From L3 to SeL4 What Have we Learnt in 20 Years of L4 Microkernels? (SOSP'13)

ECE5658-41 - Operating System Design

Cassiano Campes

2017.03.13

Difference among kernel models

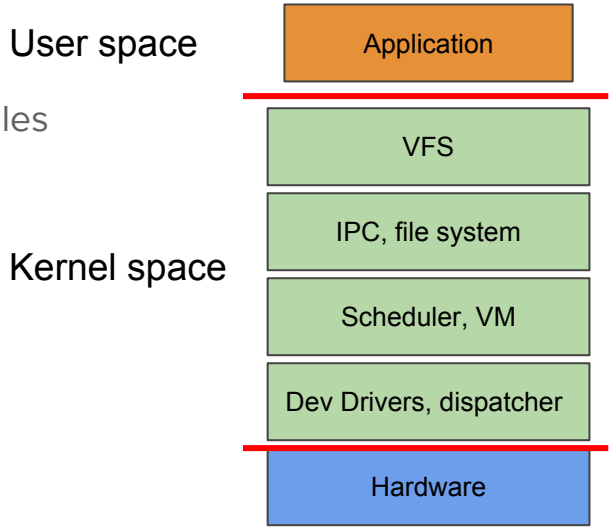
- Monolithic kernels
- Modular kernels
- Microkernels
- Nanokernels
- Exokernels

Difference among kernel models

- Monolithic kernels 
- Modular kernels
- Microkernels 
- Nanokernels
- Exokernels

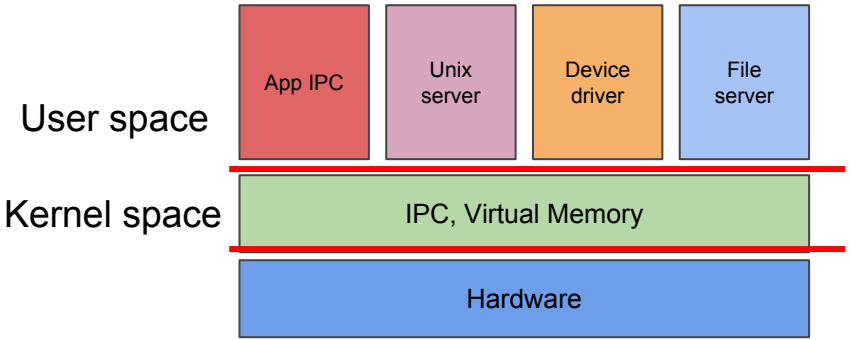
What is a monolithic kernel?

- All basic system services are packaged into a single module in kernel space
- Huge size footprint
- Hard to maintain due to “single block”
  - Modules became possible, dynamically loading kernel modules
  - Maintainability become easy after introduction of modules



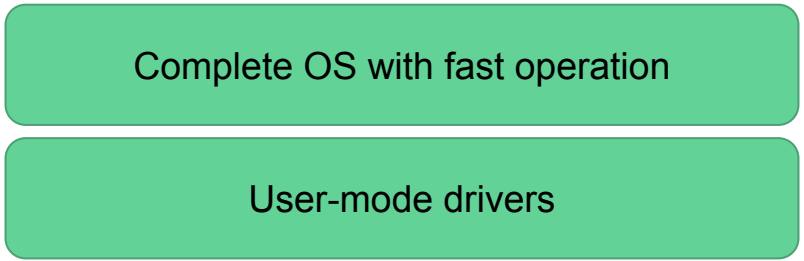
What is microkernel?

- Minimum amount of software that can implement an OS
  - Address space management
  - Thread management
  - Inter-process communication(IPC)



The beginning of L3 kernel - 1988

- Derived from the Eumel OS (aka L2) - 1979 Z80
- Developed by **Jochen Liedtke**
- Developed in the Intel x86 architecture

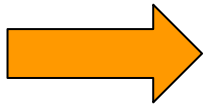


The beginning of L3 kernel - 1988

- Derived from the Eumel OS (aka L2) - 1979 Z80
- Developed by **Jochen Liedtke**
- Developed in the Intel x86 architecture

