RWTH AACHEN UNIVERSITY

Bare-Metal Execution of Hard Real-Time Tasks Within a General-Purpose Operating System

Georg Wassen and Stefan Lankes

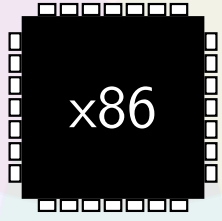
Operating Systems Research Group

7 July 2015

Research Group Background

Foundations (Teaching)

- OS, Interrupts
- Memory Management
- Assembler (low-level)



Real-Time

- Linux, RTOS
- Verification
- API: POSIX

Parallel Systems

- Synchronization
- Communication
- Threads

Agenda

Motivation

Concept

Implementation

Evaluation

Conclusion

RWTH AACHEN UNIVERSITY

Motivation

Concept

Implementation

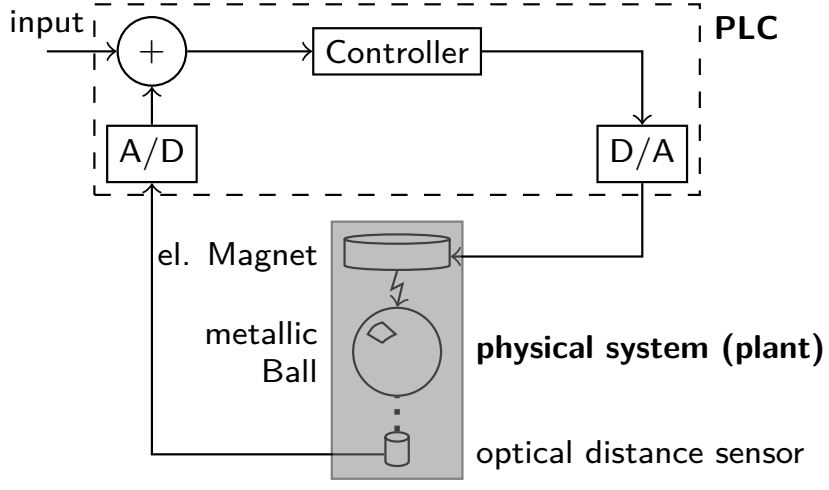
Evaluation

Conclusion

Application Area

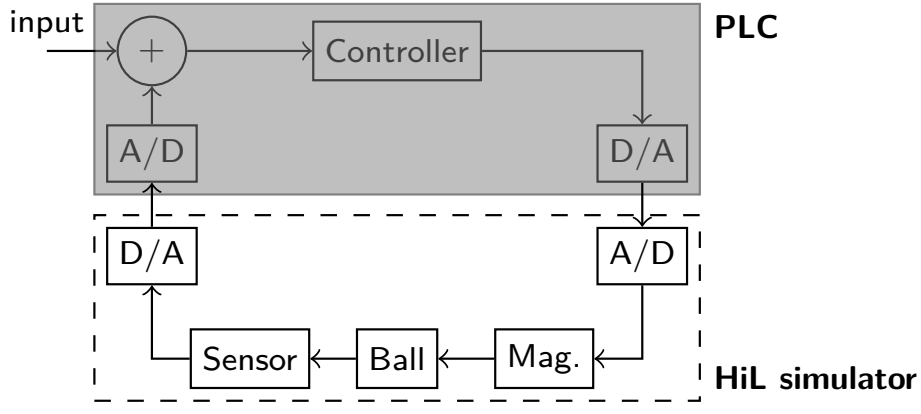
Closed-Loop Control

- Given: Physical system
- Control Model
- Real-Time: Programmable Logic Controller

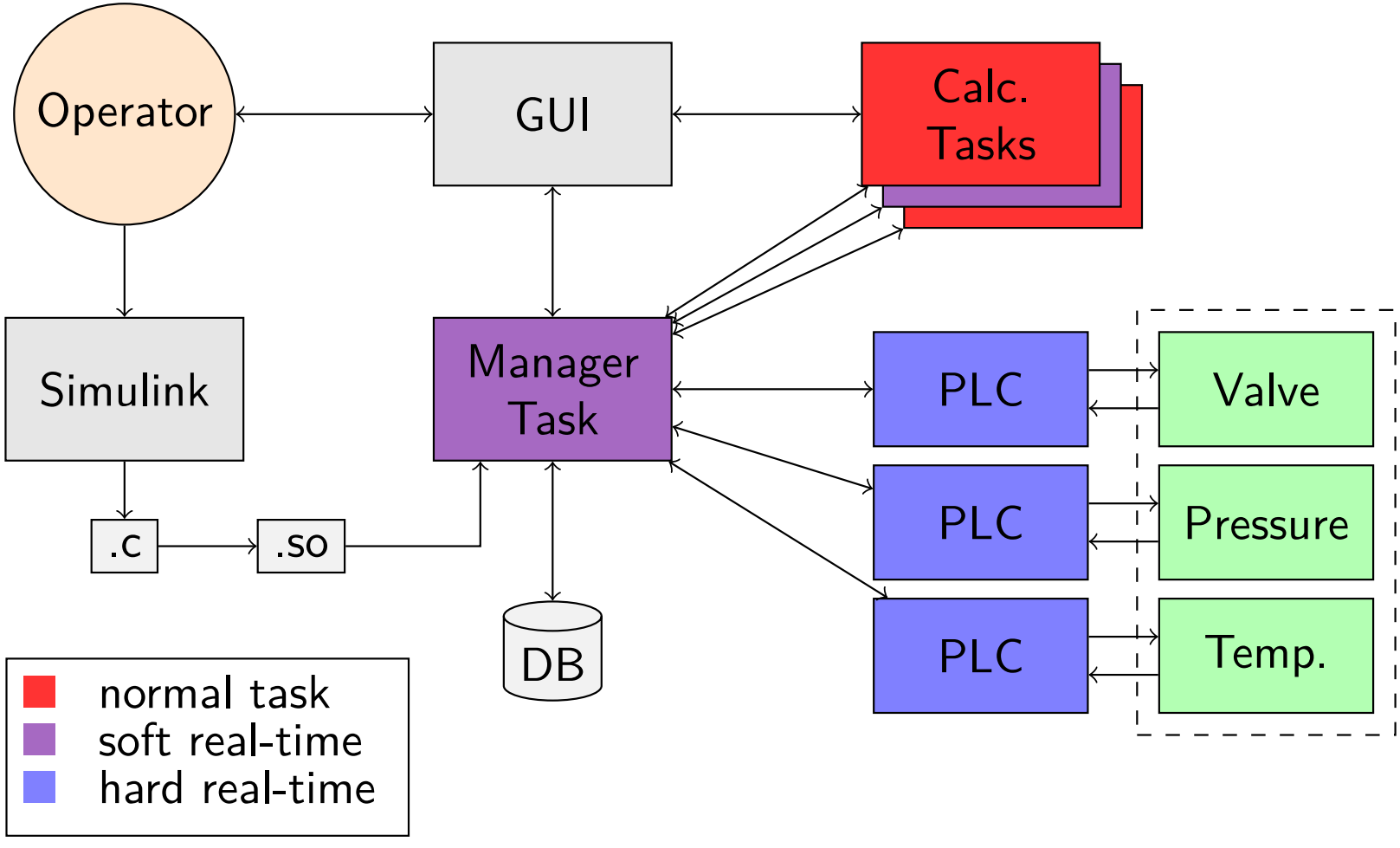


Hardware-in-the-Loop Simulation

- Validation of PLC implementation (μC)
- When physical system not available



Control Application Example



Control Application Example

Goal/Demands

- Hard real-time I/O with latency below 10 μs
- Soft real-time tasks requiring high compute power
- Threaded Application utilizing multiple CPUs
- Versatility: build on existing code and use available libraries

System

- Multi-processor x86 (Non-Uniform Memory Architecture)
- PCI-Express I/O adapter(s)
- Linux (C/C++, POSIX, Qt, etc.)

(Concept generally transferable to other architectures)

RWTH AACHEN UNIVERSITY

Motivation

Concept

Implementation

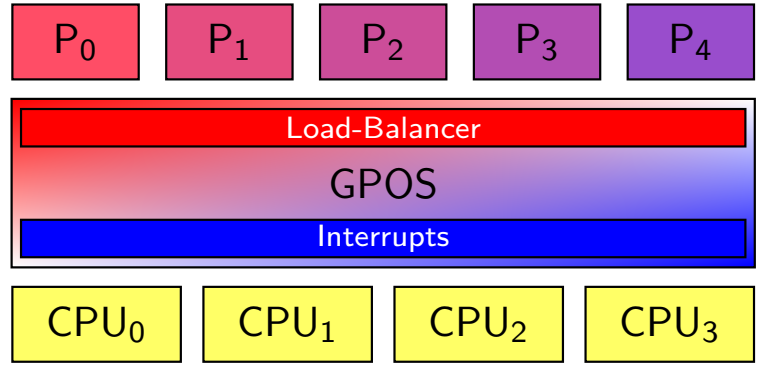
Evaluation

Conclusion

Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

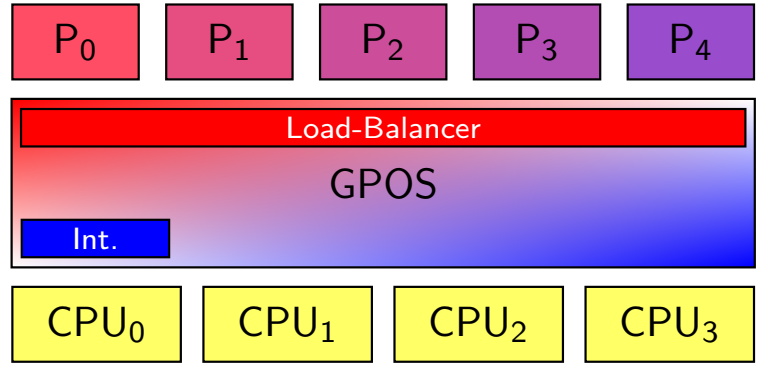


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)

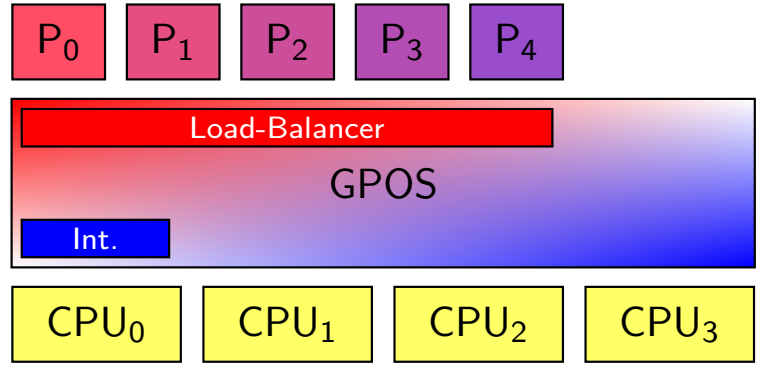


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)
- CPU-Set (to partition CPUs)

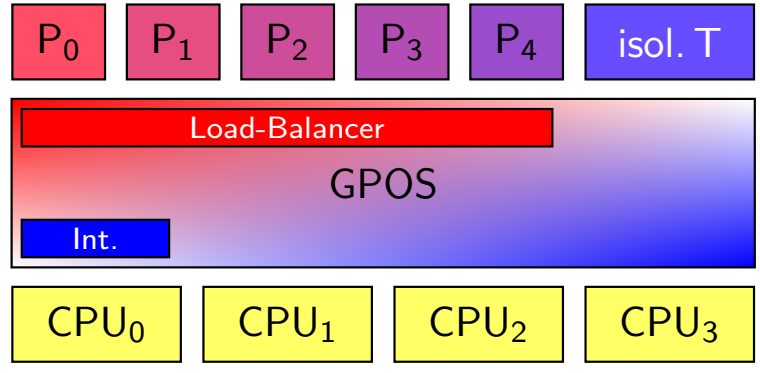


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)
- CPU-Set (to partition CPUs)
- Isolated Task on dedicated CPU

