Berkeley Architecture Research

AuRORA Introduction

Seah Kim
UC Berkeley
seah@berkeley.edu

CHIPYARD

GEMMINI

AuRORA: Virtualized Accelerator Orchestration for Multi-Tenant Workloads
Selected as One of "Top Picks from Computer Architecture Conferences"
Seah Kim, Jerry Zhao, Krste Asanovic, Borivoje Nikolic, Yakun Sophia Shao
International Symposium on Microarchitecture (MICRO), October 2023.

Pre-Fetch RTL Build

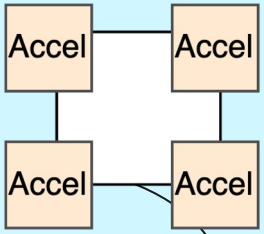
```
cd /root/chipyard/generators/sim/verilator

make CONFIG=TutorialGemminiReRoCCConfig
```

Background: Trends in Modern SoCs

More accele

- More compute-bo
  workloads require
  acceleration

How to architect many-accelerator SoCs?

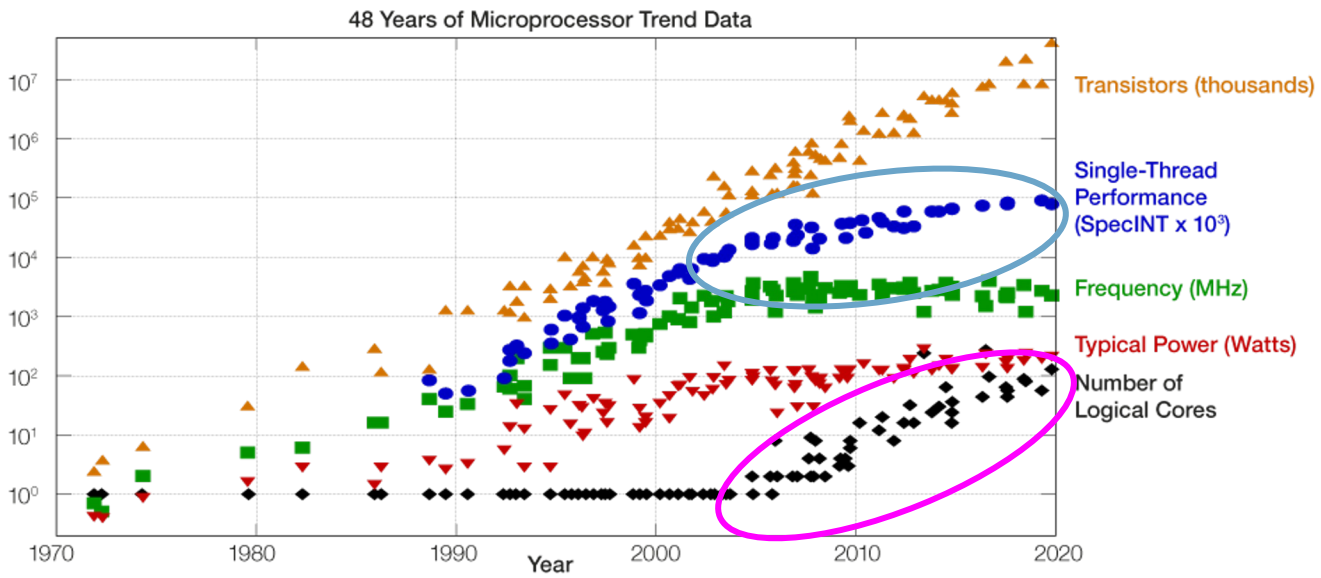


e applications

- Software stacks grow in complexity
- Graphics/multimedia/AI are pervasive

Trends in Modern SoCs: Multi-Core

End of single-thread performance scale -> multi-core architecture



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

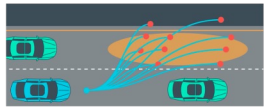
Trends in Modern SoCs: More Applications

