TECHNISCHE UNIVERSITÄT DRESDEN

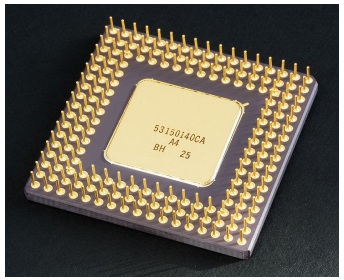
Microkernels and Portability

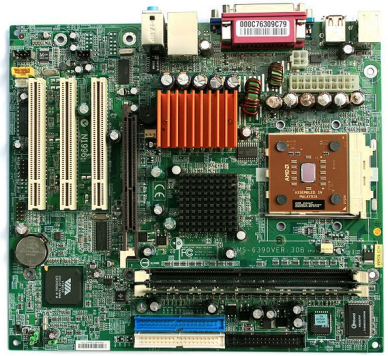
What is Portability wrt Operating Systems?

Reuse of code for different platforms and processor architectures.

Contents

CPUs & Platforms

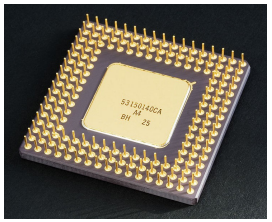




- IA-32 aka x86
- IA-64
- AMD 64
- ARM
- m68k
- PPC ...

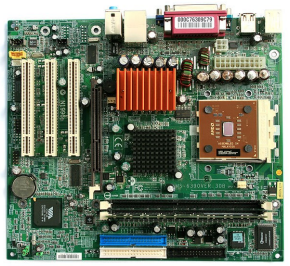
- x86 PC
- HP IA-64 Workstation
- AMD 64 PC
- iPAQ PDA, iPhone
- Atari

Different CPUs



- Instruction Set Architecture (ISA) — CISC, RISC...
- Word and address width — 16bit, 32bit, 64bit...
- Byte order — Little Endian, Big Endian
- Processor state (execution context) — Registers...
- Processor modes
- Memory subsystem (MMU/TLB)
- Caches
- Pipelines

Different Platforms



- Peripheral buses
  - PCI-Bus, ISA-Bus, CAN-Bus, AXI-Bus
- Peripheral devices
  - Graphics adapter, network adapter, UART, timer, IRQ controller ...

History of Microkernel Portability

- First microkernels (Mach) designed for portability
  - Mediocre performance
- Second generation (L4) designed for minimality and performance
  - Assembly language
  - Unportable (however **small and fast**)

    "The additional layer per se costs performance"

    "It cannot take precautions to circumvent or avoid performance problems of specific hardware"

    "Such a µ-kernel cannot take advantage of specific hardware"

    [On µ-Kernel Construction (J. Liedtke SOSP 1995)]

    - Even i486 and Pentium are incompatible

History (2) — Towards Portability

- First HLL implementation of L4
  - Fiasco (designed for preemptability) → acceptable performance
- L4 versions for Alpha, MIPS, ARM (written from scratch)
- New processors every year
  → Maintenance problems

Portable L4 microkernels:
L4::Hazelnut (C/C++), L4/Fiasco (C++), L4::Pistachio (C++)

Prerequisites

- High Level Language (done since Fiasco)
  - Assembler code is per se unportable
  - necessary but not **sufficient**