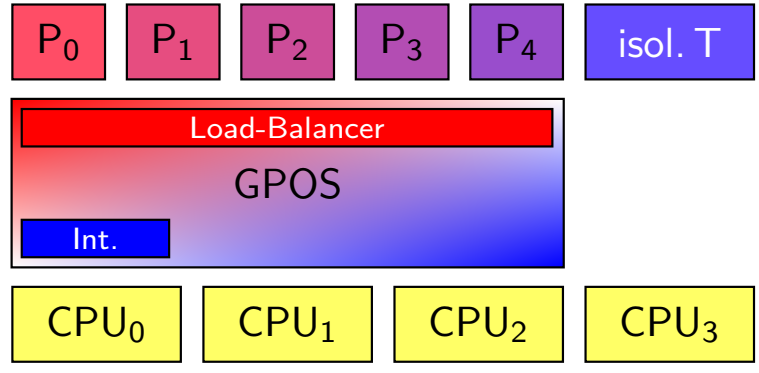


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)
- CPU-Set (to partition CPUs)
- Isolated Task on dedicated CPU
- No Syscalls, Timer deactivated

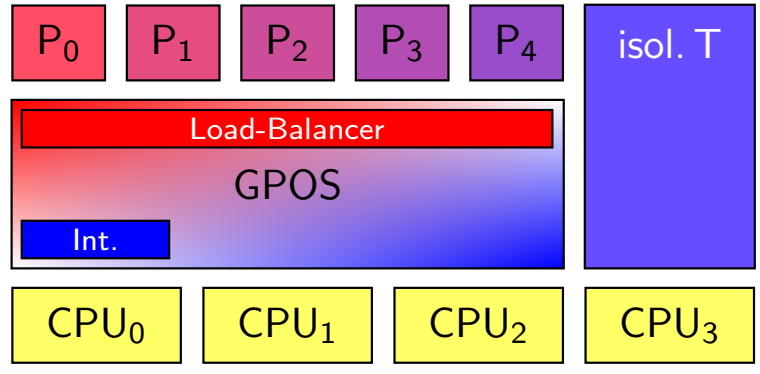


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)
- CPU-Set (to partition CPUs)
- Isolated Task on dedicated CPU
- No Syscalls, Timer deactivated
- Direct Hardware Access (I/O)

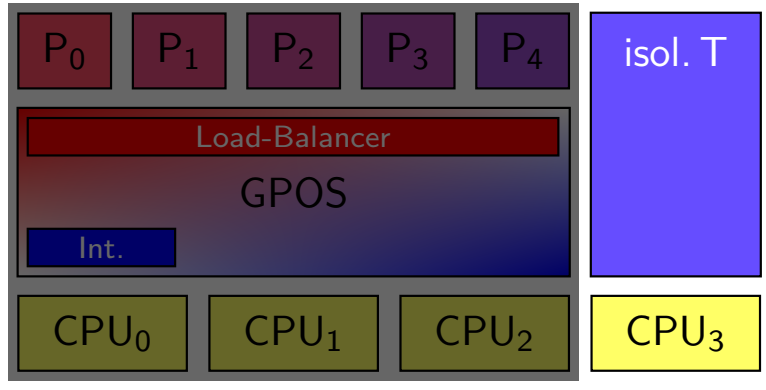


Hard Real-Time by Isolation

Concept of Bare-Metal Tasks

Multi-Processor System with Standard Operating System (GPOS)

- IRQ Affinity (to bind Interrupts)
- CPU-Set (to partition CPUs)
- Isolated Task on dedicated CPU
- No Syscalls, Timer deactivated
- Direct Hardware Access (I/O)
- Communication via shared memory



Verification of Real-Time (Scheduling)

Analogy: *Bare-Metal* Execution on Single-Processor System

RWTH AACHEN UNIVERSITY

Motivation

Concept

Implementation

Evaluation

Conclusion

Preliminary Analysis

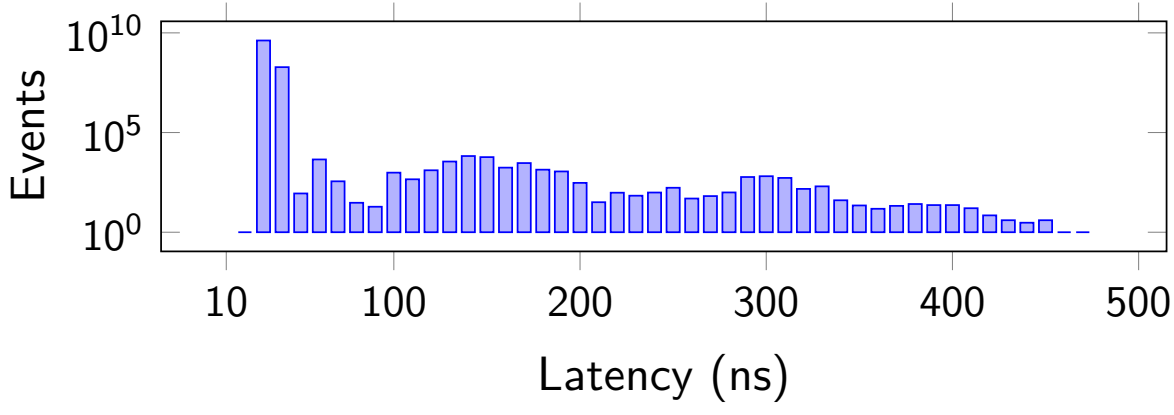
- No Modification of Linux Kernel, portable C Library
- Flexible Configuration

Preliminary Analysis

- No Modification of Linux Kernel, portable C Library
- Flexible Configuration
- Results on Intel Core i7 (Nehalem)
  - ➔ Heavy Load on Remaining Cores
    (CPU, Memory, Interrupts, Fork-Bomb, etc.)
  - ➔ 1 h Benchmark
  - ➔ Hourglass-Algorithm: RDTSC-Loop, Detecting *Gaps* in the Execution

Preliminary Analysis

- No Modification of Linux Kernel, portable C Library
- Flexible Configuration
- Results on Intel Core i7 (Nehalem)
  - ➔ Heavy Load on Remaining Cores
    (CPU, Memory, Interrupts, Fork-Bomb, etc.)
  - ➔ 1 h Benchmark
  - ➔ Hourglass-Algorithm: RDTSC-Loop, Detecting *Gaps* in the Execution
  - ➔ max. Jitter: $< 0{,}5\,\mu$s (Without Load: Close to 0)



- ➔ Impact from Shared L3 Cache, not the OS

Bare-Metal Task in User-Mode

- Complete Isolation (no Interrupts, no Syscalls)
  - ➔ Temporal Behavior of Kernel code unpredictable
- Only Known Code
  - ➔ Formal Verification Possible
- Now: Reconstruct Usability
  - ➔ Synchronization, Communication
  - ➔ I/O Access, Drivers

Trade-Off and Work-Around

- Initialization
- Inter-Process Communication
- Interrupts
- Further Operating System Services

Trade-Off and Work-Around

- Initialization (e. g. `malloc()`):
  - ➔ Before (Start-up Phase)
  - ➔ No Dynamic Data Structures Possible (use Ring-Buffer)
- Inter-Process Communication
- Interrupts
- Further Operating System Services

Trade-Off and Work-Around

- Initialization (e. g. `malloc()`):
  - ➔ Before (Start-up Phase)
  - ➔ No Dynamic Data Structures Possible (use Ring-Buffer)
- Inter-Process Communication: based on Shared-Memory (User-Mode: Atomic Operations and Active Waiting)
  - ➔ Flag, Mutex, Barrier (Synchronization)
  - ➔ Lock- and Wait-free data structures
- Interrupts
- Further Operating System Services

Trade-Off and Work-Around

- Initialization (e. g. `malloc()`):
  - ➔ Before (Start-up Phase)
  - ➔ No Dynamic Data Structures Possible (use Ring-Buffer)
- Inter-Process Communication: based on Shared-Memory (User-Mode: Atomic Operations and Active Waiting)
  - ➔ Flag, Mutex, Barrier (Synchronization)
  - ➔ Lock- and Wait-free data structures
- Interrupts
  - ➔ Handling in User-Mode possible (Forwarding)
  - ➔ Preemptive Scheduling in Isolated Process
- Further Operating System Services

Trade-Off and Work-Around

- Initialization (e. g. `malloc()`):
  - ➔ Before (Start-up Phase)
  - ➔ No Dynamic Data Structures Possible (use Ring-Buffer)
- Inter-Process Communication: based on Shared-Memory (User-Mode: Atomic Operations and Active Waiting)
  - ➔ Flag, Mutex, Barrier (Synchronization)
  - ➔ Lock- and Wait-free data structures
- Interrupts
  - ➔ Handling in User-Mode possible (Forwarding)
  - ➔ Preemptive Scheduling in Isolated Process
- Further Operating System Services
  - ➔ Helper Process
  - ➔ Error Handling (Endless Loop, Exceptions)

Programming

RWTH AACHEN UNIVERSITY

For the best Performance

- Parallel Systems:

- Especially for Isolated Tasks:

Programming

RWTH AACHEN UNIVERSITY

For the best Performance

- Parallel Systems:
  - ➔ Regard Synchronization
  - ➔ Avoid False-Sharing
    (C++: Where is the Data?)
- Especially for Isolated Tasks:

Programming

For the best Performance

- Parallel Systems:
  - ➔ Regard Synchronization
  - ➔ Avoid False-Sharing
    (C++: Where is the Data?)
- Especially for Isolated Tasks:
  - ➔ no SysCalls
    ("does memcpy() contain a SysCall?")
  - ➔ avoid Faults (Page, FPU, . . .)
    - » Warm-up: Pre-Fault all Pages

Stabilizing the System

RWTH AACHEN UNIVERSITY

Observation

- So far: idle System or static Load
- Under dynamic Load: System CPUs block after few Minutes

Stabilizing the System

RWTH AACHEN UNIVERSITY

Observation

- So far: idle System or static Load
- Under dynamic Load: System CPUs block after few Minutes

Causes

- System Blocks: Kernel does not tolerate CPUs not responding
- Memory Consumption: Kernel Buffers not freed

➔ Changes to the Kernel unavoidable

Linux Kernel Modification

- Imitate CPU Hotplugging
  - ➔ Notify subsystems, but leave one process executing
  - ➔ Fixes Read-Copy-Update and Slab Kernel memory
  - ➔ Fixes problems with the wall clock system
- Mask Inter-Processor Interrupts
  - ➔ Identify architectural implementation
  - ➔ Asynchronous: just don't send them
  - ➔ Waiting: skip masked CPUs
- Deactivate Timer Interrupt
  - ➔ Interrupt flag must not be cleared
  - ➔ Allows to recover from stuck real-time code

Linux Kernel Modification

- Imitate CPU Hotplugging
  - ➔ Notify subsystems, but leave one process executing
  - ➔ Fixes Read-Copy-Update and Slab Kernel memory
  - ➔ Fixes problems with the wall clock system
- Mask Inter-Processor Interrupts
  - ➔ Identify architectural implementation
  - ➔ Asynchronous: just don't send them
  - ➔ Waiting: skip masked CPUs
- Deactivate Timer Interrupt
  - ➔ Interrupt flag must not be cleared
  - ➔ Allows to recover from stuck real-time code

Kernel Modification

4 Files Changed.

System is stable (72h) and used in a Production Environment.