Multiple tasks share system resources

Detections

Tracking

Motion Prediction

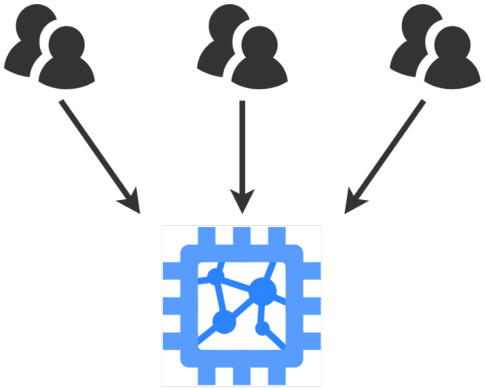


Trajectory Planning

Autonomous Vehicle

@ Edge

Clients

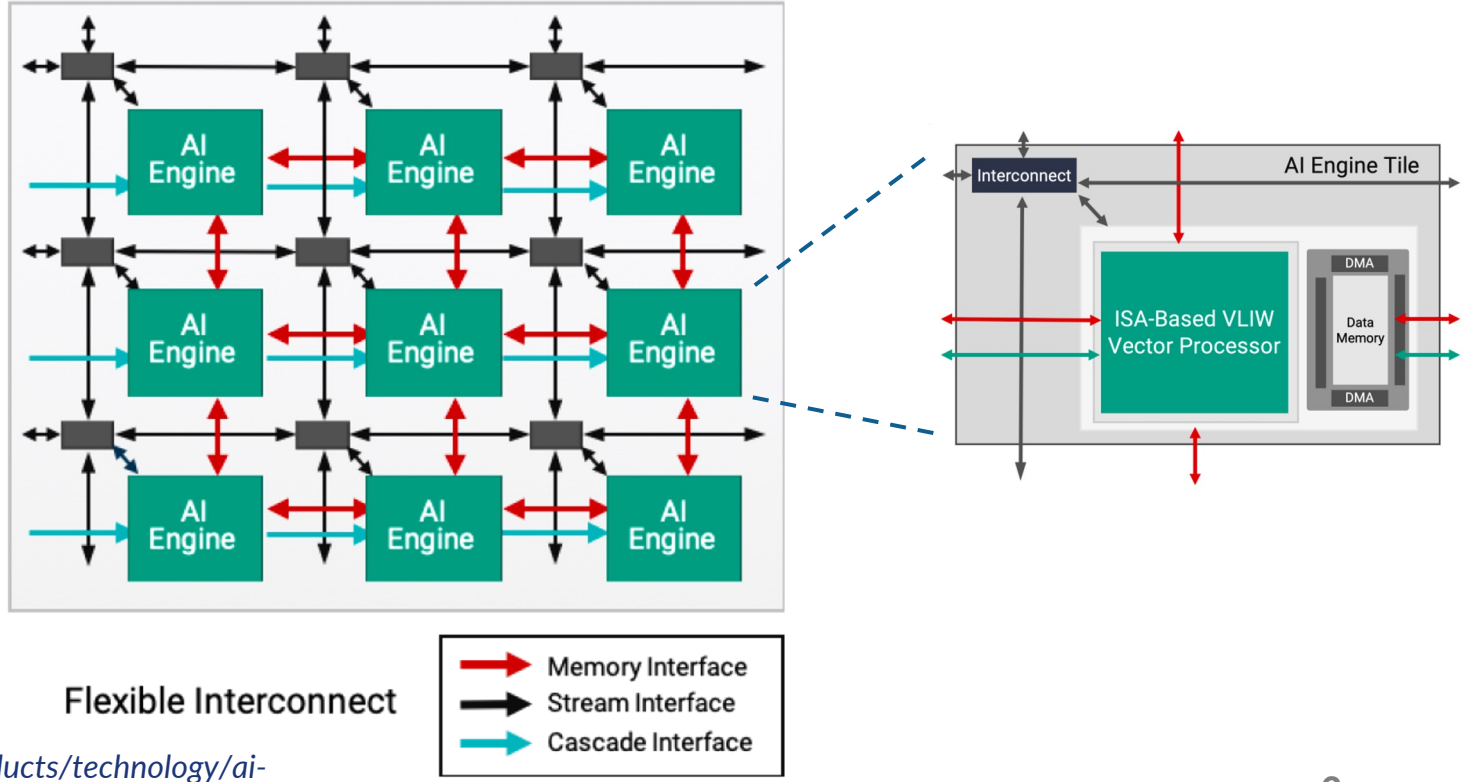


Cloud Accelerator

@ Cloud

Trends in Modern SoCs: Multi-Accelerator

To keep up with application that are becoming more demanding…



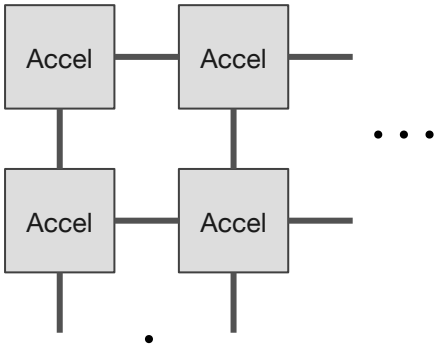
AMD AI Engine Technology (https://www.xilinx.com/products/technology/ai-engine.html)

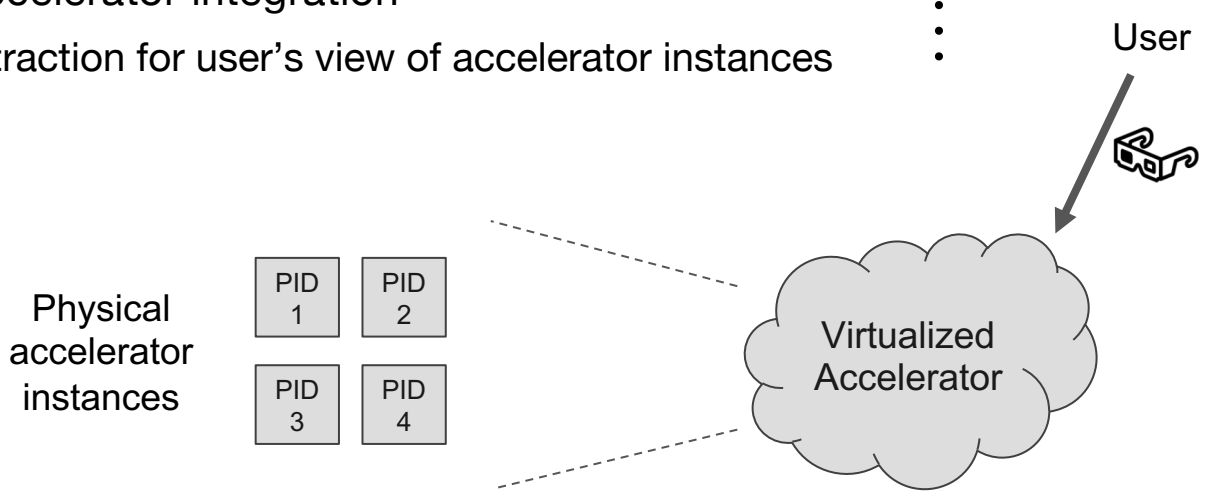
Requirements for Accelerator Integration

Goal 1: Scalable to many-accelerator systems

Requirements:

- Physical scalability
  - Many-accelerator integration
- Virtual accelerator integration
  - Abstraction for user's view of accelerator instances
  - ???





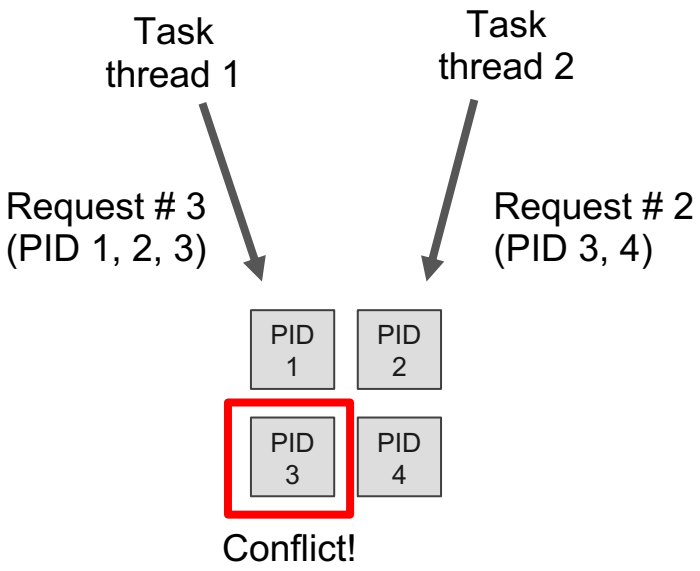
Physical Accelerator Integration

Program physical accelerator resources

- Request accelerator using physical ID (PID)