Complete OS with fast operation

User-mode drivers



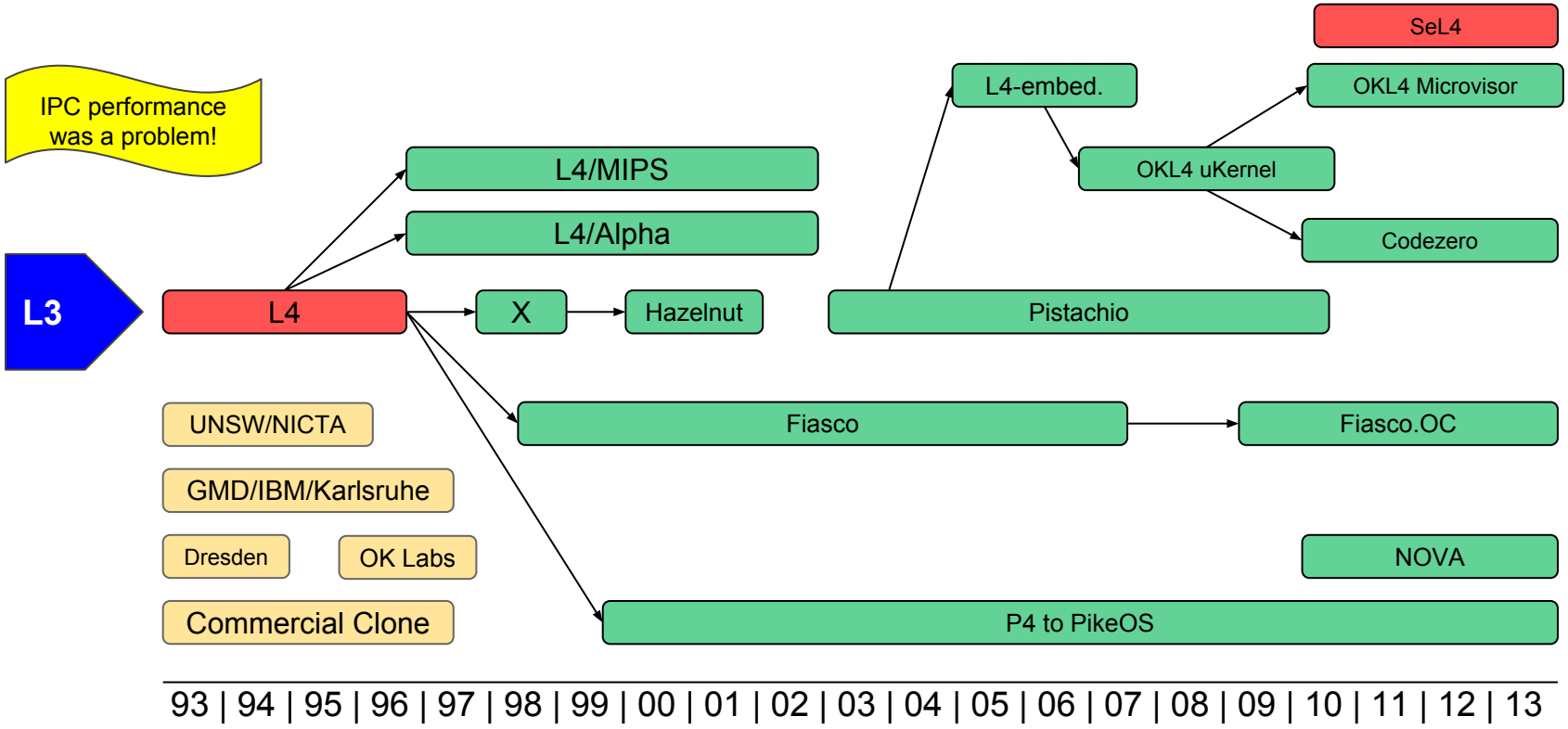
The beginning of L3 kernel - 1988

- Derived from the Eumel OS (aka L2) - 1979
- Developed by **Jochen Liedtke**
- Developed in the Intel x86 architecture