RWTH AACHEN UNIVERSITY

Motivation

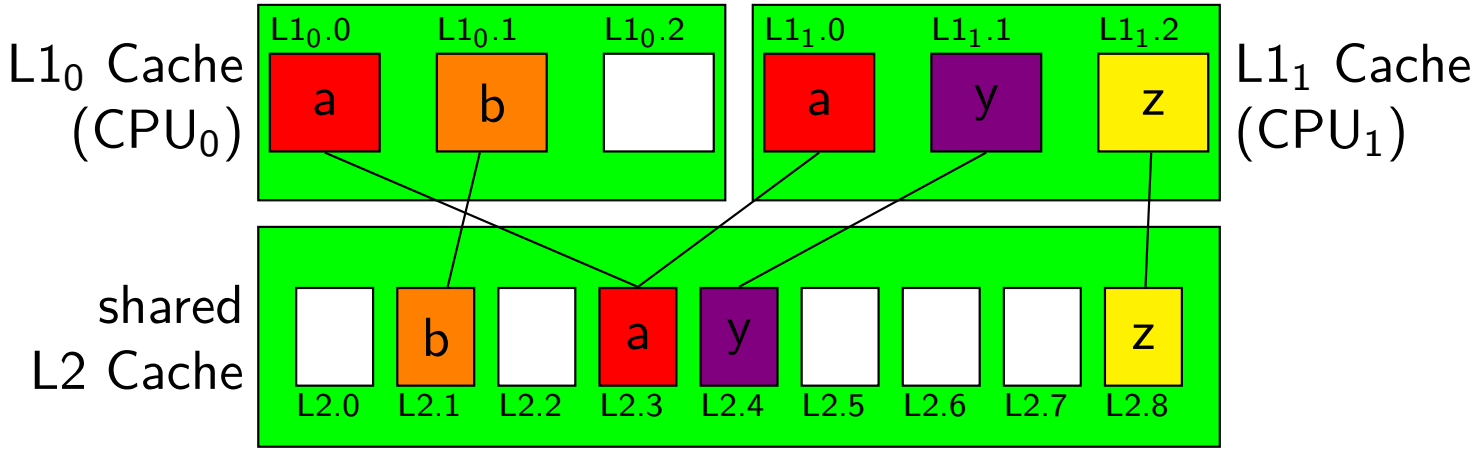
Concept

Implementation

Evaluation

Conclusion

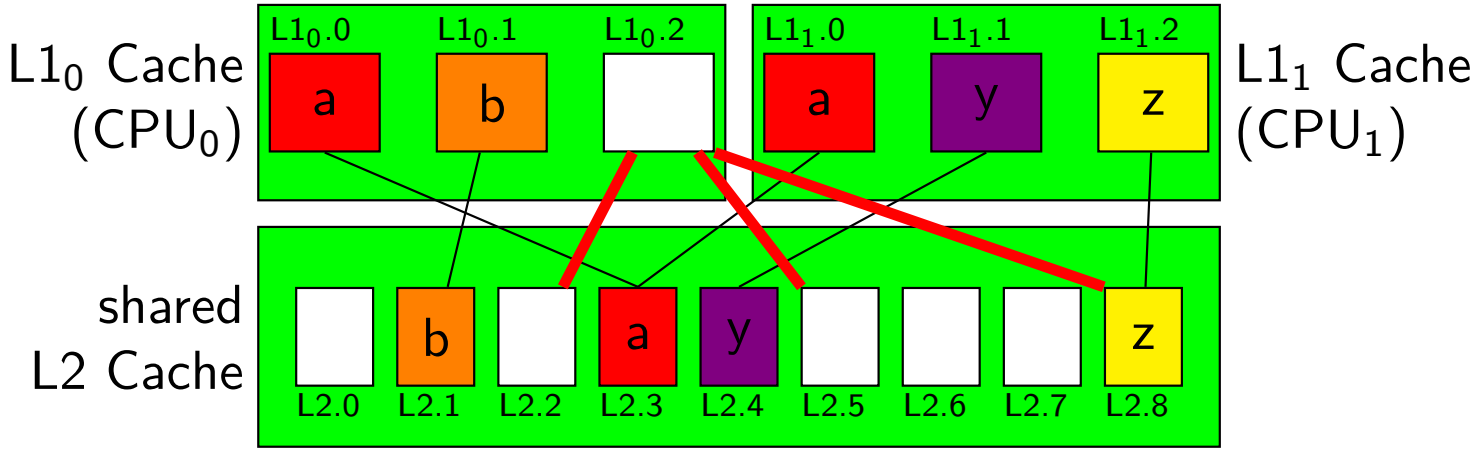
Cache Victims



Intel: Inclusive Cache

- All Elements of L1$ also in L2$

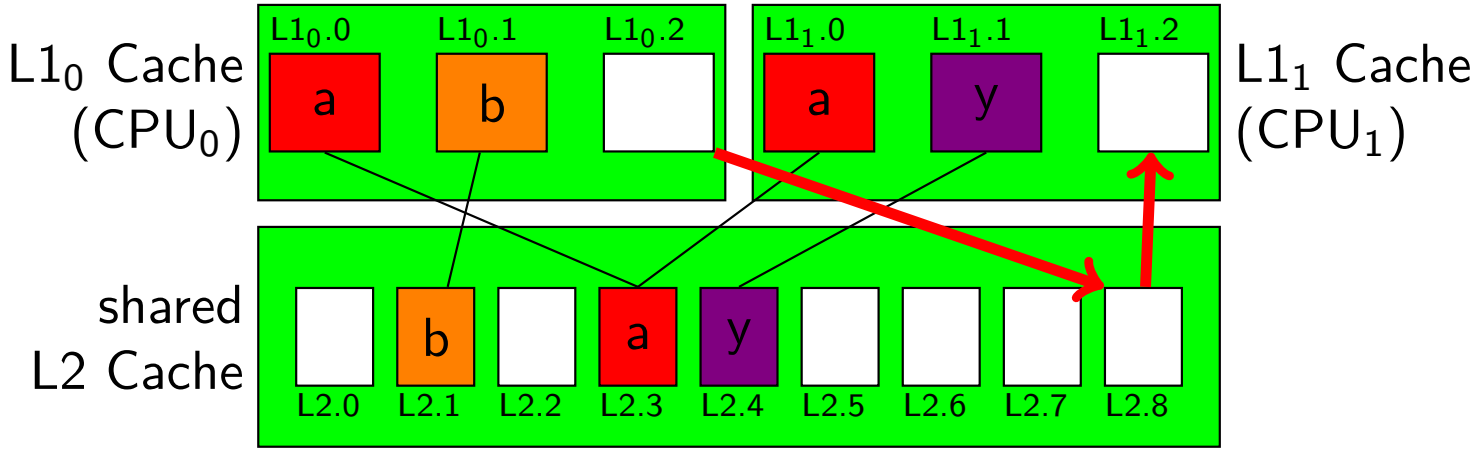
Cache Victims



Intel: Inclusive Cache

- All Elements of L1$ also in L2$
- Cache Miss: Associativity restricts Selection of L2.

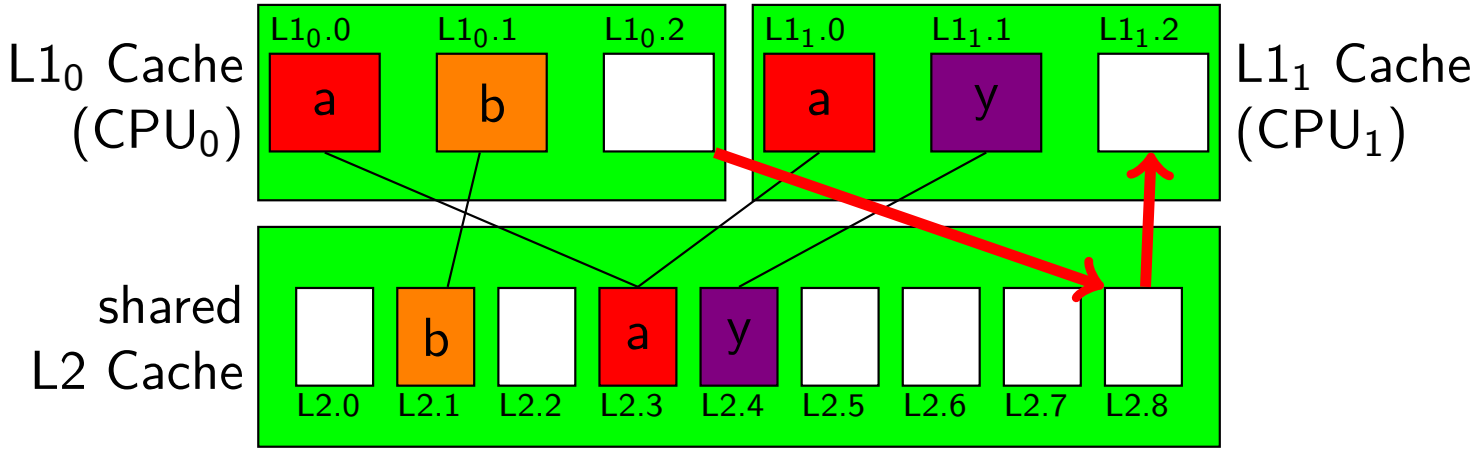
Cache Victims



Intel: Inclusive Cache

- All Elements of L1$ also in L2$
- Cache Miss: Associativity restricts Selection of L2.
- Inclusivity: Eviction from L1 of other Cores possible.

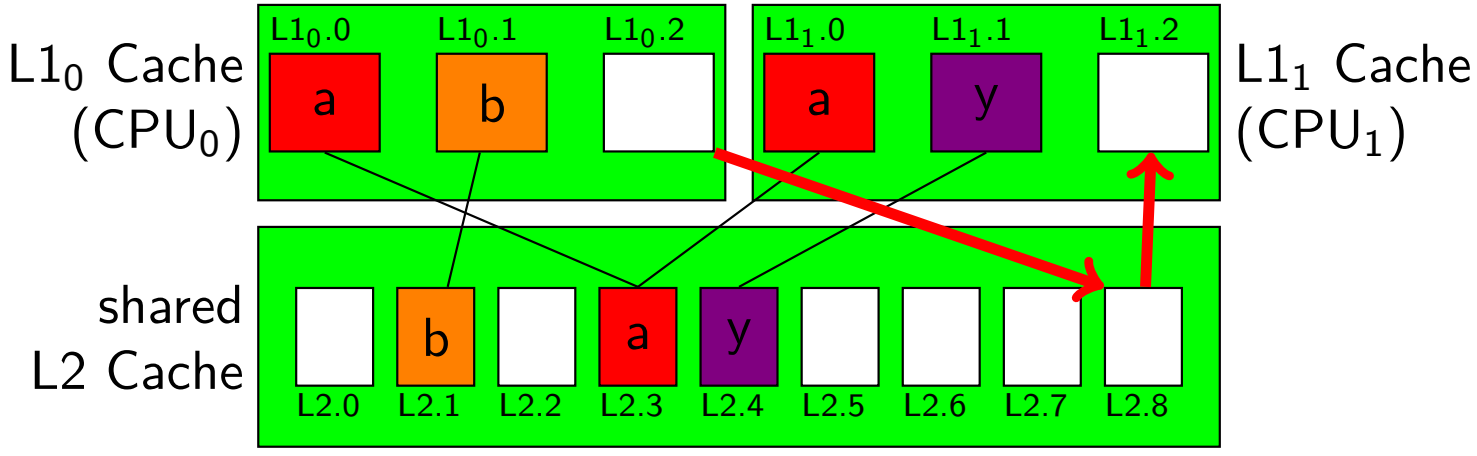
Cache Victims



Intel: Inclusive Cache

- All Elements of L1$ also in L2$
- Cache Miss: Associativity restricts Selection of L2.
- Inclusivity: Eviction from L1 of other Cores possible.
- Measurable Effect (short Loop in Isolation):
  - ➔ Other CPU uses small Buffer: 44 – 64 Cycles