Issues under multi-tenancy

- Programming burden
- Resource conflict
  - Hard to repartition resource frequently
  - Cause stall: Low utilization



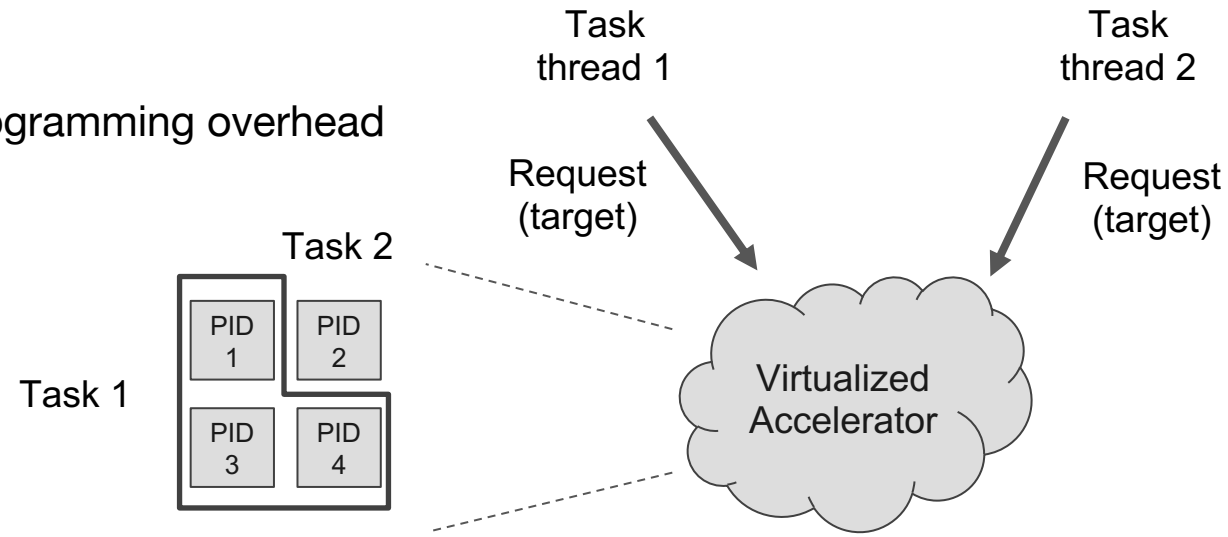
Virtual Accelerator Integration

Provides abstraction between user's view of accelerator and accelerator instances

Enable scalable many-accelerator for multi-tenancy

Requirements …

- Low latency
- Minimize programming overhead

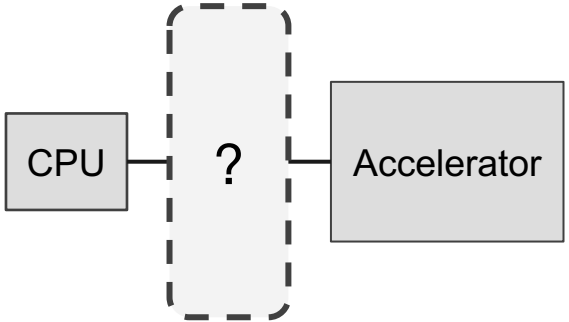


Requirements for Accelerator Integration

Goal: Enable scalable **many-accelerator** for **multi-tenant execution**

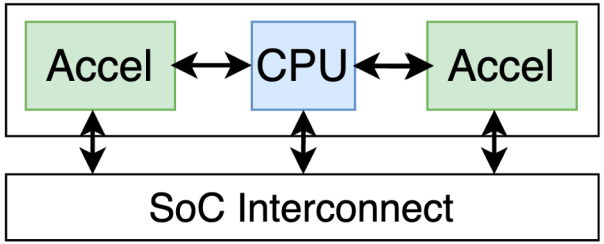
Requirements:

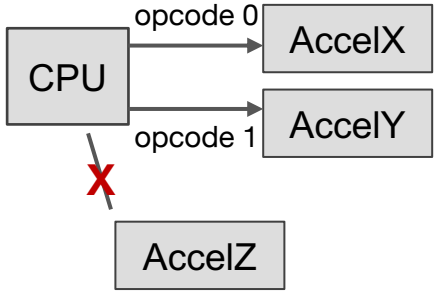
- Physical scalability
- Virtual accelerator integration
  - Low latency
  - Minimal programming overhead



Interface: How accelerator interacts with host CPU and system

Existing Physical Accelerator Integration Methodologies



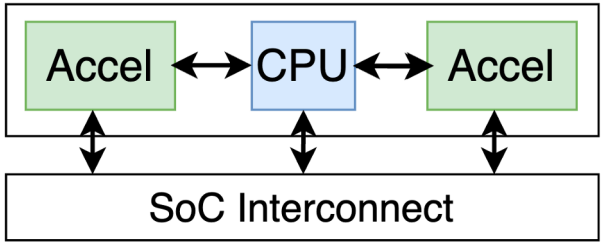


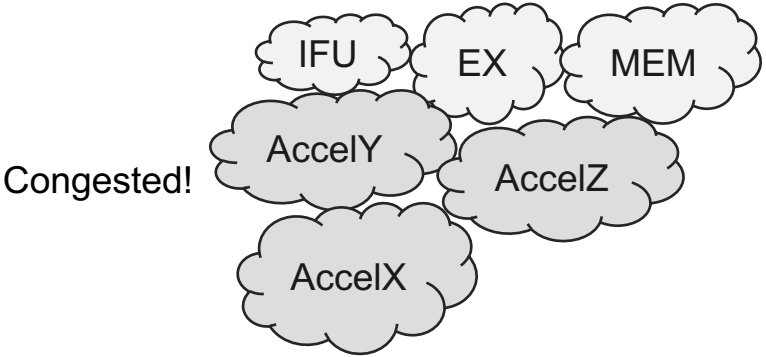
Tightly CPU-coupled:

Limited opcode space → Limited accelerator per core

Physically attached to CPU

Existing Physical Accelerator Integration Methodologies





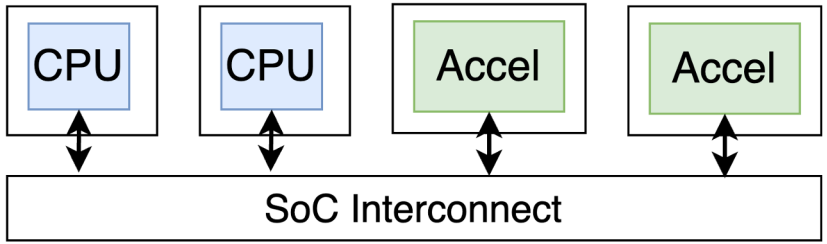
Tightly CPU-coupled:

Limited opcode space → Limited accelerator per core

Physically attached to CPU → Physical design challenge

Scalability issue

Existing Physical Accelerator Integration Methodologies



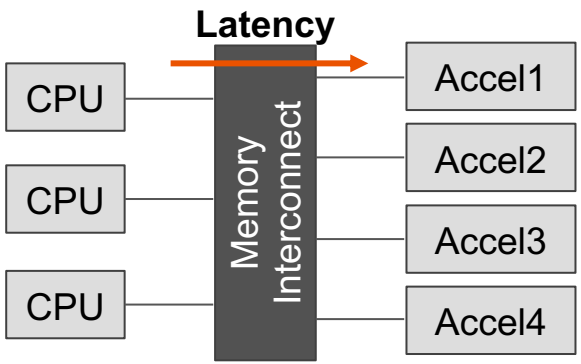
Memory-mapped over interconnect

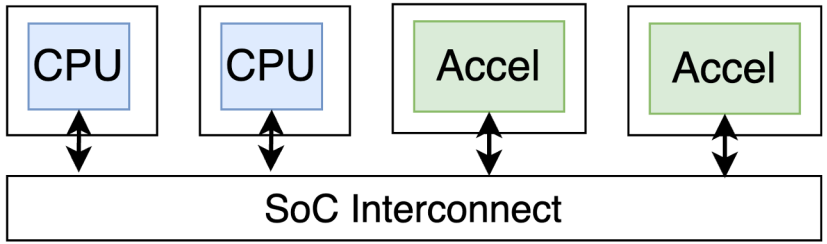
Software complexity ←

Setup accelerator IOMMU for address translation

Access accelerator over system bus, memory hierarchy

Existing Physical Accelerator Integration Methodologies





Memory-mapped over interconnect

Software complexity ← Setup accelerator IOMMU for address translation

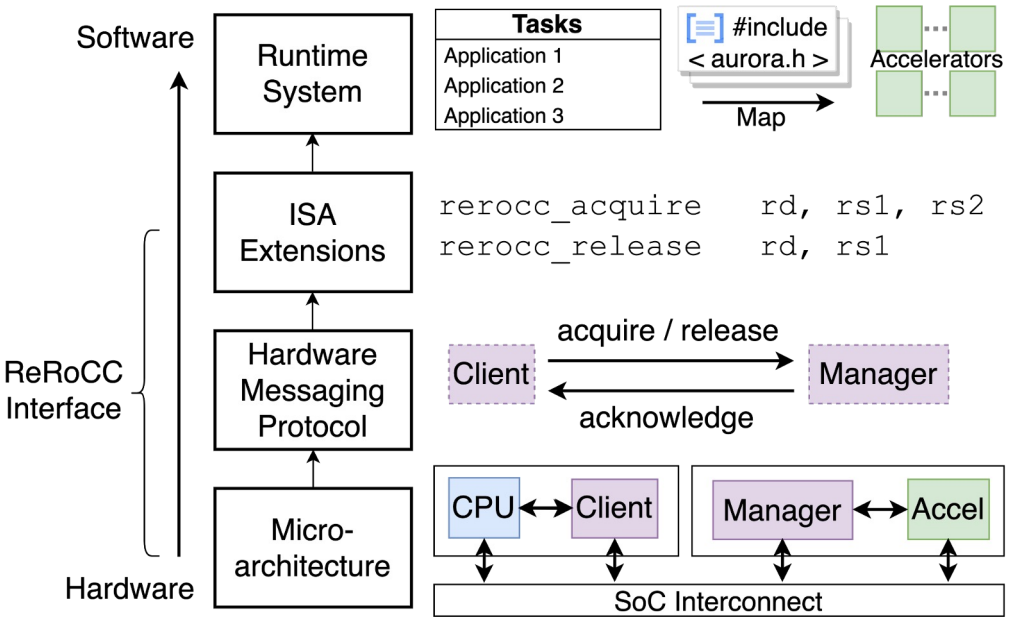
Latency overhead ← Access accelerator over system bus, memory hierarchy

Virtual integration difficulties

AuRORA: Virtual Accelerator Integration and Orchestration

A **full-stack system** enabling **scalable deployment** and **virtualized integration** of accelerators

✓: Virtualization
♣: Scalability



✓ Virtualized accelerator management

♣ Enable acquiring many-accelerators
✓ Enable programmable virtual interface

✓ Low latency
✓ Enable virtual to physical mapping

♣ Enable physical disaggregation
✓ Provides illusion of tight-coupling

AuRORA and ReRoCC

AuRORA: A ISA-agnostic full-stack methodology for accelerator integration

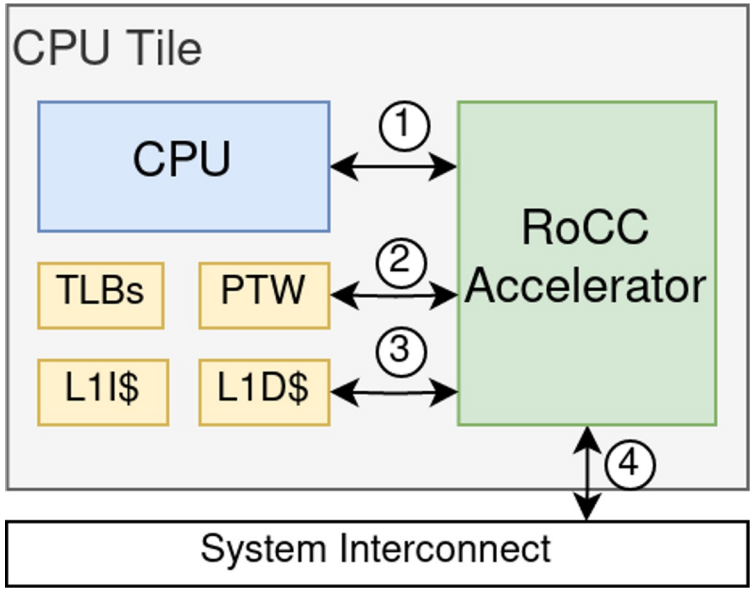
ReRoCC: An implementation of **AuRORA** targeting the existing RoCC interface for RISC-V accelerators