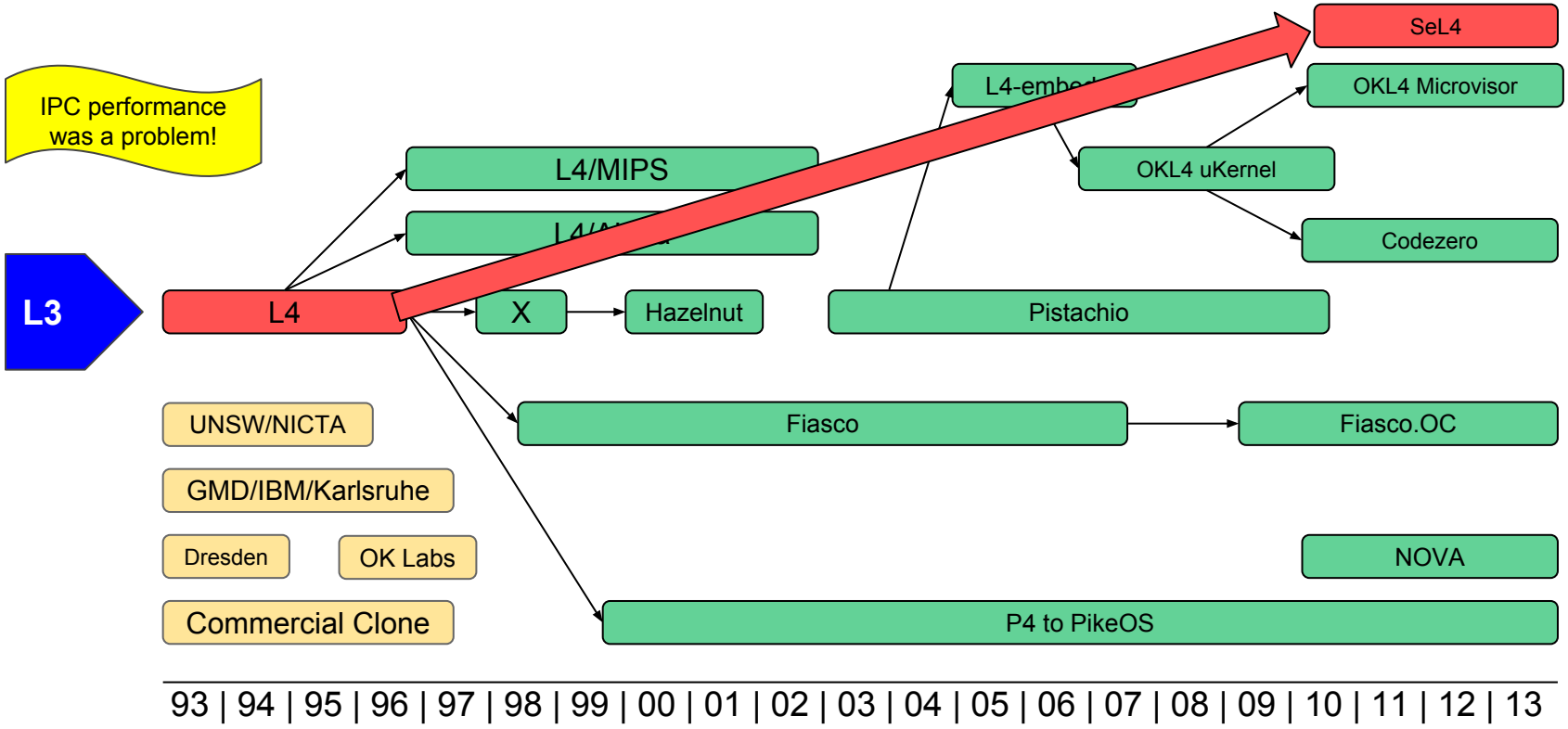
L4 microkernel development came from the re-design of the L3 kernel

mode dr

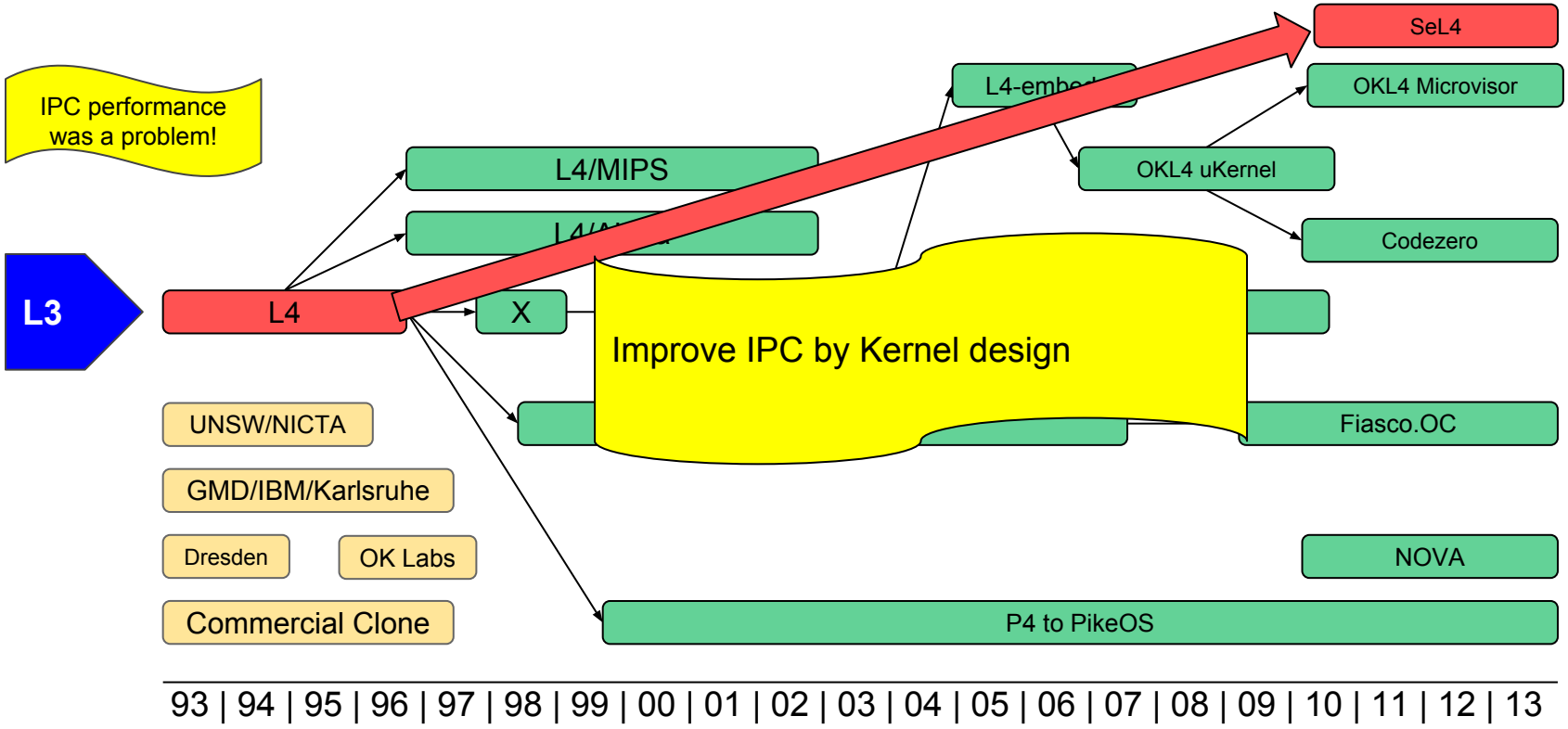
The L4 family tree (simplified)