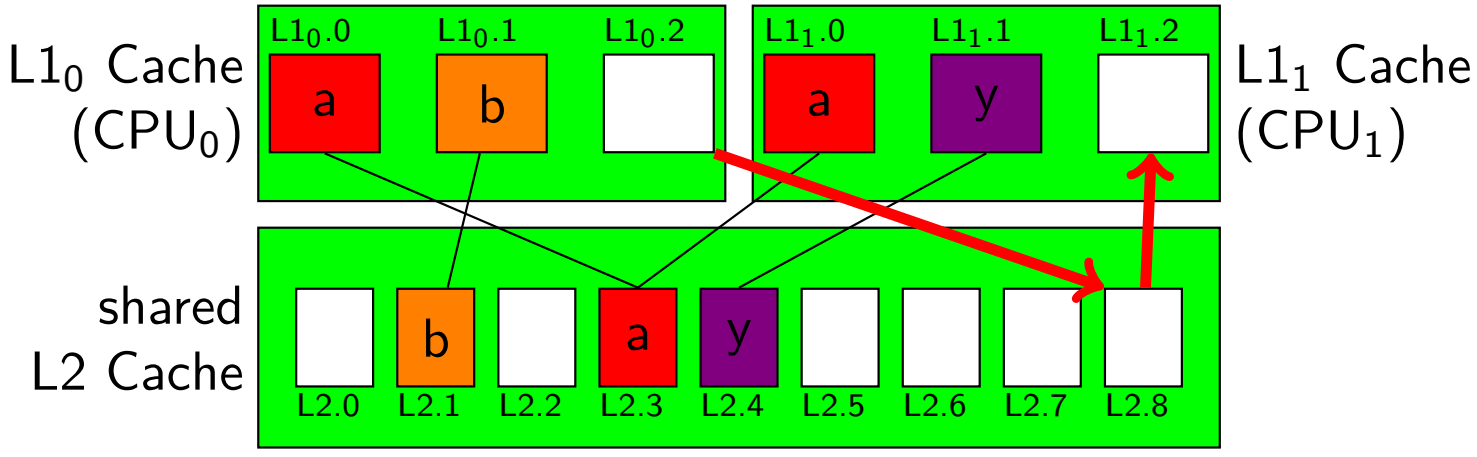
Cache Victims



Intel: Inclusive Cache

- All Elements of L1$ also in L2$
- Cache Miss: Associativity restricts Selection of L2.
- Inclusivity: Eviction from L1 of other Cores possible.
- Measurable Effect (short Loop in Isolation):
  - ➔ Other CPU uses small Buffer: 44 – 64 Cycles
  - ➔ Other CPU uses large Buffer: 44 – 1340 Cycles

Cache Victims



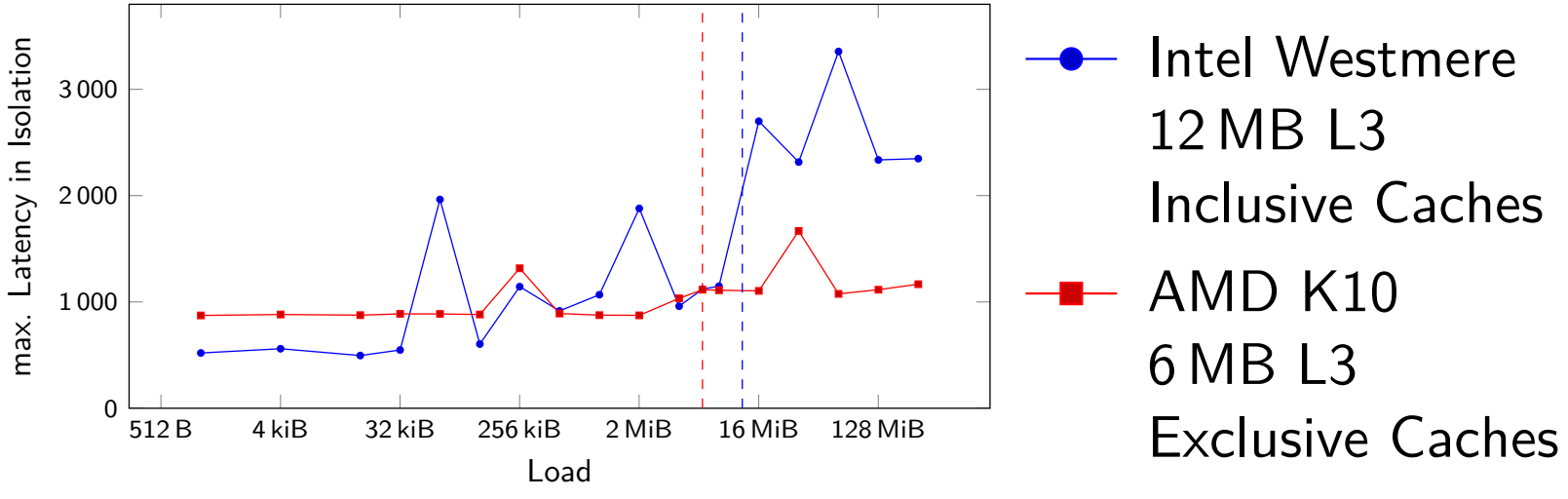
Intel: Inclusive Cache

- All Elements of L1$ also in L2$
- Cache Miss: Associativity restricts Selection of L2.
- Inclusivity: Eviction from L1 of other Cores possible.
- Measurable Effect (short Loop in Isolation):
  - ➔ Other CPU uses small Buffer: 44 – 64 Cycles (Ø: 47 Cycles)
  - ➔ Other CPU uses large Buffer: 44 – 1340 Cycles (Ø: 47 Cycles)

Inclusive vs. Exclusive Cache

Shared L3-Cache

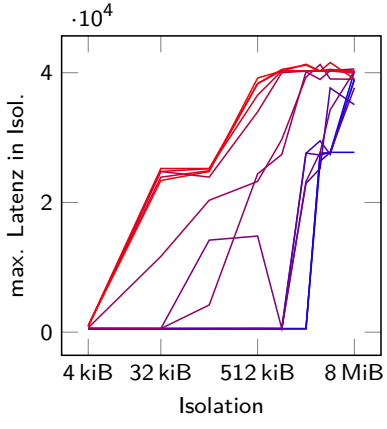
- Difference Inclusive/Exclusive Caching

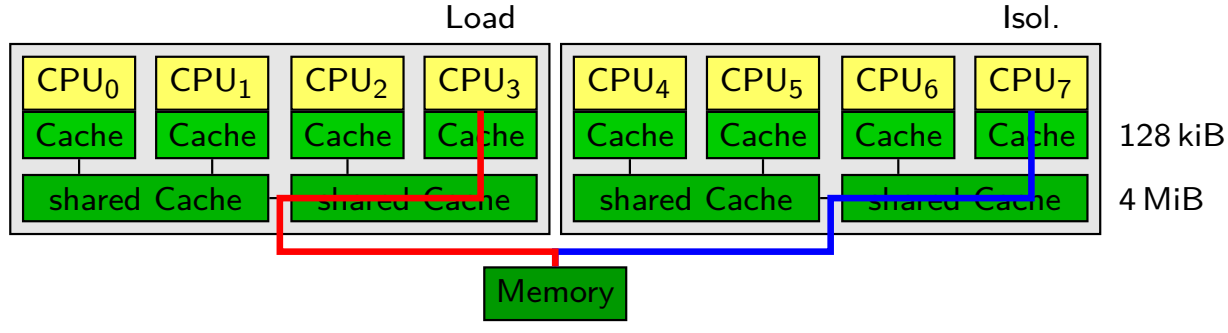


Uniform Memory Access

Architecture

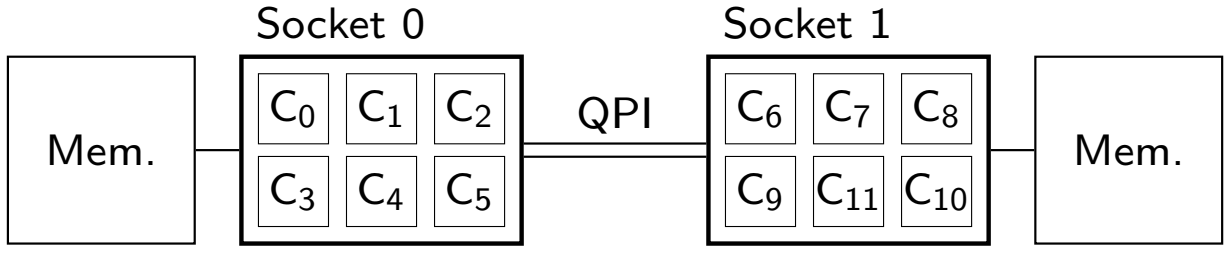
- Intel Core, $2 \times 2 \times 2$ CPUs
- Separate L3-Caches
- Still Influenced by Cache-Coherence Protocol





