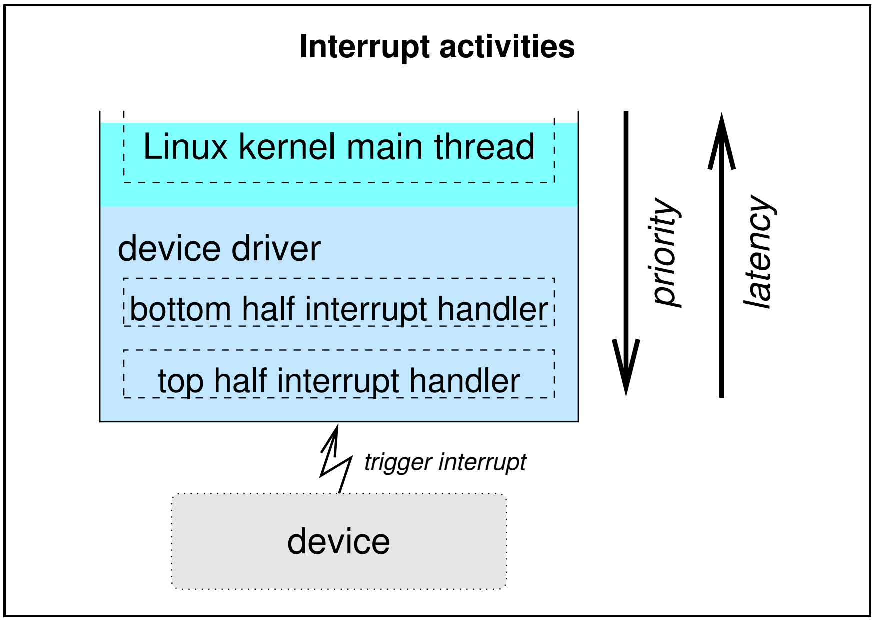
Slide 10



next architecture-specific detail to look at: signal delivery

- suppose someone (else) has sent this user thread a signal which must be delivered and handled by a signal handler, but the thread is busy executing independently.
- we decided to handle signal delivery completely within the L^4Linux server

  in order to be able to deliver the signal, the user thread must enter the kernel so that its stack can be manipulated safely.
- under L4, a task can't directly manipulate another task's threads, so
  - signals are first forwarded to a special signal thread running within the user task.
  - this thread then simulates an exception within the user thread and forces it to enter the kernel where it can then be manipulated.



Slide 11

Finally, lets take a look at the adaption of the low-level interrupt interface.

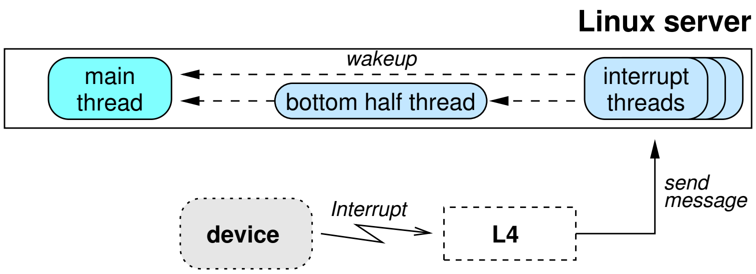
The Linux device driver model knows about two interrupt handler priorities: top half and bottom half. Top half handlers are for low-latency handling of device interrupts, and more work-intensive tasks should be left to bottom half handlers.

These handlers are independent activities and can preempt lower-prioritised activities.

How are these semantics preserved in L^4Linux?

Slide 12

Interrupts in L^{4}Linux



In L4, hardware interrupts can be attached to threads and handled on user level.

When an interrupt occurs, the interrupt is converted to an IPC message (XXX latency???) and delivered to the respective thread.

In L^{4}Linux, there is a thread for each hardware interrupt. These interrupt threads directly execute the low-latency top-half handlers.