The L4 family tree (simplified)



The L4 family tree (simplified)



Minimality concept

- “Only minimal mechanisms and no policy in kernel”
  - Challenges the community to remove features or replace with more general ones
  - Nothing that is necessary in kernel must be removed
  - Ensure IPC performance

Name	Archi-tecture	Size (kLOC)		
		C/C++	asm	Total
Original	486	0	6.4	6.4
L4/Alpha	Alpha	0	14.2	14.2
L4/MIPS	MIPS64	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

The L4 IPC challenge

- Some architectures were not context-switch friendly
- Intel Pentium 4 was the worst context-switch friendly
- ITanium 2 was the best context-switch friendly

Name	Year	Processor	MHz	Cycles	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

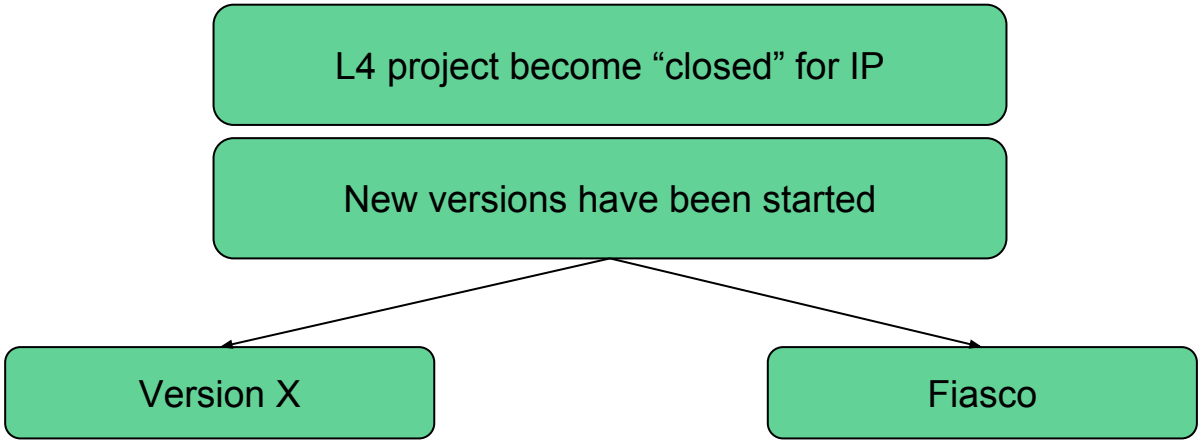
L4 as the evolution of L3

- Liedtke, again, started a re-design of the kernel
- It triggered a twenty-year evolution in the L4 kernel
- Targeting to reduce the IPC overhead to sub-microsecond latency

L4 project become “closed” for IP

L4 as the evolution of L3

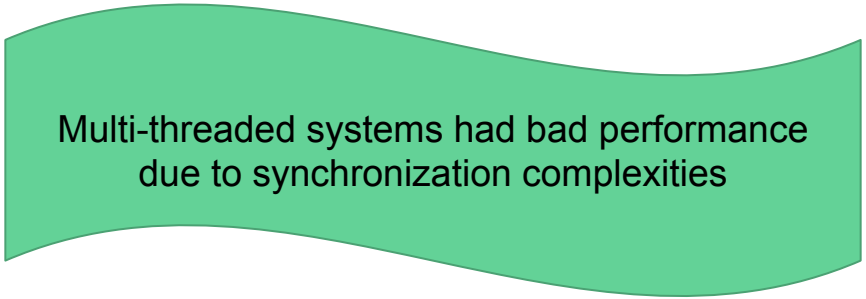
- Liedtke, again, started a re-design of the kernel
- It triggered a twenty-year evolution in the L4 kernel
- Targeting to reduce the IPC overhead to sub-microsecond latency