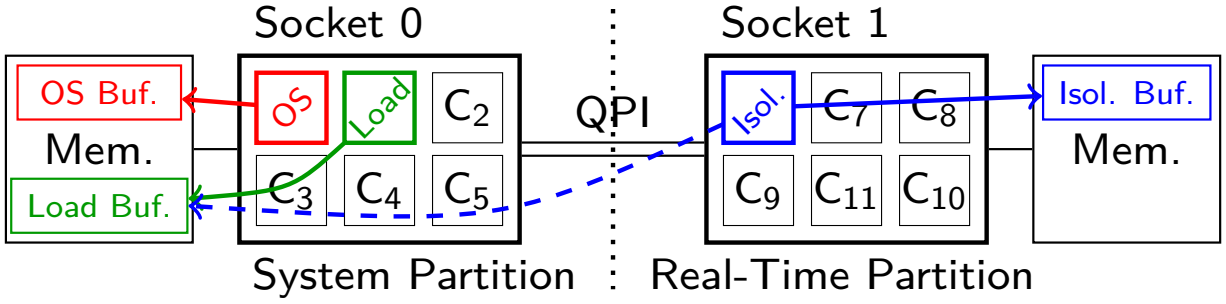
Non-Uniform Memory Access

Architecture



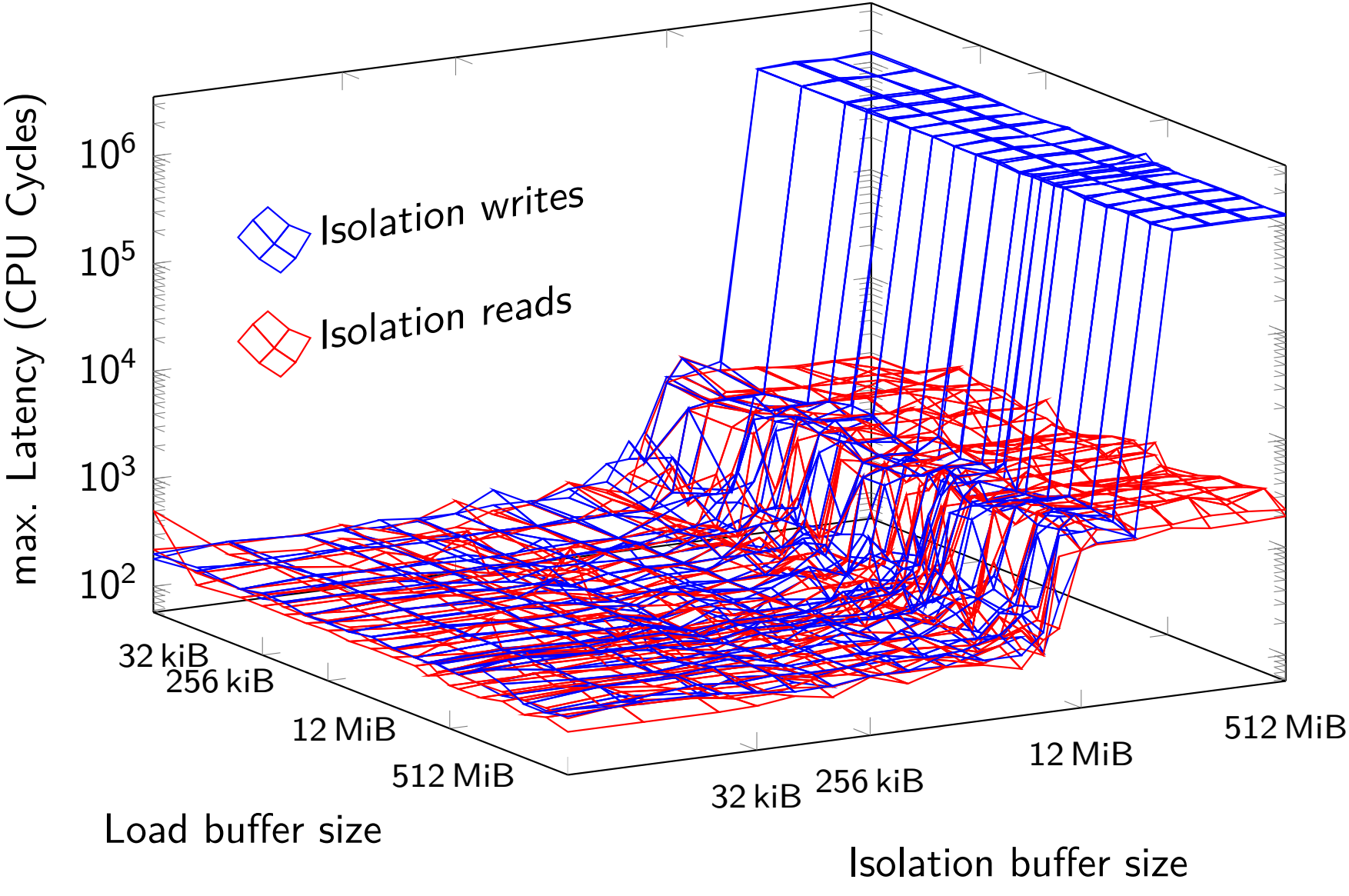
Non-Uniform Memory Access

Architecture

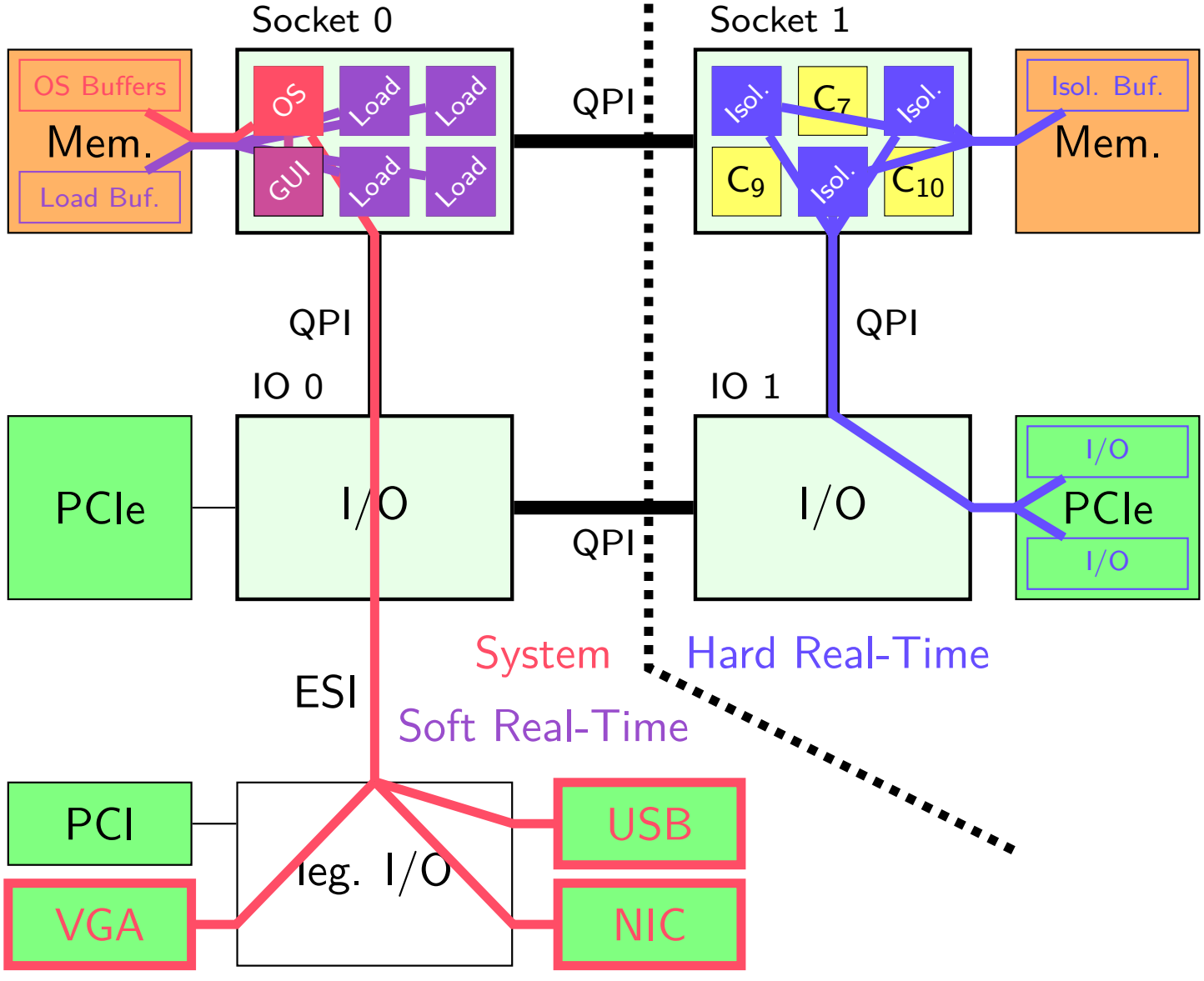


- Partitioning the System by Sockets/NUMA-Nodes

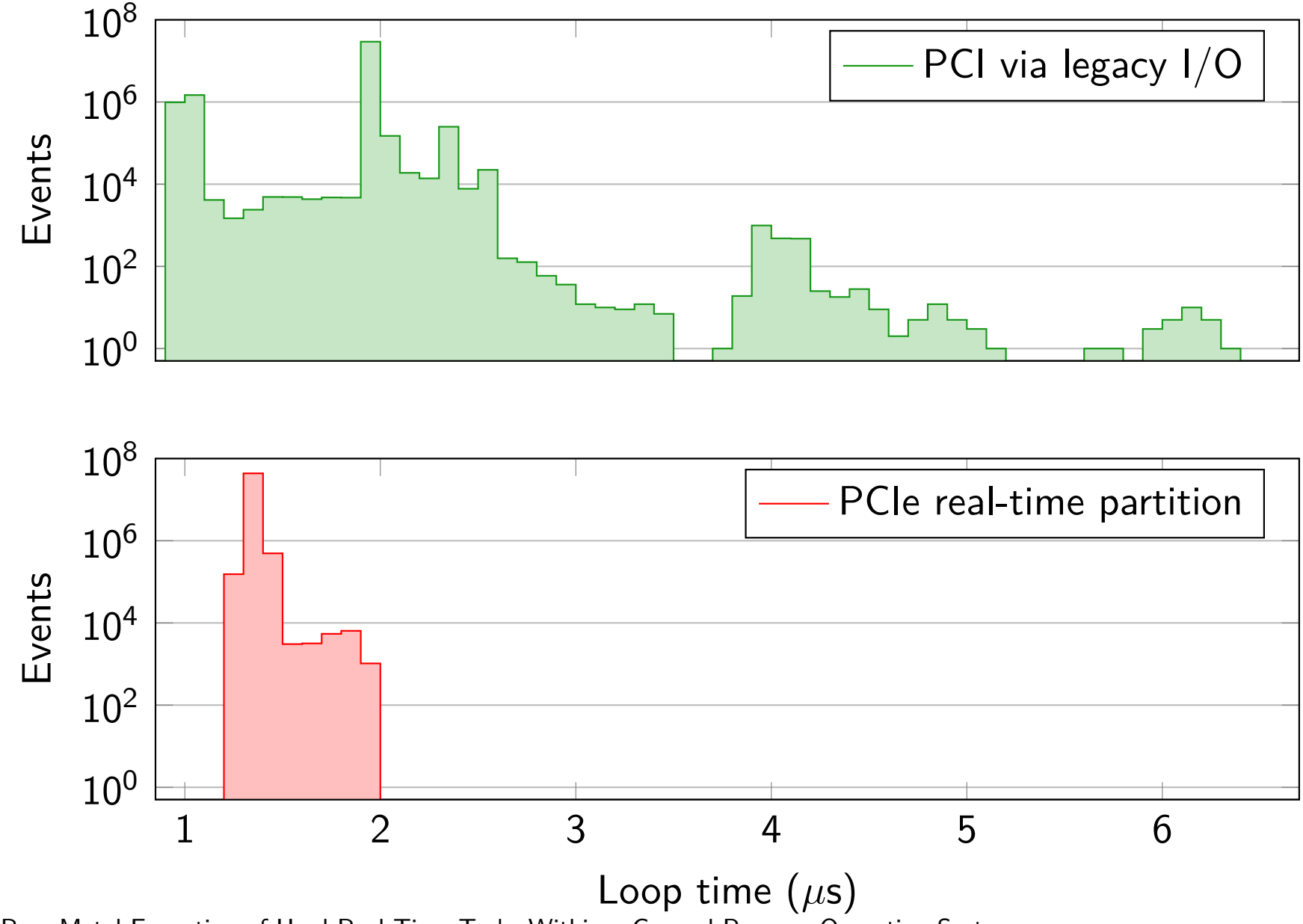
Non-Uniform Memory Access



Non-Uniform I/O
Full Application Example



I/O Latency


PCI via legacy I/O
PCIe real-time partition
Events
10^8
10^6
10^4
10^2
10^0
1
2
3
4
5
6
Loop time (μs)

RWTH AACHEN UNIVERSITY

Motivation

Concept

Implementation

Evaluation

Conclusion

Conclusion

- Full Control of dedicated CPUs
  Other CPUs available for remaining System
- WCET estimation of isolated CPUs (single processor like)
  But still interaction through shared-memory
- Infrastructure of isolated Processes:
  - ➔ System Services replaced with Shared-Memory IPC
  - ➔ HW Access via User-Mode Drivers (IN/OUT and mmap'd-I/O)
  - ➔ Interrupts and Preemptive Scheduling possible
- Arbitrary, flexible Partitioning
- Some Modifications to the Linux Kernel Required
- Transferable to Other Architectures

RWTH AACHEN UNIVERSITY

Thank you for your kind attention!