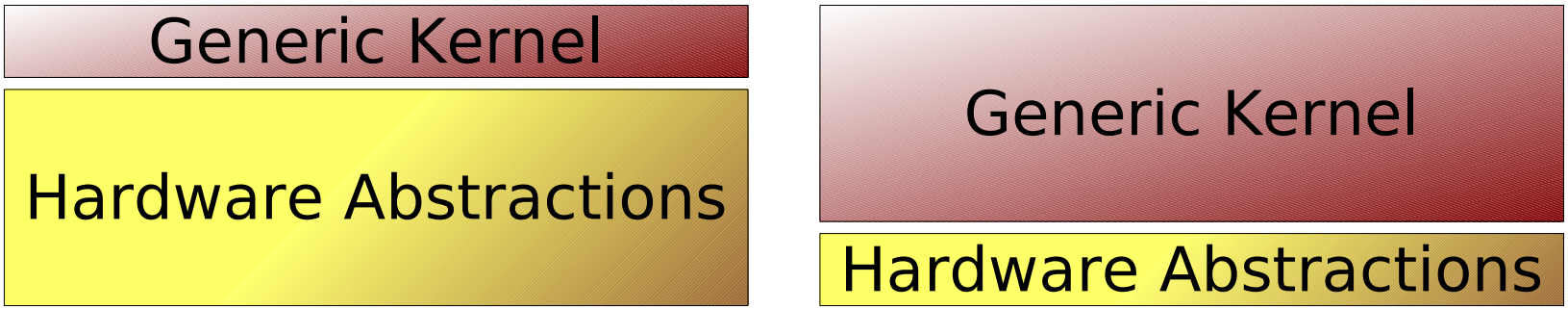
- HOWEVER -

- No OS without some assembler code

  → Only parts of the code are reusable/portable

Tradeoff — Performance vs. Portability

Where to put the boundary ?



L4 Statement

Design Goal is Performance

What to do to achieve maximum performance and acceptable portability?

L4 Microkernels in Detail

CPU: ISA*, pipelines*, word width, byte order, processor modes, processor state, MMU/TLB, caches

Platform: Physical memory layout, boot-up, debugging console, IRQ controller, timer

*Handled by the compiler

Handling in Fiasco

Simple (but important)

- Word width → carefully defined data types
- Byte order → no use of C bit fields for the ABI

Not Performance critical

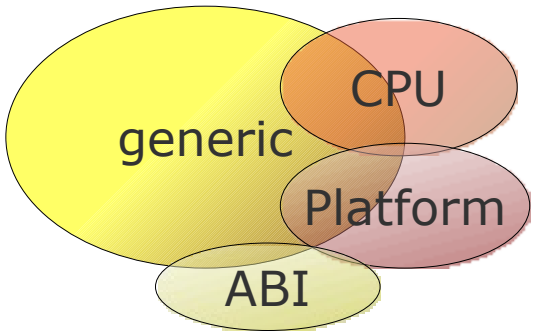
- Bootstrap → some platform-specific init code
- Debug console → complete abstract framework

Handling in Fiasco (2)

More tricky (affect the critical path — IPC)

- Processor modes → mapping to kernel mode and user mode, mode switches
- Processor state → context switches
- MMU/TLB → specific address-space/page-table code
- Caches → specific cache-consistency handling
- IRQ controller → abstract controller interface

Fiasco Structure

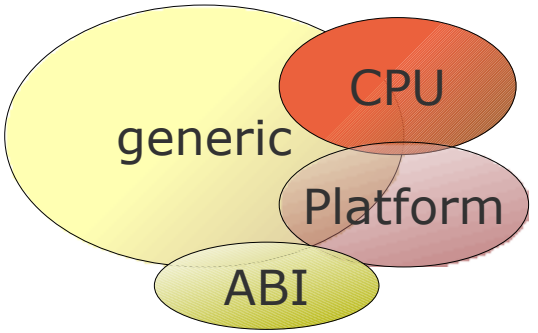


Generic Code

- Memory management
- Page-fault handling
- IPC Path
- Mapping database
- Base of the kernel debugger
- Most code of L4 abstractions
  - Thread and address-space management

Fiasco Structure (2)

Processor and ABI Specific Code

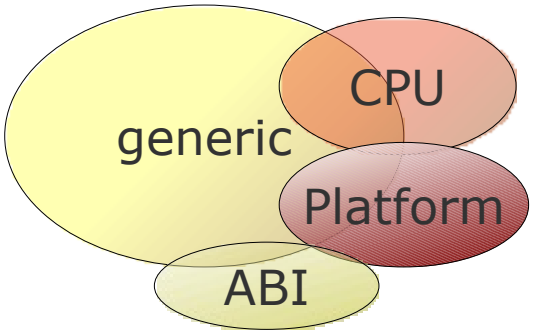


- Basic data types
- Processor abstraction
  - IRQ control, sleep-mode support
- Atomic operations
- Page tables
- Parts of L4 abstractions
  - Switch of CPU and FPU state
- CPU specific optimizations

Fiasco Structure (3)

Platform Specific Code

- Interrupt-controller driver
- UART driver (debugger)
- Graphics adapter driver (debugger)



Interaction of Code Parts

Interesting interactions:

- Generic code calls *x*-specific functions
- Data structures contain generic and *x*-specific members

Solution: (Fiasco is C++)

- Use OO principles, such as inheritance and polymorphism

Polymorphism — C++ Virtual Functions

- Virtual functions impose indirections (Indirections are necessary for runtime polymorphism)