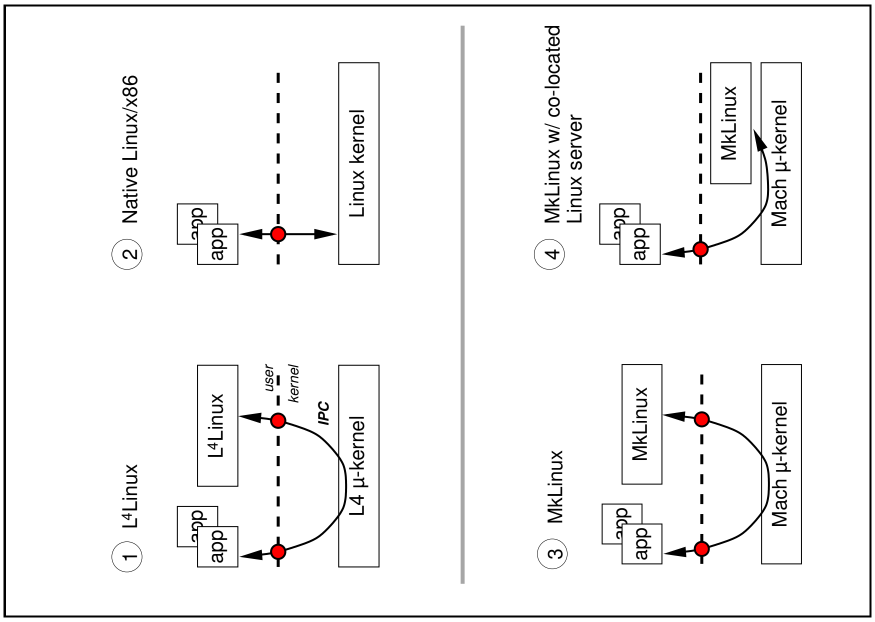
bottom half handlers are executed in a separate thread.

this thread has a lower priority to ensure the Linux preemption semantics.

should any of the interrupt threads detect that a lower-prioritised activity should be started, the respective thread is woken up using an IPC message.

Now let's look into the performance of L^{4}Linux.

Slide 13



For performance evaluation, we compared L^4Linux to three different systems:

2. monolithic Linux, because we aimed at a performance similar to the native version
3. MkLinux, a Linux server running on OSF Mach 3

   It has an architecture similar to L^4Linux in that the Linux server runs in user space, so there are two user/kernel boundary crossings for each IPC.
4. For MkLinux, there is an optimized variant where the Linux server has been co-located into the µ-kernel's address space, thereby saving one user-kernel boundary crossing.