Tight-coupled example: RoCC

- RoCC interface helps attach accelerators to Rocket CPU
- RoCC accelerator follows standard instruction format
  - 4 opcodes for non-standard instructions
  - 2 source registers, 1 destination register can be passed to the accelerator

31 – 25	24 – 20	19 – 15	14	13	12	11 – 7	6 – 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7
roccinst[6:0]	src2	src1				dest	*custom-0/1/2/3*

Background on Tightly-coupled RoCC Interface



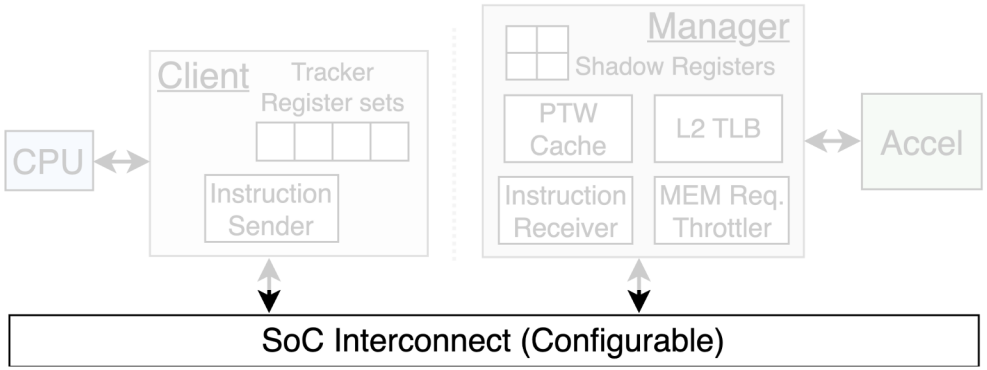
1. **CMD/RESP interface:** CPU issues custom instructions to accelerator
2. **PTW access:** Accelerator can access host PTW/TLBs for virtual memory
3. **L1D$ access:** shared data cache
4. **Bus access:** direct access to coherent/incoherent memory

Comparable to commercial core-IP custom extension interfaces

Flexible interface with many existing open-source accelerator implementations/resources

Strategy: Build off/improve RoCC, retain backwards compatibility with existing work

AuRORA Microarchitectural Components



Client:

Attaches to CPUs via RoCC

Forwards accelerator instructions to acquired manager

Manager:

Attaches to existing RoCC accelerators

Shadow thread architectural state

Eliminate need of user-/supervisor-managed IOMMU

AuRORA Hardware Messaging Protocol

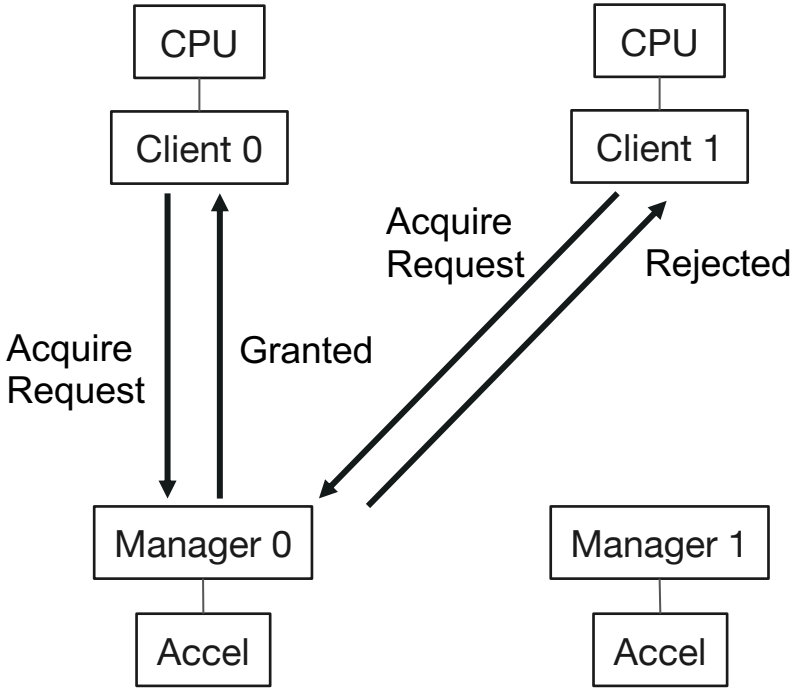
Manages communication between Clients and Managers

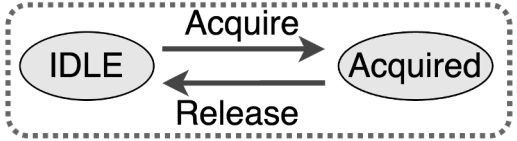
Non-blocking

Protocol supports:

- Client-manager synchronization
- Maintenance of shadowed architectural state on managers
- Client-to-manager instruction forwarding

Physical transport layer: network-on-chip interconnect





Manager states

AuRORA Hardware Messaging Protocol

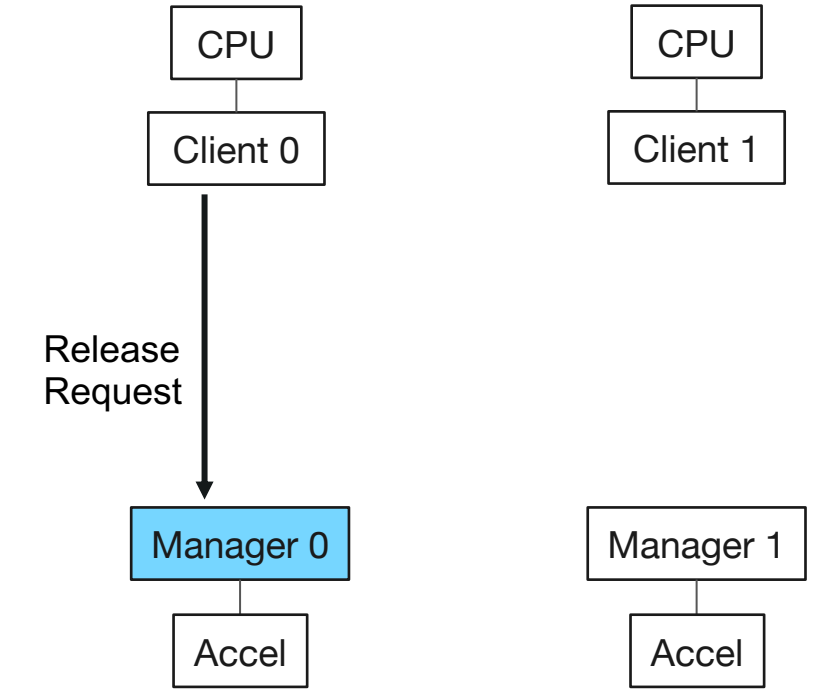
Manages communication between Clients and Managers