- Is the oldest L4 codebase still actively maintained

Synchronous IPC

- The original L4 supported synchronous IPC as comm, sync, and sig mechanism
  - It avoids buffering messages in the kernel, reducing the copy overhead
  - The kernel works on the process context, avoiding copying registers back-and-forth



Synchronous IPC

- The original L4 supported synchronous IPC as comm, sync, and sig mechanism
  - It avoids buffering messages in the kernel, reducing the copy overhead
  - The kernel works on the process context, avoiding copying registers back-and-forth

Asynchronous IPC

- Introduced in L4-embedded
  - Simple form of asynchronous IPC
  - Sender does not block; receiver has the choice to block or poll

Synchronous IPC

- The original L4 supported synchronous IPC as comm, sync, and sig mechanism
  - It avoids buffering messages in the kernel, reducing the copy overhead
  - The kernel works on the process context, avoiding copying registers back-and-forth

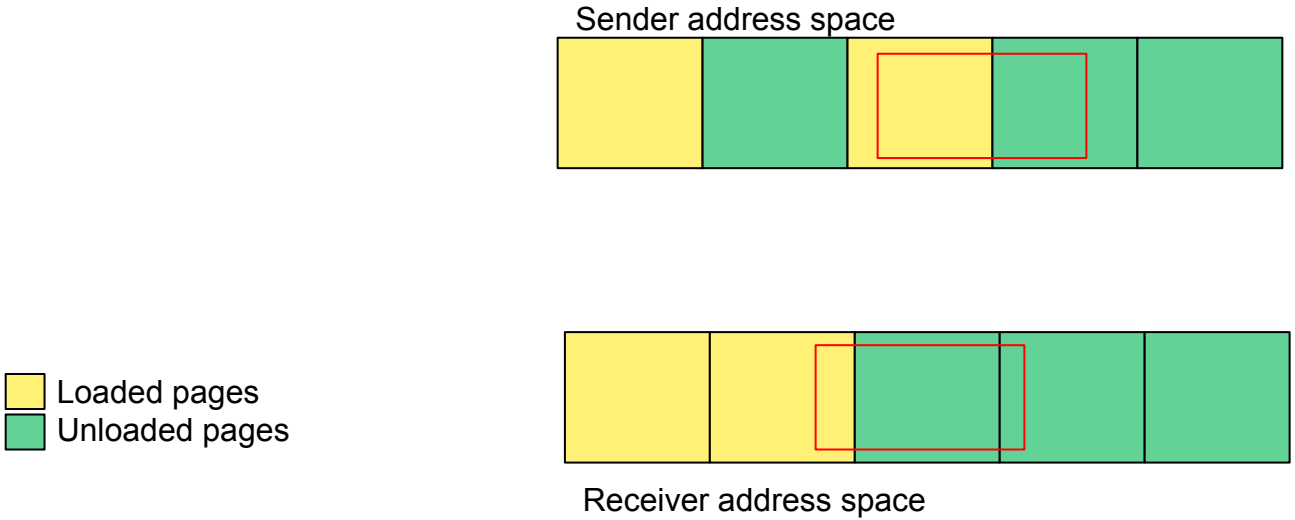
Replaced: Sync IPC augmented with (seL4, NOVA, Fiasco.OC) or replaced by (OKL4) async.

Asynchronous IPC

- Introduced in L4-embedded
  - Simple form of asynchronous IPC
  - Sender does not block; receiver has the choice to block or poll