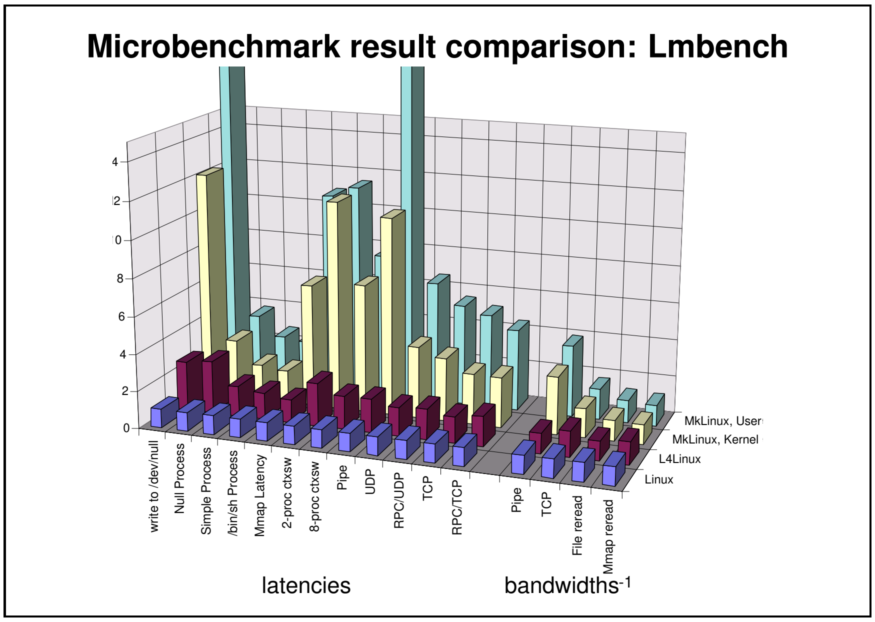
Slide 14

All measurements use the same Linux environment on one single machine:

- a 133 MHz Pentium PC
- based on an ASUS P55TP4N motherboard using Intel's 430FX chip set,
- 256 KB pipeline-burst second-level cache
- 64 MB of 60 ns Fast Page Mode RAM
- Quantum Empire SCSI disk

All measurements use the same Linux environment on the same machine. The machine is a fairly standard x86-based PC.

Slide 15



for a complete analysis of the penalties involved when using a microkernel, we used the Lmbench benchmark suite from Larry McVoy and Carl Staelin.

on the left side are system latencies, and on the right side are inverted bandwidths.

this diagram shows the results normalized to native Linux/x86. they are presented as slowdowns: shorter bar means better result.

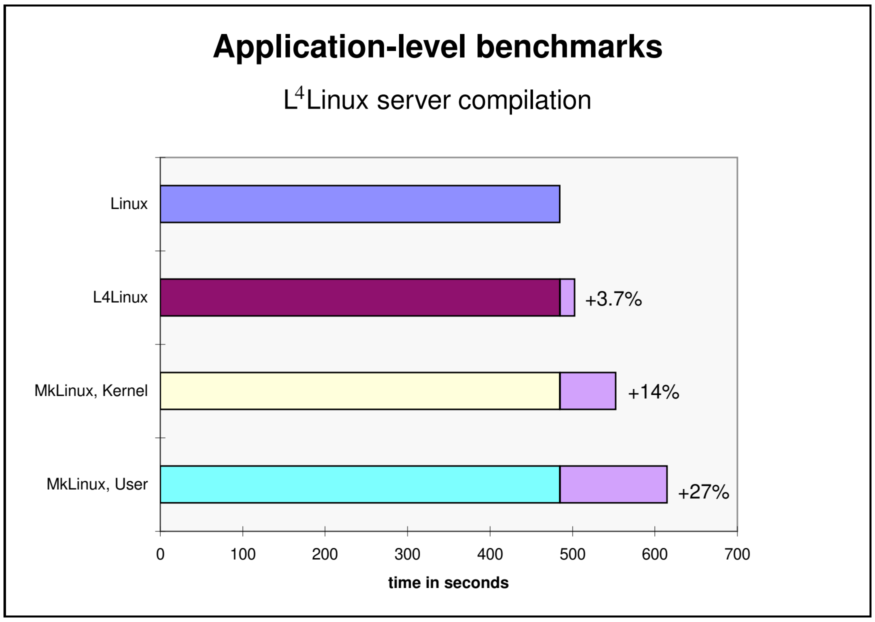
(XXX evtl. Bild mit Absolutzahlen für Linux/x86 im jeweils ersten Kästchen?)

interpretation:

1. the penalty for using L4 is quite high, and in some instances we haven't yet fully understood why, but it is a lot lower than the penalty for using Mach.