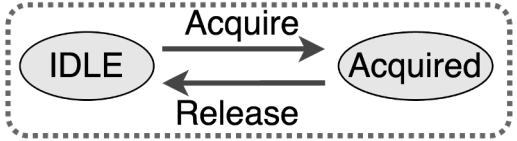
Non-blocking

Protocol supports:

- Client-manager synchronization
- Maintenance of shadowed architectural state on managers
- Client-to-manager instruction forwarding

Physical transport layer: network-on-chip interconnect





Manager states

AuRORA ISA Extensions

AuRORA PseudoInst.
`rr_acquire`
`rr_assign`
`rr_release`
`rr_fence`

Allows user thread to interact with HW in programmable fashion

Low-overhead: bounded by interconnect latency

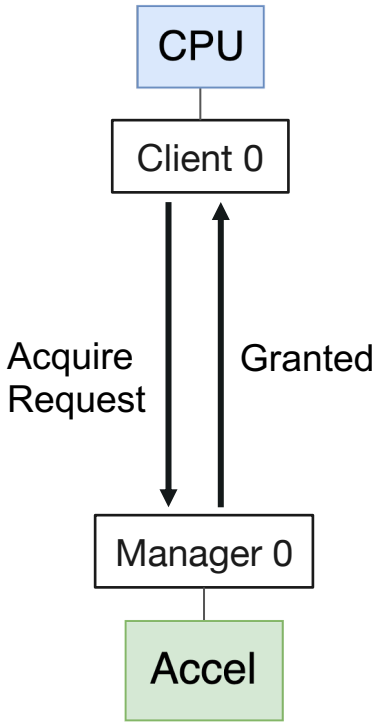
AuRORA ISA Extensions

rr_acquire

- Maps physical accelerator to virtual accelerator index
- Return success status

rr_assign

- Maps virtual accelerator to available opcode on its architectural thread
- Allows an architectural thread to acquire more accelerators than the available opcode space



AuRORA Runtime

Backwards compatibility with accelerator SW

- Invoked only before entry of DNN layer execution

Low overhead

- Implements in user-space
- No need to preempt during layer execution

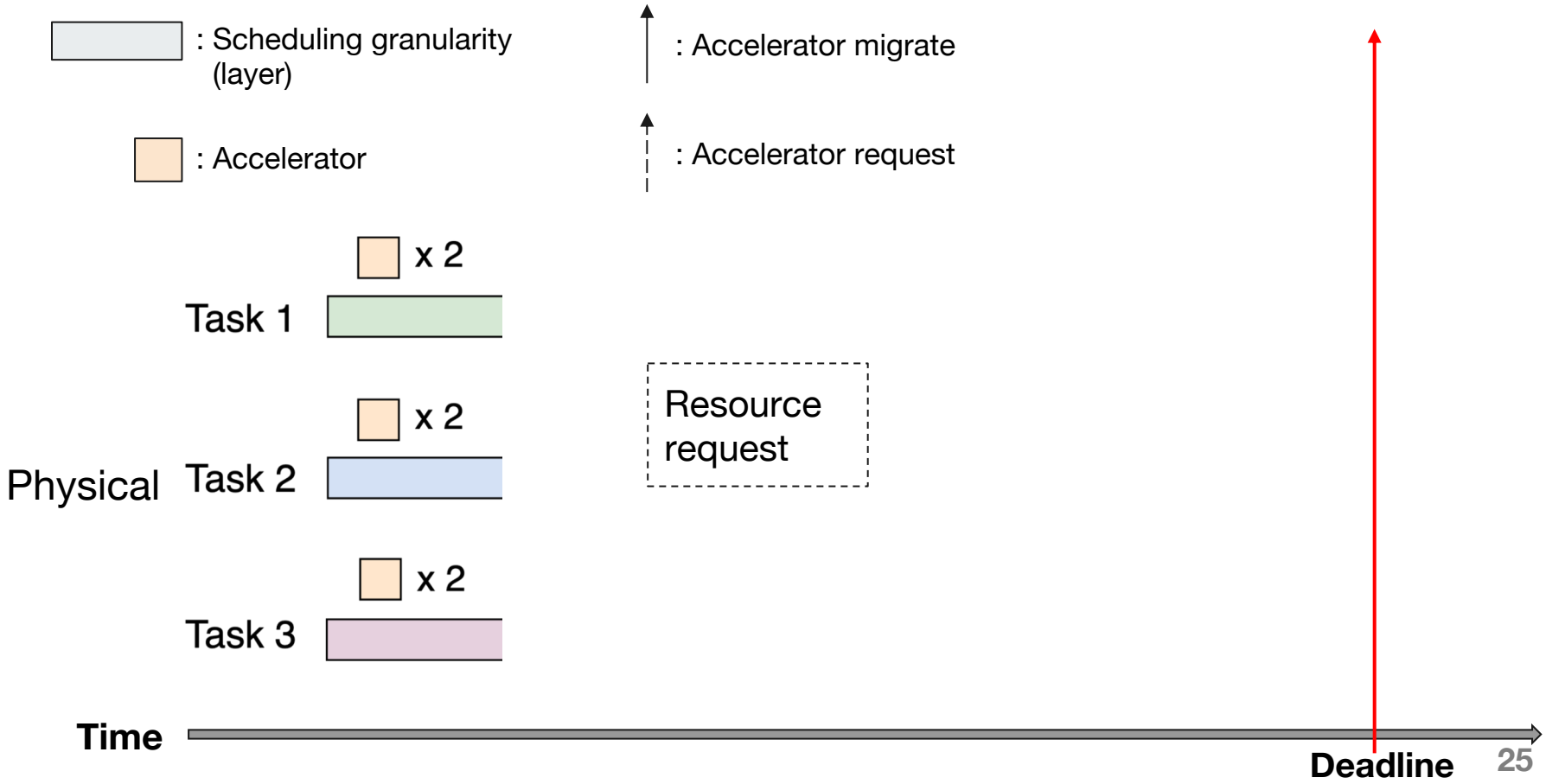
Dynamically allocate resources for multi-tenant workload