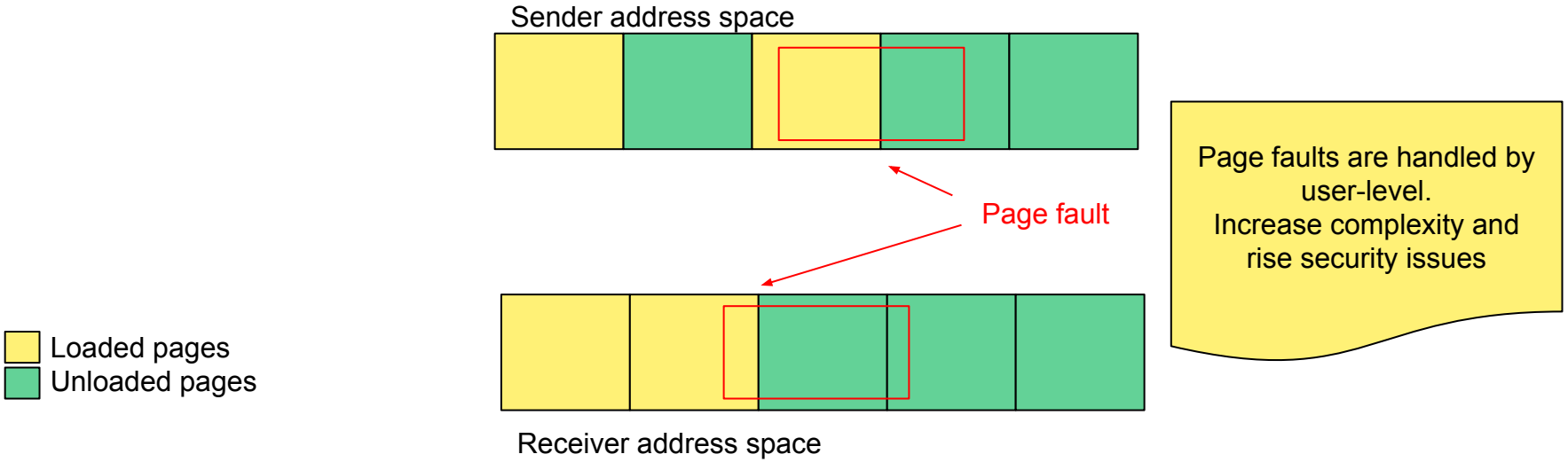
The “long” IPC messages

- Allows long messages to be passed through processes
  - Page faults may occurs frequently



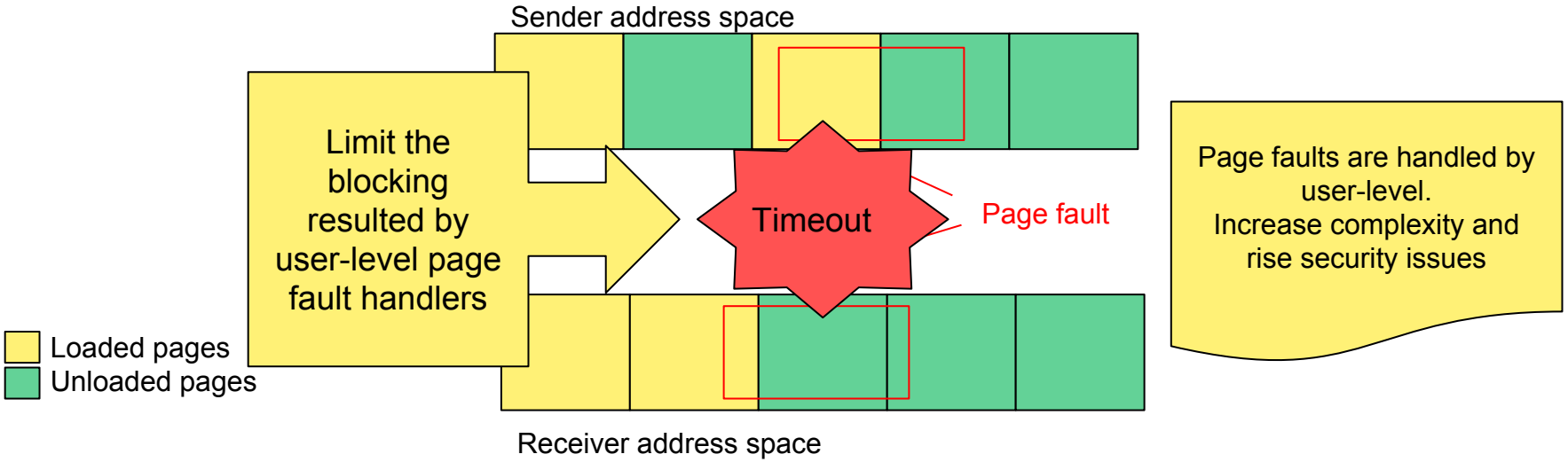
The “long” IPC messages

- Allows long messages to be passed through processes
  - Page faults may occurs frequently



The “long” IPC messages

- Allows long messages to be passed through processes
  - Page faults may occurs frequently



The “long” IPC messages (cont’d)

- Malicious client could send a request and not attempt to collect reply
  - It results in blocking the sender unless watchdogs are implemented (timeouts)
- Malicious client could send long messages to cause page fault
  - Be threatened in user-space giving opportunity to attacks

Hard to define the timeout value

Some L4 forks abandoned timeout definitely

The “long” IPC messages (cont’d)

- Malicious client could send a request and not attempt to collect reply
  - It results in blocking the sender unless watchdogs are implemented (timeouts)
- Malicious client could send long messages to cause page fault
  - Be threate