- Hardware abstractions are *compile-time* polymorphism
  - Indirections are not acceptable
    - ✗ Extra memory references
    - ✗ Extra cache pollution
    - ✗ **No function inlining**

Solutions for the Virtual-Functions Dilemma

- Use a good compiler with virtual-function elimination
  - ✗ No known compiler with this feature
- *Class Flattening* — developed in Karlsruhe
  - Can deal with data-member layout and optimize it
  - ✗ Is a research project
- *Preprocess* — C++ Preprocessor
  - Provides *class extension* for compile-time diversity
  - Can deal with data-member layout (no optimization)
  - ✔ Is used by Fiasco since the beginning

Break Point

We already know

- What is portability about
- What is the basic structure of Fiasco
- How do we solve the diversity (in theory)

Practical Comparison — IA-32, IA-64, and ARM

- **Processor Modes:** Fiasco model knows *user mode* and *privileged mode* (critical for kernel entries and exits)
- **Processor State:** Complete register set and status words (critical for thread switches)
- **MMU/TLB:** L4 Address-Space Model (critical for task switches)
- **Caches:** Cache consistency must be maintained (critical for task switches)

Processor Modes

IA-32
- 3 = User*
- 2 = ...
- 1 = ...
- 0 = Kern*

IA-64
- 3 = User*
- 2 = ...
- 1 = ...
- 0 = Kern*

ARM
- User*
- Abort
- Undefined
- IRQ
- FIQ
- Supervisor*
- System

*effectively used in Fiasco; green modes are privileged

Processor State

IA-32
- 8 GPRs
- FPU state

IA-64
- 32 GPRs
- Register Stack
  - min. 96 GPRs
  - Backing store
- FPU state

ARM
- 15 GPRs
- FPU state?

- ca. 140 Byte w/o MMX/SSE
- ca. 0.5kB mit MMX/SSE

- ca. 1kB GPRs + 2kB FPU

- ca. 64 Bytes + (FPU?)

MMU/TLB

IA-32

- HW page tables
- Implicit/explicit TLB control
- Segmentation

IA-64

- SW loaded TLB
- HW page table option
- Decoupled protection
- TLB tagging
- Explicit TLB control

ARM

- HW page table
- Decoupled protection
- kind of TLB tagging
- Explicit TLB control

Caches

IA-32

- Phys. tagged and indexed
- HW consistency

IA-64

- Phys. tagged and indexed
- HW consistency

ARM

- Virt. tagged and indexed
- SW consistency

Issues

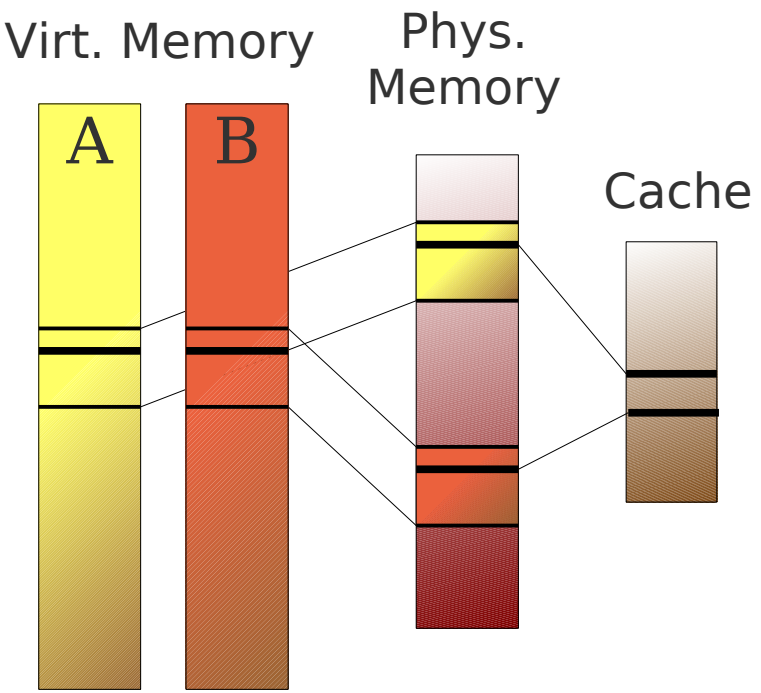
- TLB consistency
- Cache consistency
- Use of specific hardware
  - Reduce TLB overhead
  - Reduce cache overhead
  - Reduce processor state save/restore-overhead

TLB Consistency
w/o address-space IDs in the TLB

- TLB flush on address-space switches (always)
- TLB flush on page-rights downgrade
- IA-64 and ARM
  - TLB flush on page-rights upgrade (eager or lazy)