- Latency target-aware resource allocation

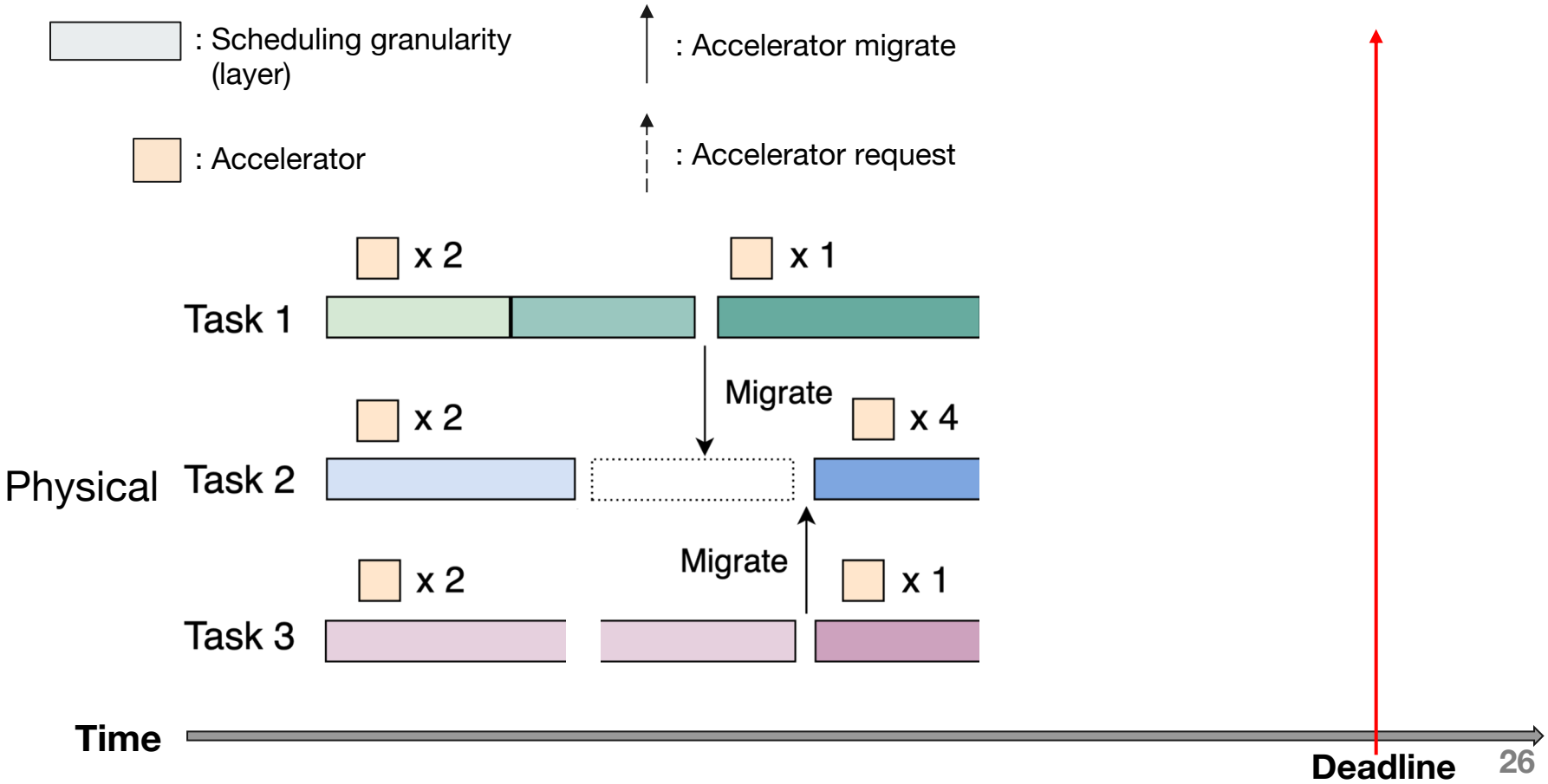
Dynamic score:

```
ddl_score <- time_left_to_target / estimate_latency(# accel)
```