Abandoned: Long IPC

Abandoned: IPC timeouts in seL4,OKL4

User-level device drivers

- Result of the minimality principle - novelty of L4
- User-level drivers have benefited from performance
  - Applications can easily pass-through device access
  - Reduction in interrupt overhead are noticed

User-level device drivers

- Result of the minimality principle - novelty of L4
- User-level drivers have benefited from performance
  - Applications can easily pass-through device access
  - Reduction in interrupt overhead are noticed

Retained: User-level drivers as a core feature

Kernel memory

- Traditionally L4 had fixed-size heap for its data structures
- Original L4 had a ***kernel pager*** where kernel could request more memory from user-space dynamically
  - Not supported anymore due to security issues
- Open question: how isolate user processes through the shared kernel heap
  - Proposed per-process kernel heaps

Time

- CPU can only be used by a single thread at a time
- Time-multiplexed
  - Fixed-policy scheduler
  - Hurts the policy-freedom religion in microkernel
- Attempts to export scheduling to user-level
  - High overheads, impractical

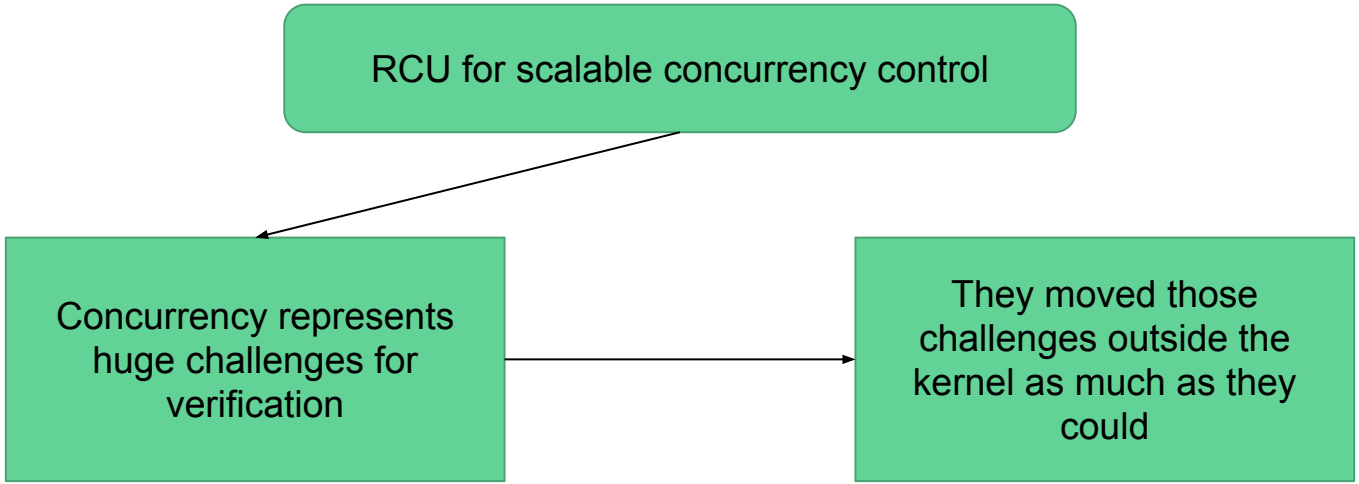
Time

- CPU can only be used by a single thread at a time
- Time-multiplexed
  - Fixed-policy scheduler
  - Hurts the policy-freedom religion in microkernel
- Attempts to export scheduling to user-level
  - High overheads, impractical