2. for many benchmarks, the effect of using L4 versus Mach is larger than the effect of using a co-located Mach versus a user-mode server

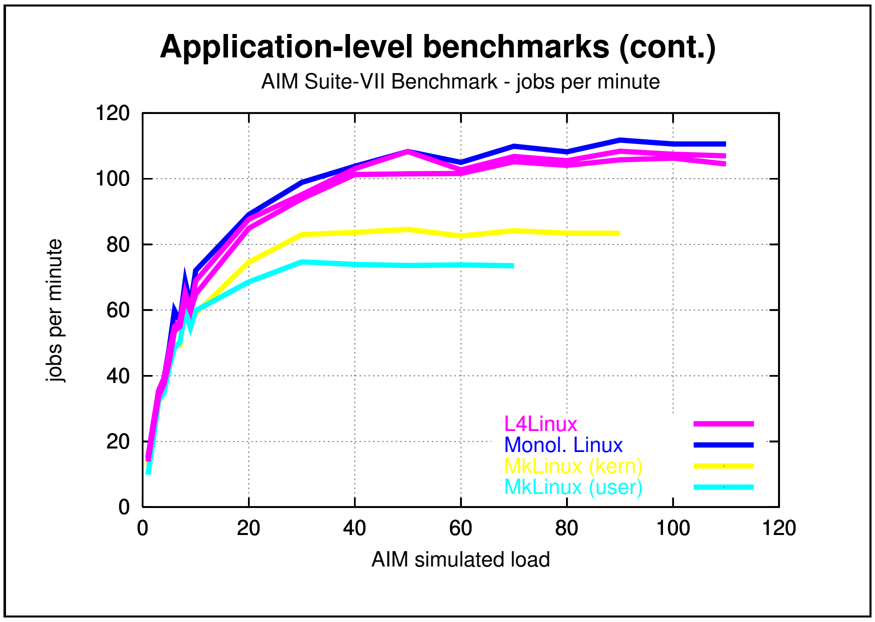
Slide 16



For an intuitive real-world performance evaluation we compiled our L^{4}Linux server using the measurement systems.

- At the first glance, these results corroborate the microbenchmark results, namely that the Mach penalty is much higher than the L4 penalty, and it gives us an idea of the effect of co-location.
- L^{4}Linux 3.7% slower than Linux,
- MkLinux (kernel) 14% slower,
- MkLinux (user) 27% slower

Slide 17



For a more detailed analysis, we used the AIM Multiuser Benchmark which simulates a configurable multiuser load on a system.

This diagram shows the number of AIM multiuser jobs that can be completed per minute.

- The graphs for Mach end earlier because AIM detected a lower maximum load, that is, after that load the time per job starts increasing disproportionally.
- Averaged over all loads, L^4Linux was 7% (XXX correct?) slower than native Linux. (paper: 8.3% avg, 6.8% max)
- MkLinux with a co-located server has a penalty of about 21% compared with native Linux (paper: 29%),
- and MkLinux with a user-mode server is 29% slower (paper: 49%).

(BTW, this diagram looks a little bit different from the one in the proceedings because we had to replace a hard disk drive in the measurement machine — it couldn't quite stand the sustained load.)

These numbers are similar to those published in the proceedings. After we finished the camera-ready copy of our paper, we started analyzing the exact penalties involved with L^4Linux.

- we quickly found the penalty isn't just the IPC overhead.
- a very close look revealed that many cycles were burned in the L4 µK's idler due to a missing wakeup in our low-level interrupt handler

this was of course fixed, which unfortunately or rather fortunately rendered the numbers I just presented out-of-date. Here's a new diagram:

- with this change, we could push the penalty to 2.2%.
- we also verified our assumption that the large extra memory the L4 µK reserves for its own purposes (on a 64 MB machine, that's about 11 MB) contributes to the L^4Linux penalty.
  - for that we run the monolithic Linux kernel with 11 MB less memory,
  - and compared to that result, L^4Linux is only about 1.7% slower.

Slide 18

Specialization

e. g. application-level pipes

system	**latency**	**bandwidth**
Linux	29 µs	41 MB/s
L^4Linux	46 µs	40 MB/s
L^4Linux, app-level Unix pipe	22 µs	48–70 MB/s
L^4Linux, app-level restricted pipe	5 µs	65–105 MB/s

however, nobody would use a µK-based Unix system even if the µK cost only about 3% if there weren't any benefits with such an architecture.

in our paper, we demonstrated two kinds of benefits

1. specialization: we showed that the pipe performance can be increased drastically when using a user-level pipe implementation using µK facilities.