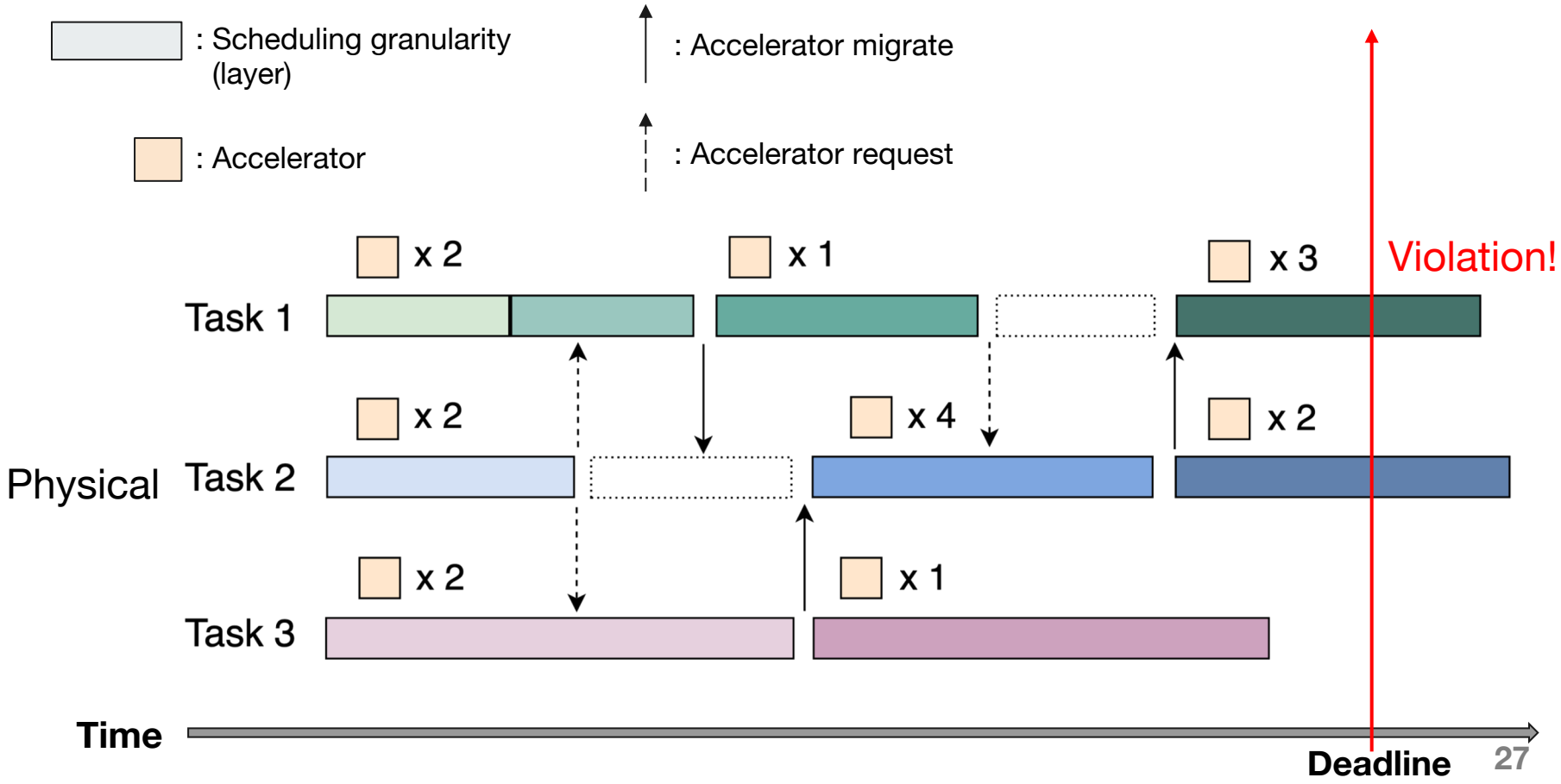
Flow Without AuRORA



Flow Without AuRORA

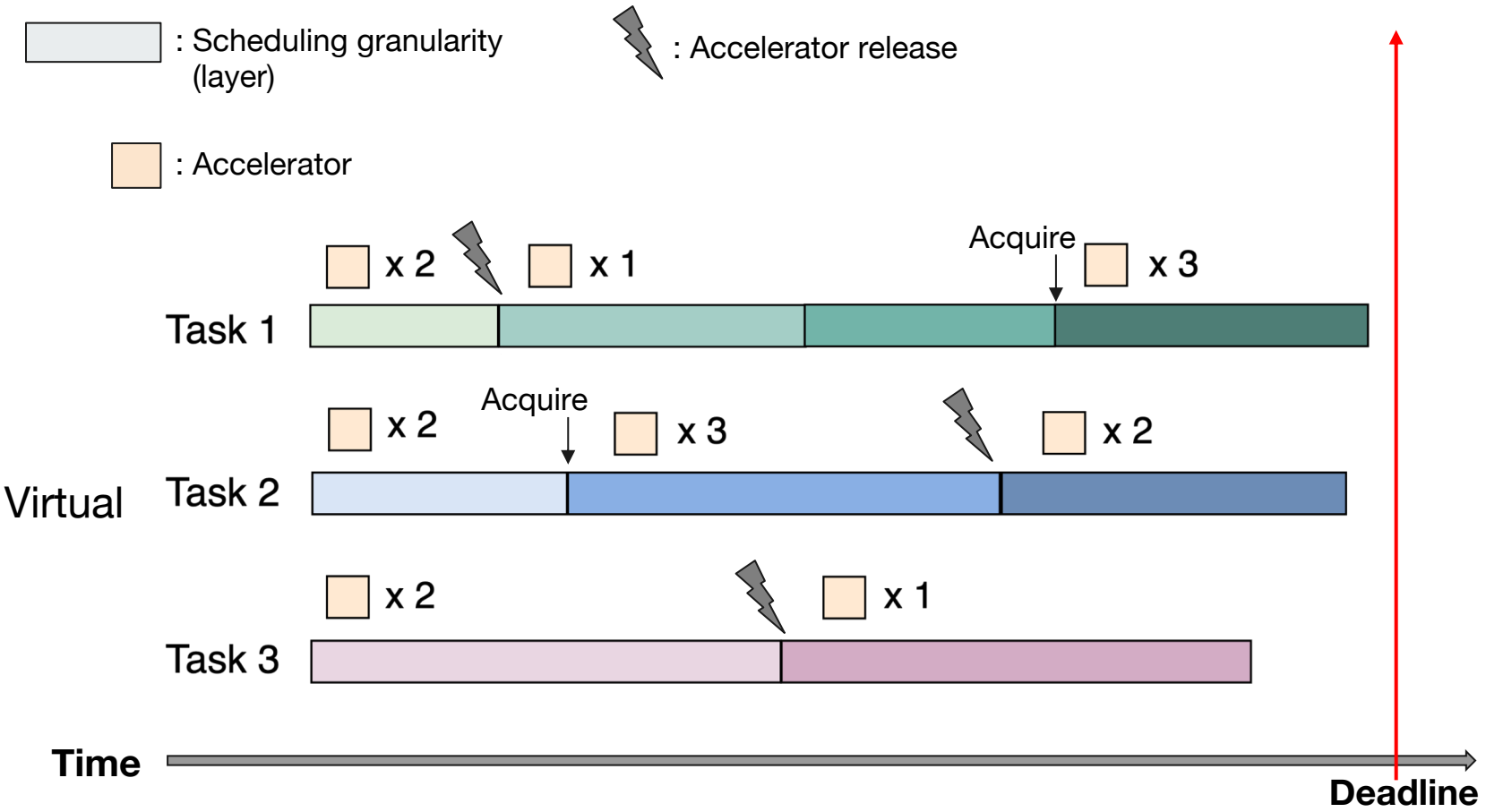


Flow Without AuRORA



AuRORA Runtime - Compute

AuRORA Runtime - Compute



Summary

AuRORA: A full-stack hardware/software integration approach to support virtualized accelerator orchestration

AuRORA enables scalable many-accelerator system for multi-tenant execution

Full-system evaluation using real SoC, real RISC-V cores and accelerators

Performance/area evaluation using physically realizable RTL

Open-sourced, integrated to Chipyard SoC design framework

Open-sourced: https://github.com/ucb-bar/AuRORA