Unresolved: policy-free control of CPU time

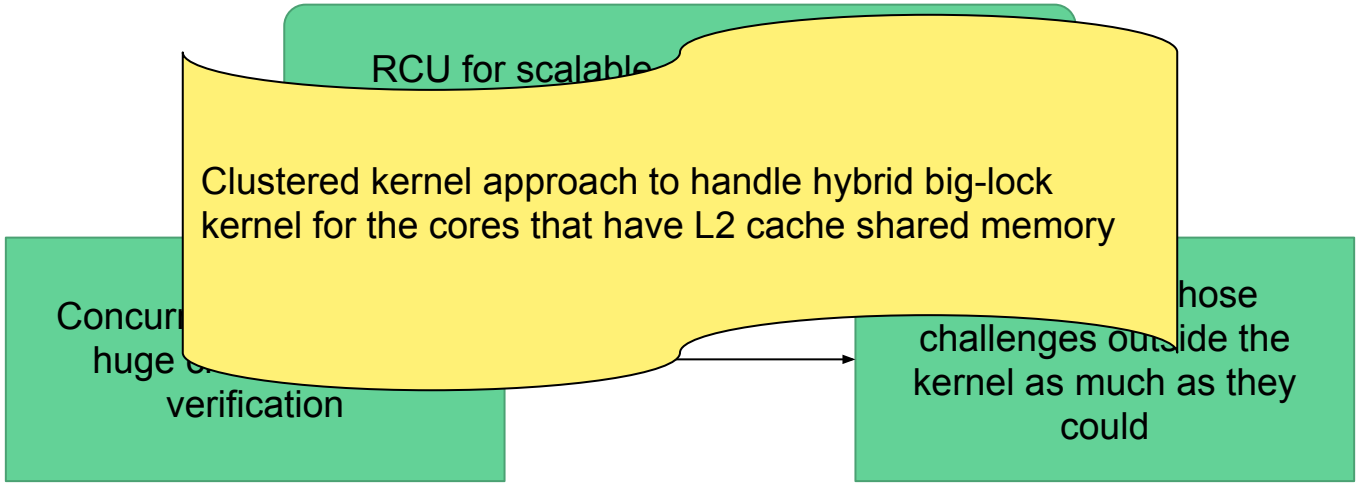
Multi-core systems barrier

- High inter-core communication costs
- No shared caches (makes thread migration hard to be done with low overhead)



Multi-core systems barrier

- High inter-core communication costs
- No shared caches (makes thread migration hard to be done with low overhead)



Lazy scheduling

- When thread blocks on IPC the kernel updates the TCB but leaves it in ready state queue
  - The thread will be ready soon (assumption)
  - Reduces the control involved in changing thread states

Benno scheduling: ready queues contain all runnable threads, except the currently executing one.

Lazy scheduling

- When thread blocks on IPC the kernel updates the TCB but leaves it in ready state queue
  - The thread will be ready soon (assumption)
  - Reduces the control involved in changing thread states

Benno scheduling: ready queues contain all runnable threads, except the currently executing one.

Replaced: Lazy scheduling by Benno scheduling

Direct process switch

- L4 tries to avoid running scheduler during IPC
  - Reduces overhead and latency
- Thread blocked by the IPC, kernel switches thread (gives high priority)
  - This approach called direct process switch

Direct process switch

- L4 tries to avoid running scheduler during IPC
  - Reduces overhead and latency
- Thread blocked by the IPC, kernel switches thread (gives high priority)
  - This approach called direct process switch

Replaced: Direct process switch subject to priorities in seL4 and optional in Fiasco.OC and NOVA.

Preemption

- Traditionally L4 implementations disabled interrupts within kernel mode
- Fiasco work made it fully preemptible for RT performance
- seL4 non-preemptible: ensures latency for safety-critical systems

Preemption

- Traditionally L4 implementations disabled interrupts within kernel mode
- Fiasco work made it fully preemptible for RT performance
- seL4 non-preemptible: ensures latency for safety-critical systems

Retained: Mostly non-preemptible design with strategic preemption points

Non-portability

- Hazelnut kernel and Pistachio achieved 80~90% architecture-agnostic
- seL4 only 50% architecture-agnostic code could be achieved
  - Big part of the code handles the virtual memory specific operations
- For porting NOVA to ARM, 50% of code had to be rewritten

Non-portability

- Hazelnut kernel and Pistachio achieved 80~90% architecture-agnostic
- seL4 only 50% architecture-agnostic code could be achieved
  - Big part of the code handles the virtual memory specific operations
- For porting NOVA to ARM, 50% of code had to be rewritten

Replaced: Non-portable implementation by significant portion of architecture-agnostic code

Implementation language

- The costs to maintain the assembly support become very high
  - Mostly L4 microkernel are written in C/C++ language
  - Ensuring IPC performance
- Compilers become robust - annotations for the likely() path
  - Assembler is not anymore justifiable for performance

Implementation language

- The costs to maintain the assembly support become very high
  - Mostly L4 microkernel are written in C/C++ language
  - Ensuring IPC performance
- Compilers become robust - annotations for the likely() path
  - Assembler is not anymore justifiable for performance

Abandoned: Assembler code for performance

Fiasco was developed in high-level language
C++ with very reasonable performance

Contents not covered in this presentation

- Process hierarchy
- Recursive page mappings
- Strict process orientation
- Virtual TCB array

Please refer to the paper for more details

Conclusions - the principles that matters

- Minimality is the concern about reducing overheads
  - L4 kernels become simple
  - Policy-mechanism separated from the kernel
  - Device drivers handled in user-level
- IPC performance
  - Tradeoff between assembly maintenance and high-level languages
  - Compilers became powerful to build better code
- Past 20 years the IPC performance metric has remained essentially unchanged
  - Even with new ISA developers could keep low cycles for IPCs
  - Code size has remained constant
- Verification and validation have proved the effectiveness of the L4 microkernel

Thank you!

Questions?