TLB flushes have high direct and indirect costs

Cache Consistency
Physically Tagged/Indexed Caches

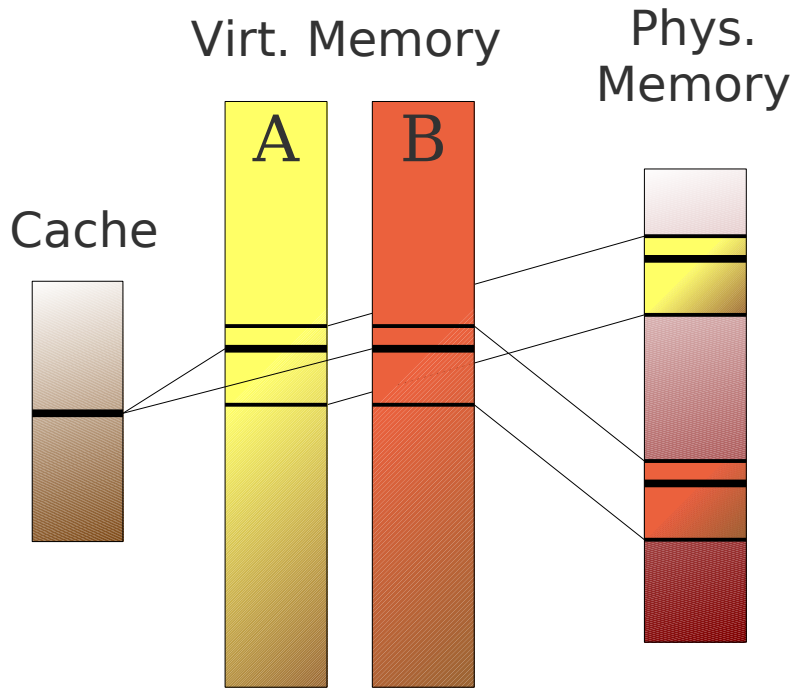


- Different phys. addresses map to different cache location

→ No consistency problems

→ No overhead on task switches

Cache Consistency (2)
Virtually Tagged/Indexed Caches

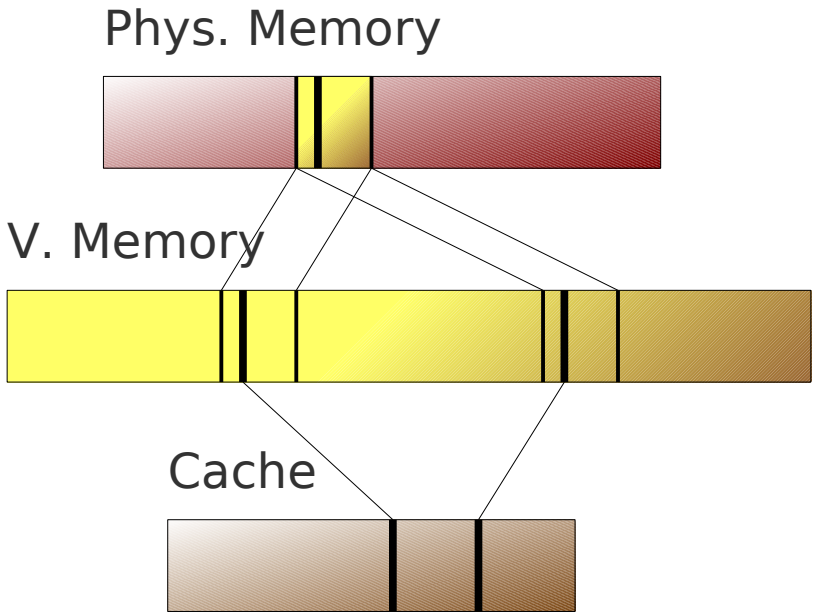


- Same virtual addresses map to same cache line

→ Cache flush on address-space switches

→ Extremely high direct and indirect costs for task switches

Cache Consistency (3)
Virtually Tagged/Indexed Caches



- Aliasing of same phys. frame

→ Duplicated cache allocation

→ Must have countermeasures for this problem

Conclusion

Where to put the boundary between generic and hardware-specific code?

There is no single answer to this question.

- All code that must be HW specific is HW specific
- Some optimization code is HW specific
  - More optimizations worse maintainability
- Carefully designed generic code allows the use of HW specific optimizations with low portability overhead