Georg Wassen and Stefan Lankes – georg.wassen@rwth-aachen.de

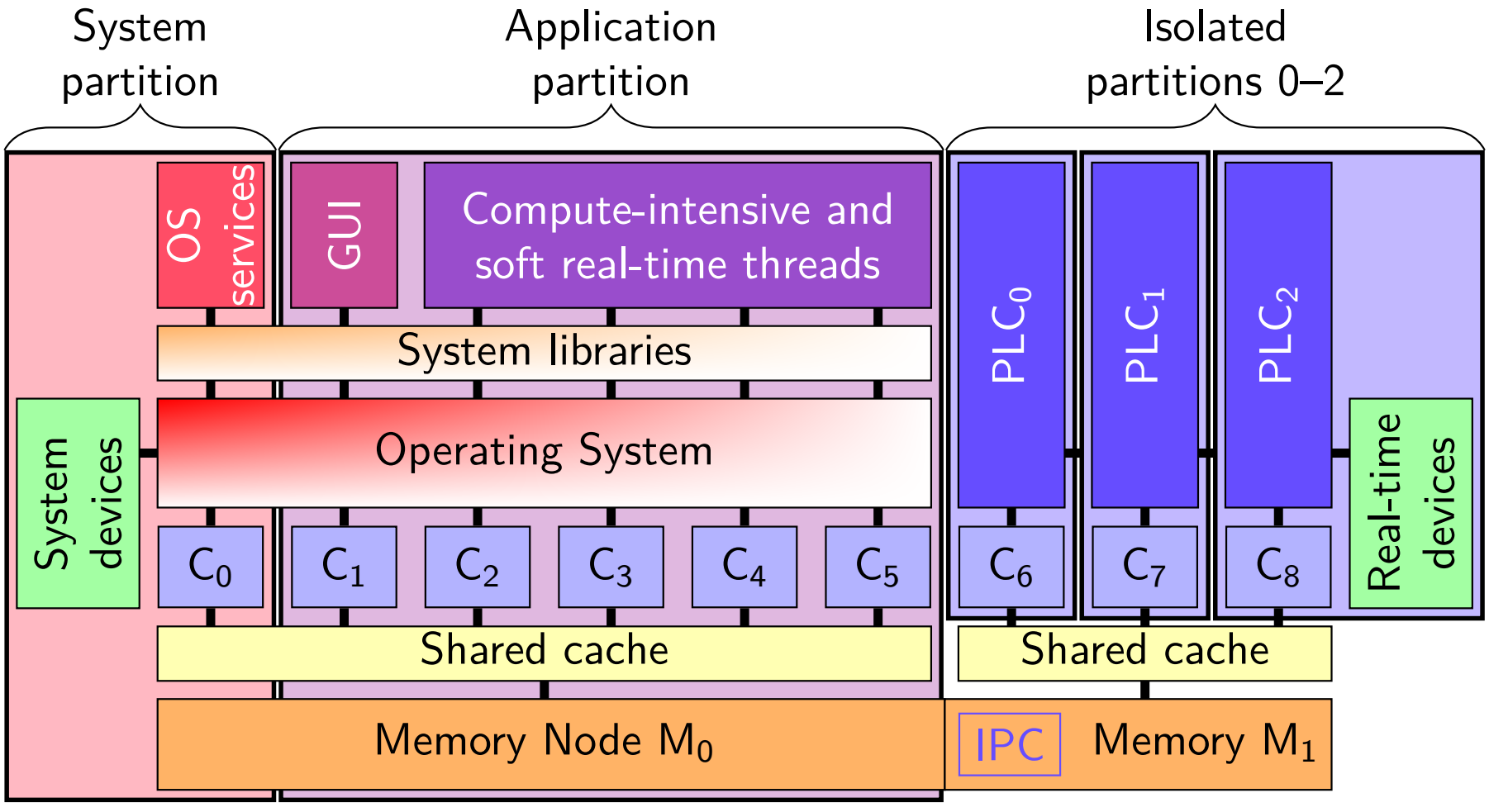
RWTH Aachen University
Templergraben 55
52056 Aachen