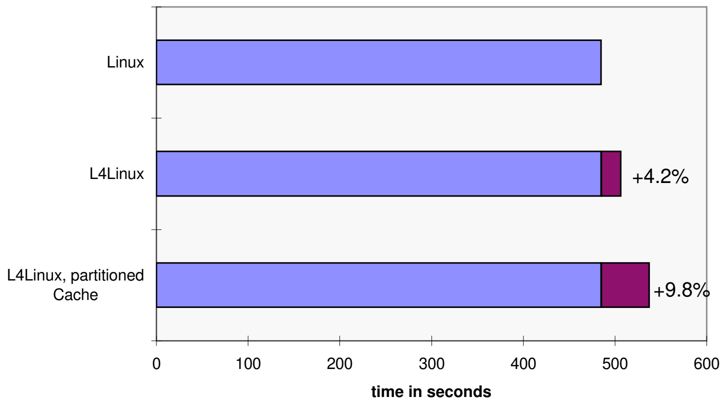
we also want to use the µK facilities for our real-time system project, so an important second benefit is the possibility to add system extensions independent from L^4Linux. . .

Slide 19

Extensibility

e. g. cache partitioning using global main memory manager

L^4Linux server compilation



2. extensibility: we showed that using a global memory manager, the 2nd-level cache can be partitioned so that the worst-case execution time of real-time tasks can be bound to lower values even in the presence of a Unix system running on the same machine.

   - the technique for cache partitioning is described in detail in our RTAS ’97 paper which also contains measurements of the impact for real-time tasks
   - for this presentation, we measured the impact for a timesharing system co-existing on the same machine.
   - diagram shows slowdowns for the compilation benchmark

Slide 20

Conclusion

- New µ-kernels perform well
  (AIM: 2.2% penalty)
- Extensibility on user-level performs well

http://os.inf.tu-dresden.de/L4/LinuxOnL4/

- 2nd-generation µ-kernels deliver significant better performance than traditionally assumed—not only for microbenchmarks, but also for real-world applications
- we found indications that extensibility in the kernel does not automatically outperform extensibility on user-level. we saw that the effect of using co-location is actually lower than the effect of using a better design.

Thanks for your attention!

Extra slides intended for inquiring questions from the audience follow...

Slide 21

Operation	PCT	IPC	Comment
enter PAL mode	5	5	
open frame	7	7	setup stack frame
send/receive	–	0.5	determine operation
test receiver valid	2	2	
test no chief xfer	–	0.5	
receiver accepts?	–	1	can we do the transfer
set my rcv timeout	–	1	
save rcv parameters	–	2	perform the receive
verify queuing status	–	1	
context switch	10	10	switch address-space number
kernel thread switch	–	6	
set caller id	2	–	save caller id for pct_ret
find callee entry	2	–	pct entry address in callee
close frame	7	7	
leave PAL mode	2	2	
total	38	45	