www.lfbs.rwth-aachen.de

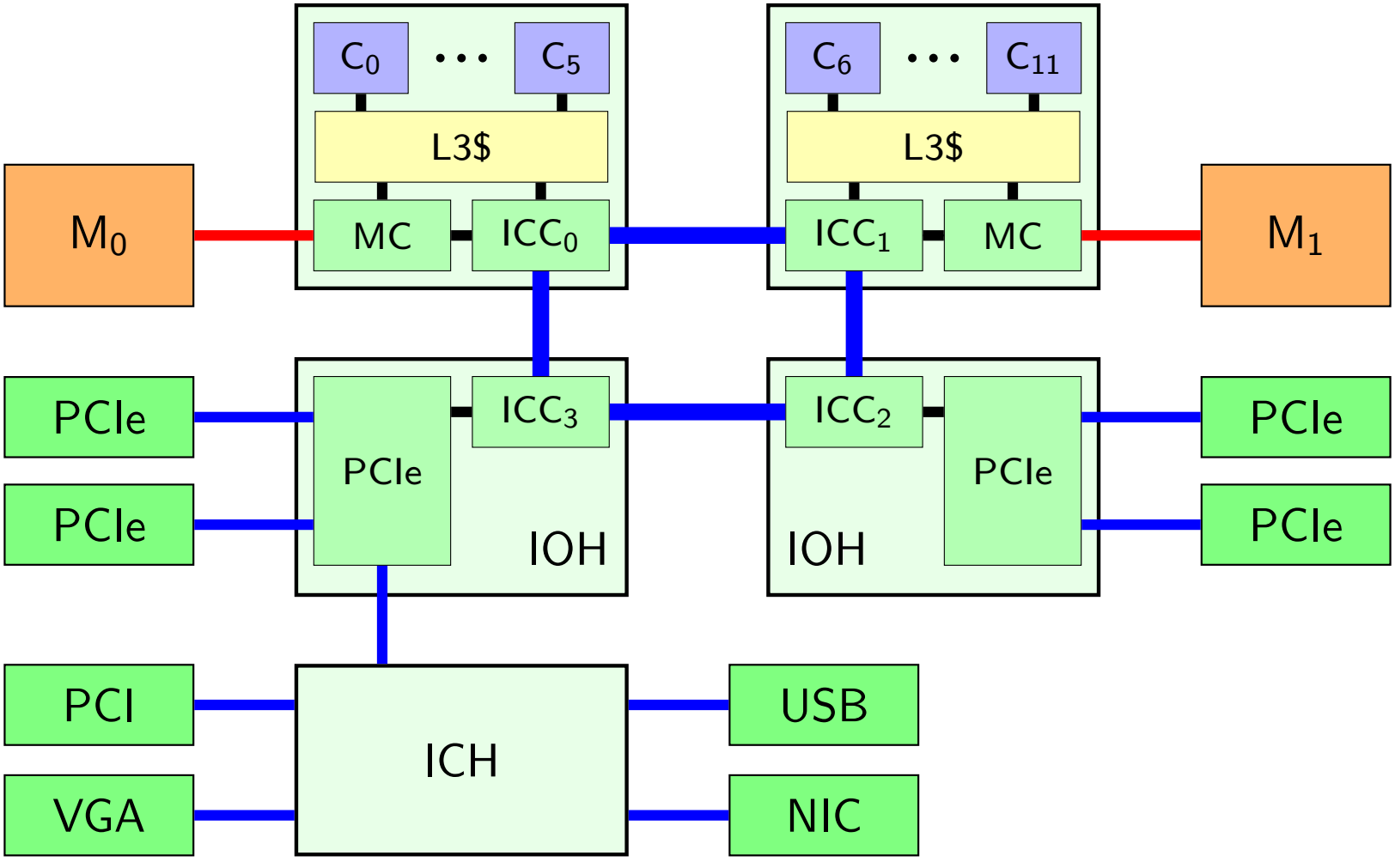
RWTH AACHEN UNIVERSITY

Backup Figures

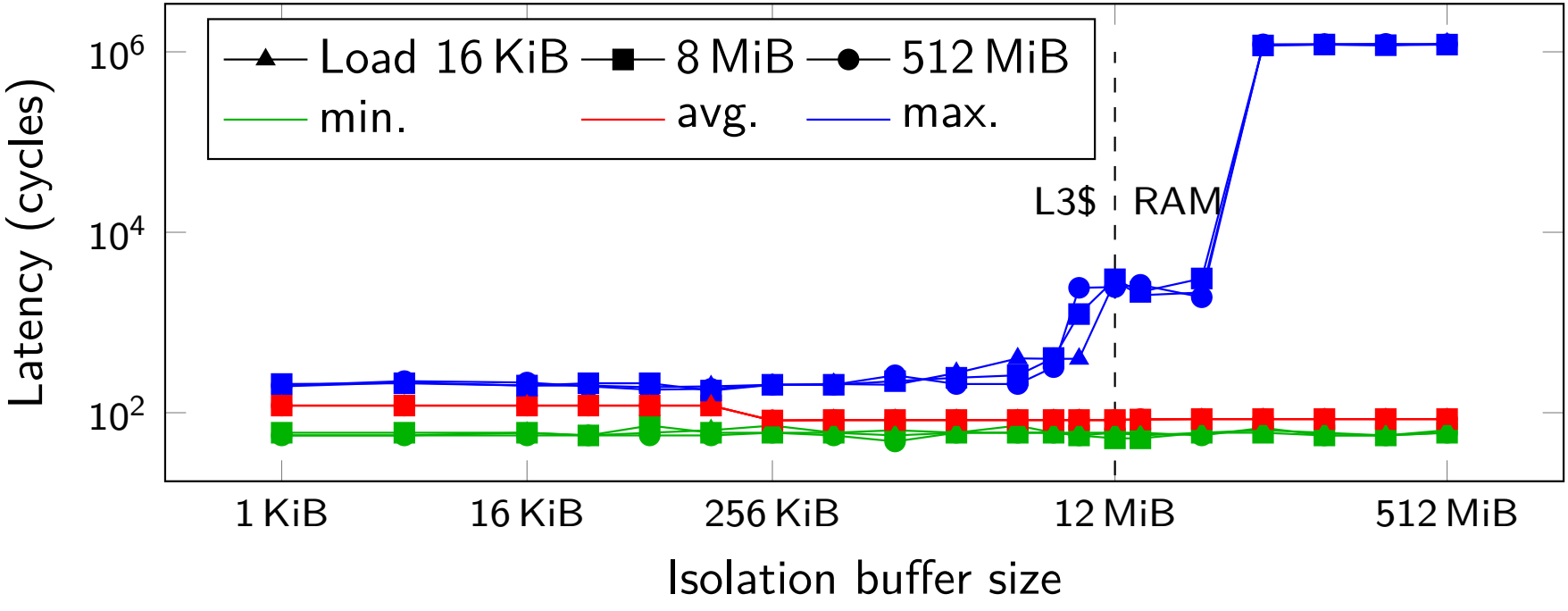
Application Architecture



Mainboard

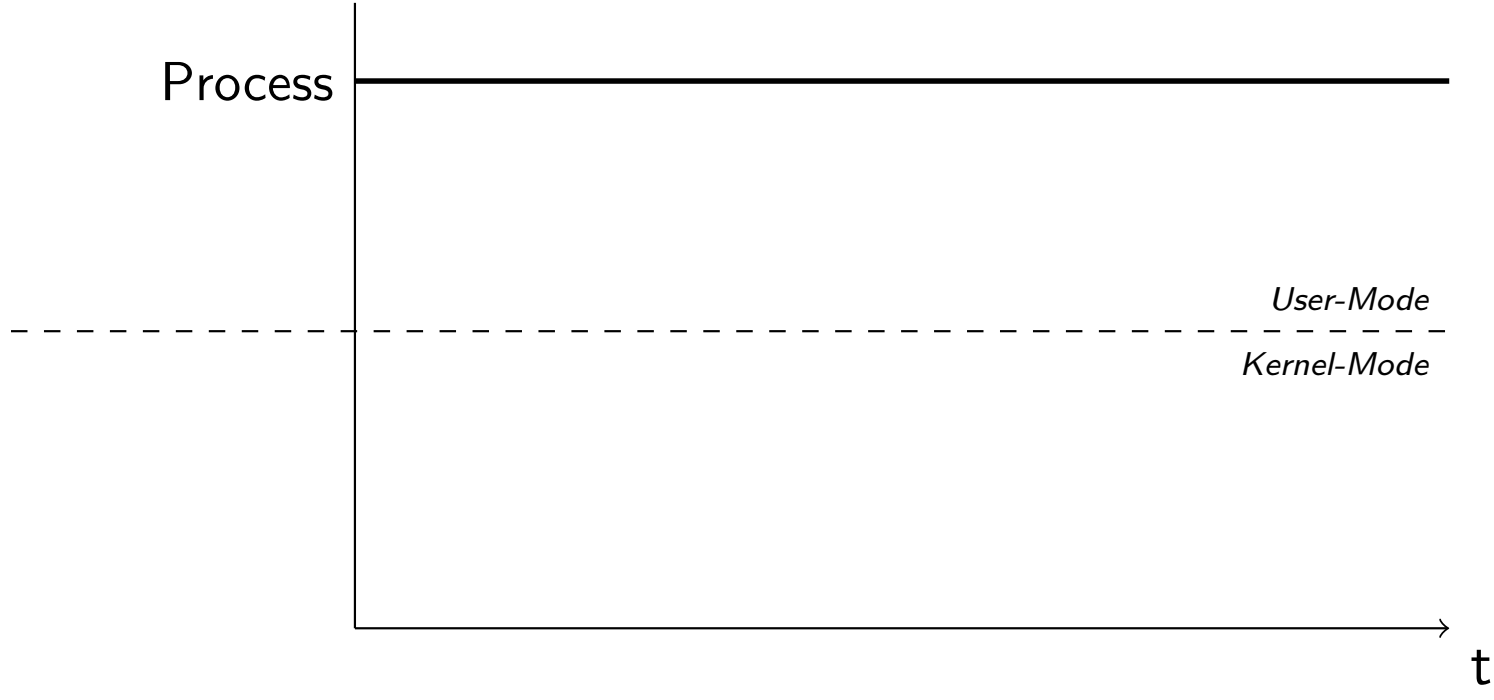

C0
C5
C6
C11
L3$
L3$
M0
MC
ICC0
ICC1
MC
M1
PCIe
PCIe
PCIe
ICC3
ICC2
PCIe
PCIe
PCIe
IOH
IOH
PCI
VGA
ICH
USB
NIC

Jitter on NUMA node



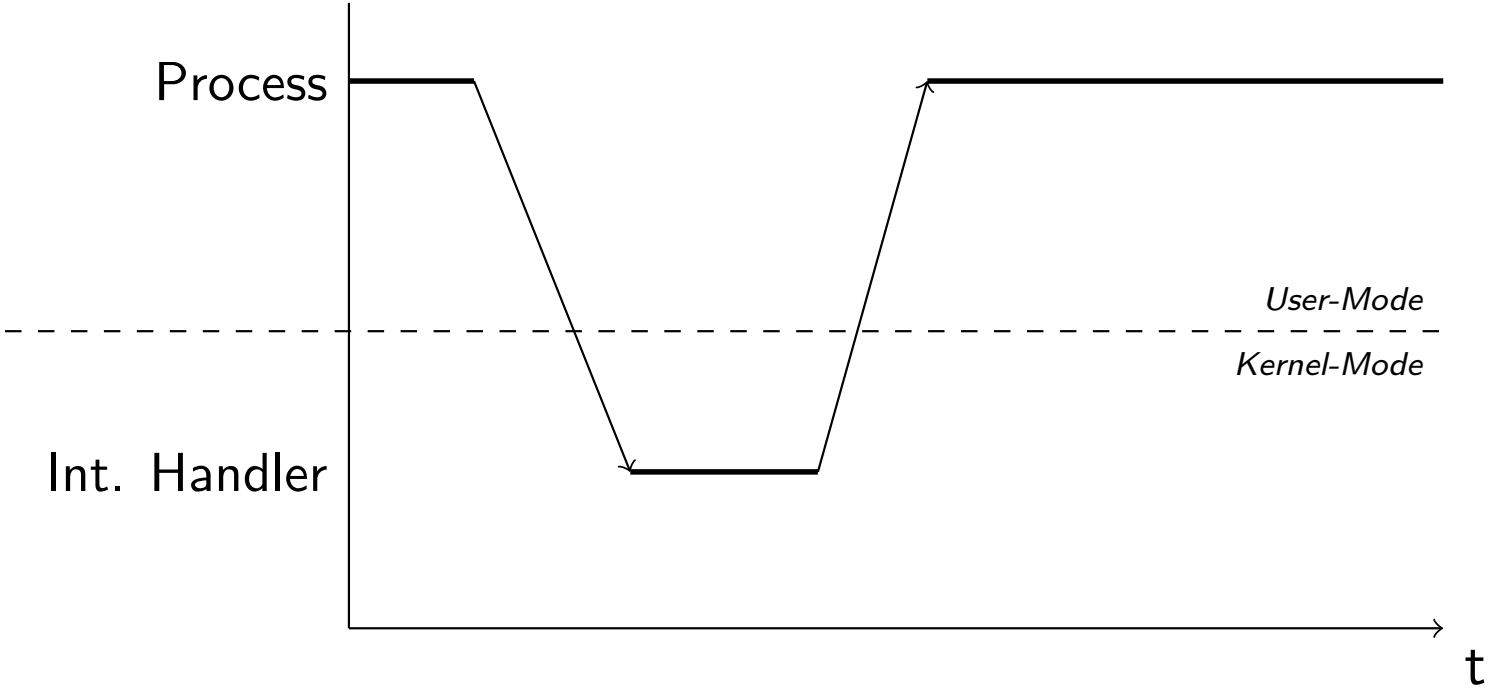
User-Mode Interrupts (UMI)

Requested Interrupts can be handled in User-Mode



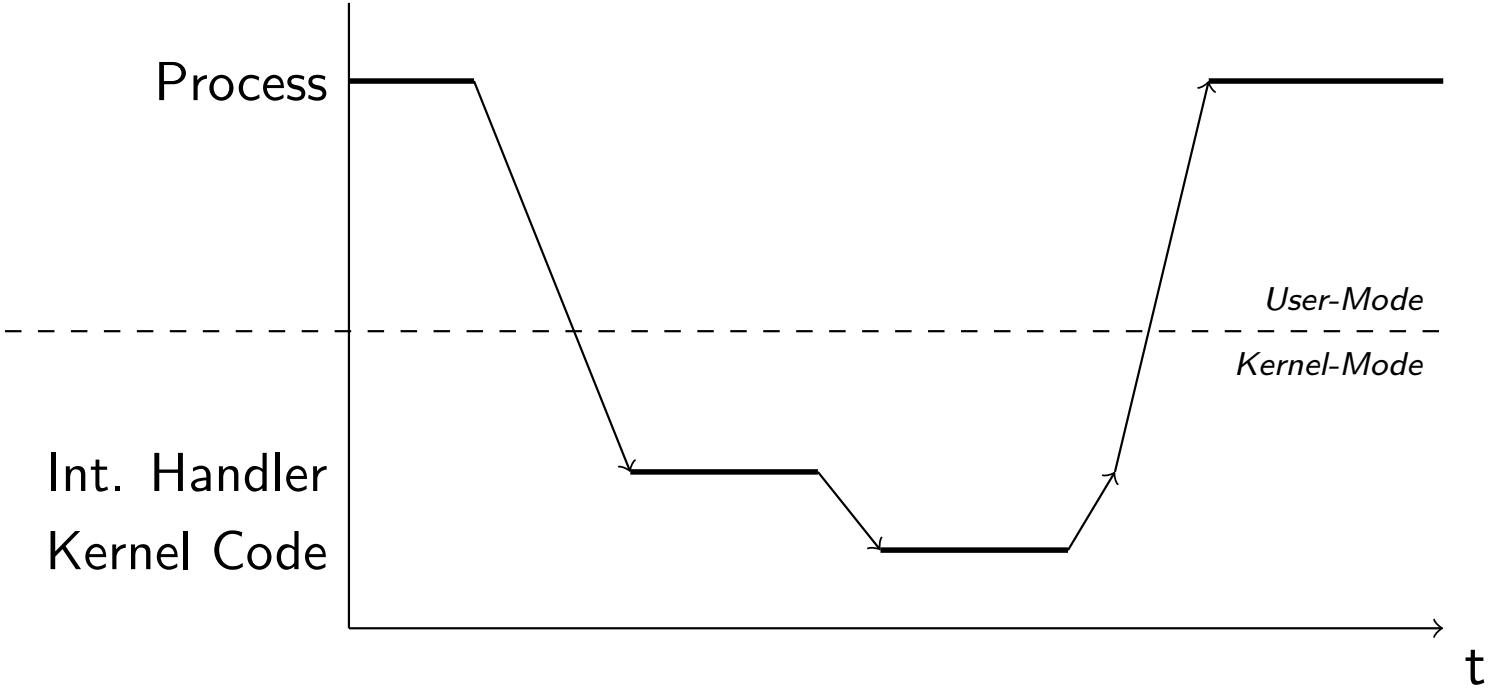
User-Mode Interrupts (UMI)

Requested Interrupts can be handled in User-Mode



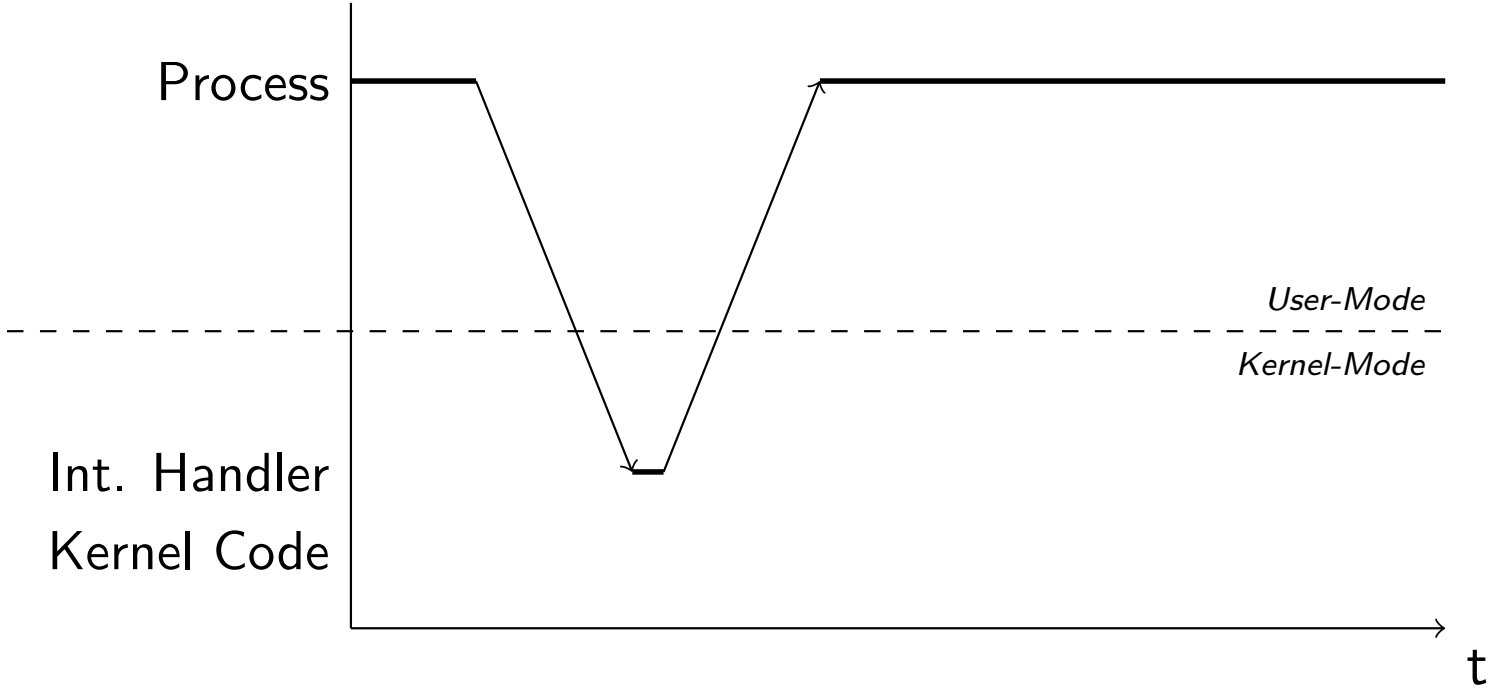
User-Mode Interrupts (UMI)

Requested Interrupts can be handled in User-Mode



User-Mode Interrupts (UMI)