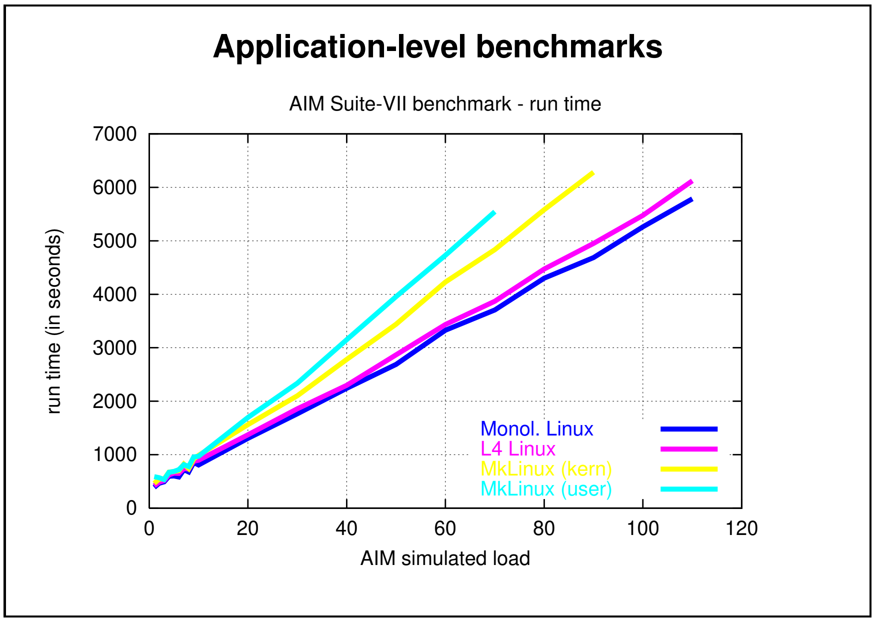
Slide 22

Virtual memory operations

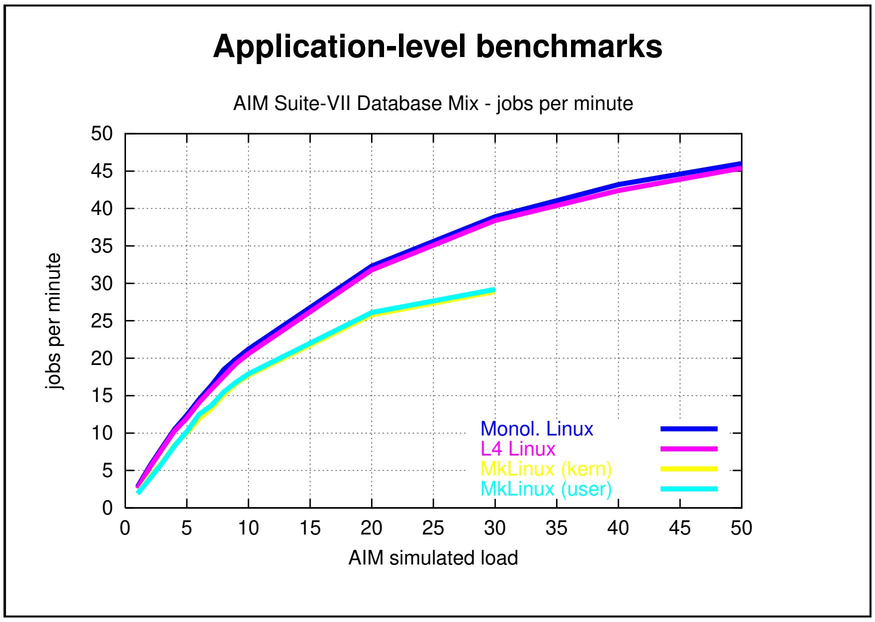
Operation	L^4Linux (133 MHz Pentium)	SPIN (133 MHz Alpha 61064)
Appel/Li 1 (trap+prot1+unprot)	12 µs	39 µs
Appel/Li 2 (prot100+trap+unprot)	10 µs	29 µs

`XXX interpret with caution!!`

Slide 23



Slide 24



Slide 25

getpid comparison

System	Time	Cycles
Linux	1.68 µs	223
L^4Linux	3.95 µs	526
L^4Linux (trampoline)	5.66 µs	753
MkLinux in-kernel	15.41 µs	2050
MkLinux user	110.60 µs	14710

Slide 26

getpid breakdown

Client	Cycles	Server
enter exception handler	20	
send system call message	168	wait for message
	131	— LINUX —
receive reply	188	send reply
leave exception handler	19	
	526	