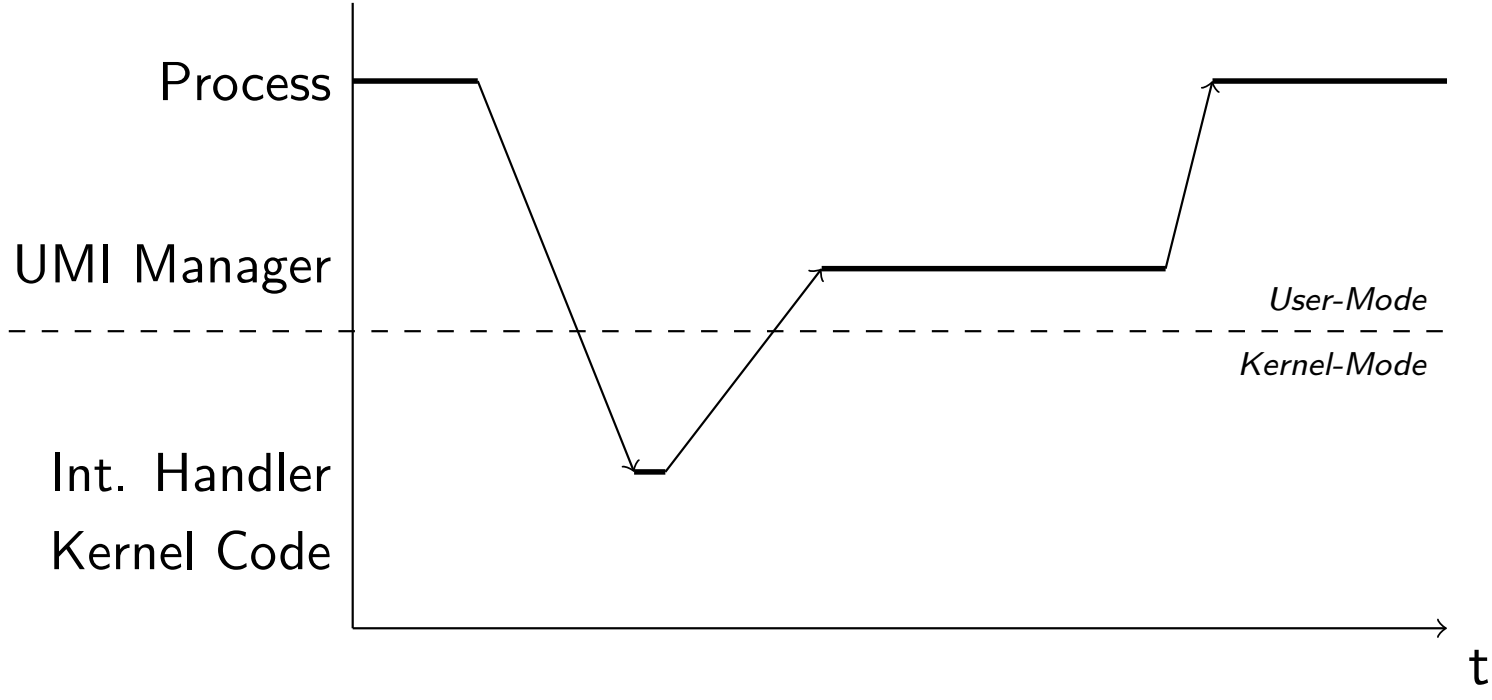
Requested Interrupts can be handled in User-Mode



- Own Interrupt Handler: Avoids Unknown Kernel Code

User-Mode Interrupts (UMI)

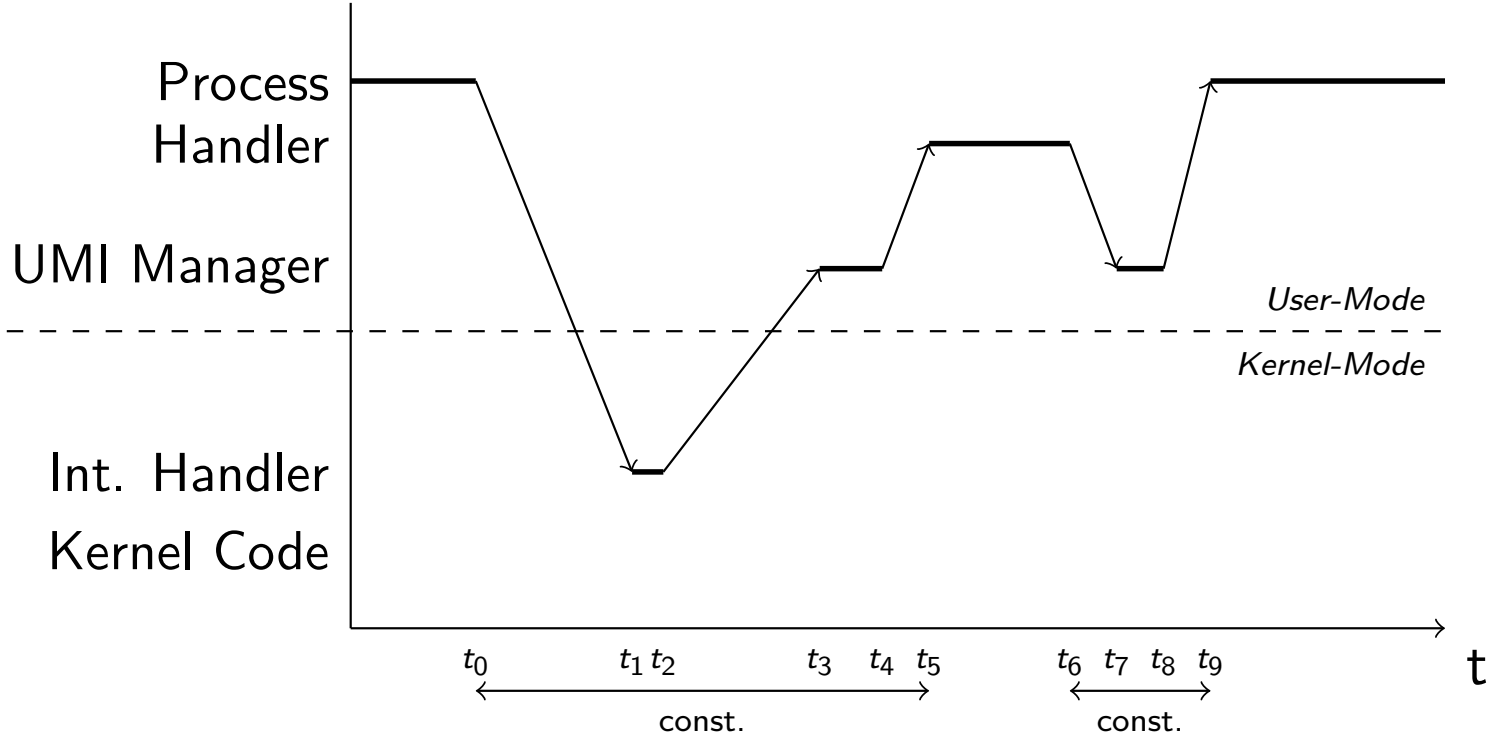
Requested Interrupts can be handled in User-Mode



- Own Interrupt Handler: Avoids Unknown Kernel Code

User-Mode Interrupts (UMI)

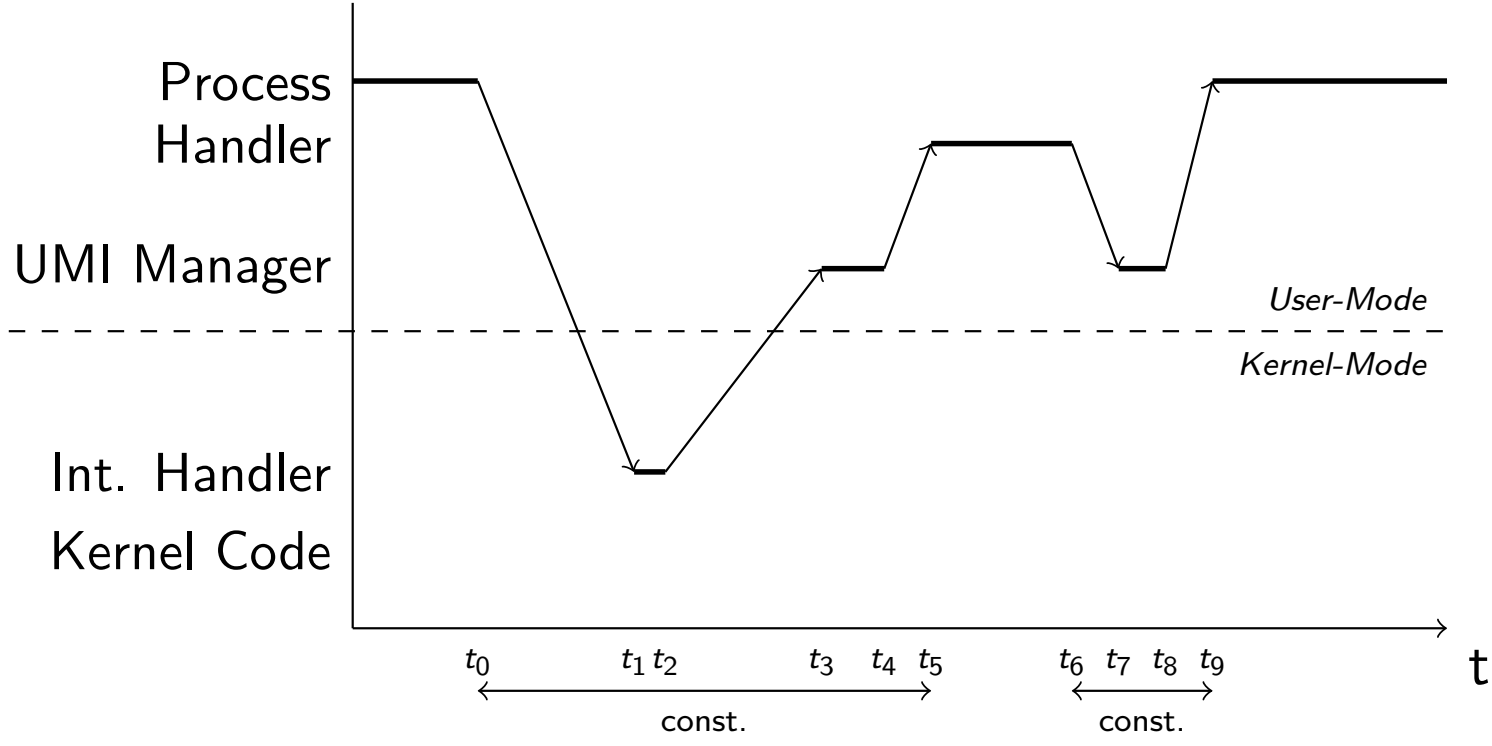
Requested Interrupts can be handled in User-Mode



- Own Interrupt Handler: Avoids Unknown Kernel Code
- Overhead ($t_5 - t_0 + t_9 - t_6$): ca. 0.5 μs (const!)

User-Mode Interrupts (UMI)

Requested Interrupts can be handled in User-Mode



- Own Interrupt Handler: Avoids Unknown Kernel Code
- Overhead ($t_5 - t_0 + t_9 - t_6$): ca. 0.5 μs (const!)
- Realized as Kernel Module

User-Mode Int. Code Example

```
void handler(void) {
  char c = IN(0x378);       /* read from I/O Port */
  printf("%c\n", c);        /* process */
}

void main (void) {
  usi_register(0x71, handler);
  sleep(10);
  usi_restore(0x71);
}
```

Porting an Embedded RTOS

RWTH AACHEN UNIVERSITY

- One Isolated Process per CPU
- Current Paradigm: Time-Triggered
- Goal: Preemptive User-Mode Threads
  - ➔ Scheduling of Multiple Tasks
  - ➔ Within a UNIX Process
  - ➔ Without Help/Interaction of the Operating System
- Trigger: Timer Interrupt handled in User-Mode

UMI Scheduler Code Example

```
void task1(void) {
  int i;
  for (i=0; i<1000; i++) {
    // ...
  }
  scheduler_stop();
}
void task2(void);          /* similar */

void main (void) {
  create_task(task1);
  create_task(task2);
  scheduler_start();
  /* returns after a task calls scheduler_stop() */
}
```

RWTH AACHEN UNIVERSITY

Thank you for your kind attention!

Georg Wassen and Stefan Lankes – georg.wassen@rwth-aachen.de

RWTH Aachen University
Templergraben 55
52056 Aachen

www.lfbs.rwth-aachen.de