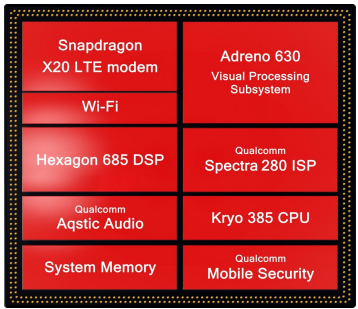
Microkernel Construction

Case Study: M³

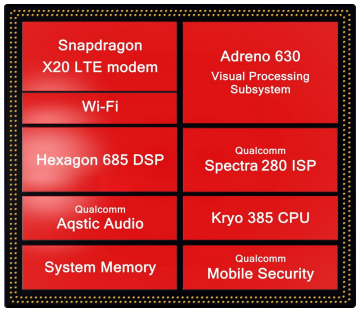
Nils Asmussen

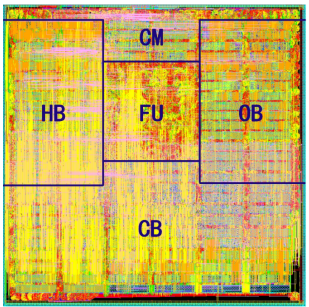
July 4th 2019

Heterogeneous Systems

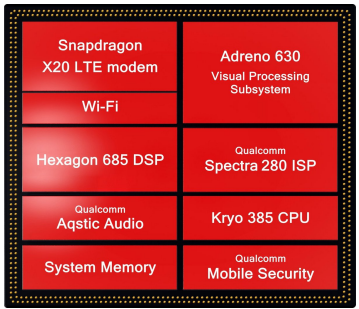


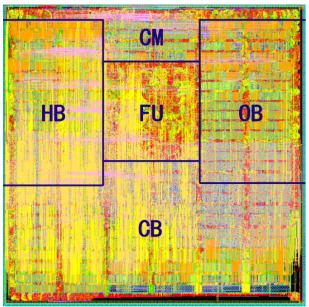
Heterogeneous Systems





Heterogeneous Systems



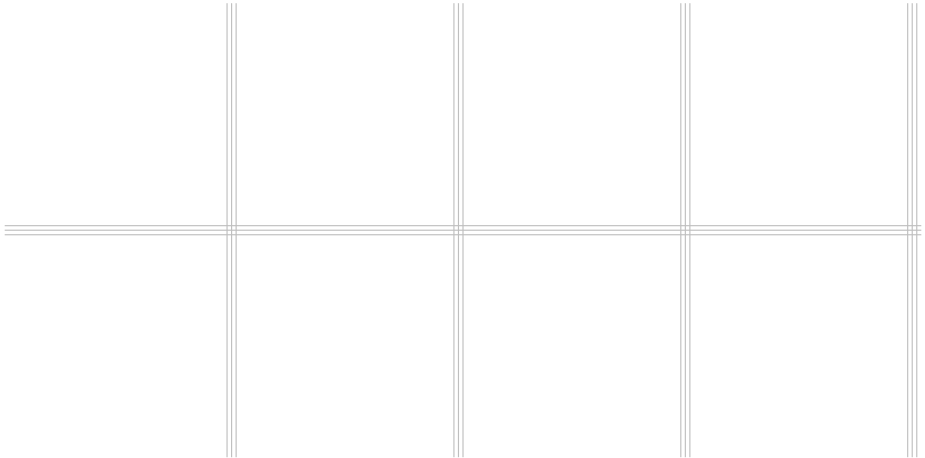


Why?

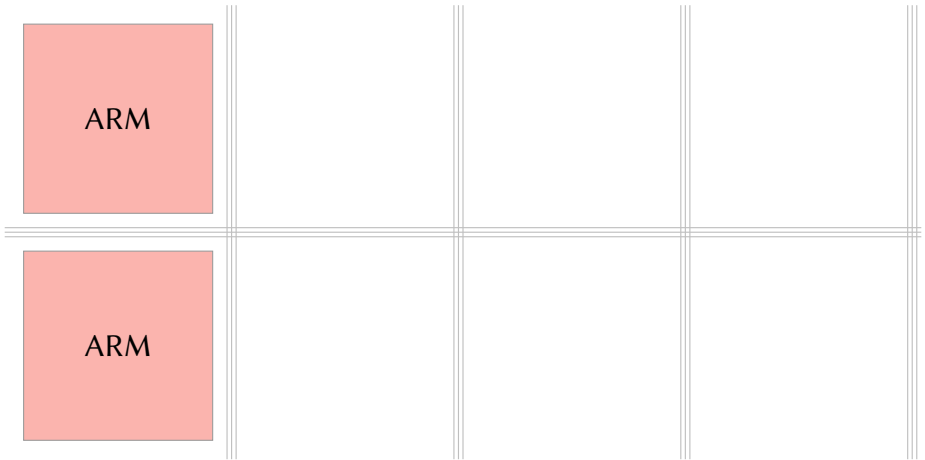
- memcached: FPGA-based implementation is 16 times better in performance per watt than Atom CPU [1]
- machine learning: custom accelerator is 20% faster than GPU and requires 128 times less energy [2]

[1] Thin servers with smart pipes: Designing SoC accelerators for memcached, ISCA'13
[2] PuDianNao: A polyvalent machine learning accelerator, ASPLOS'15

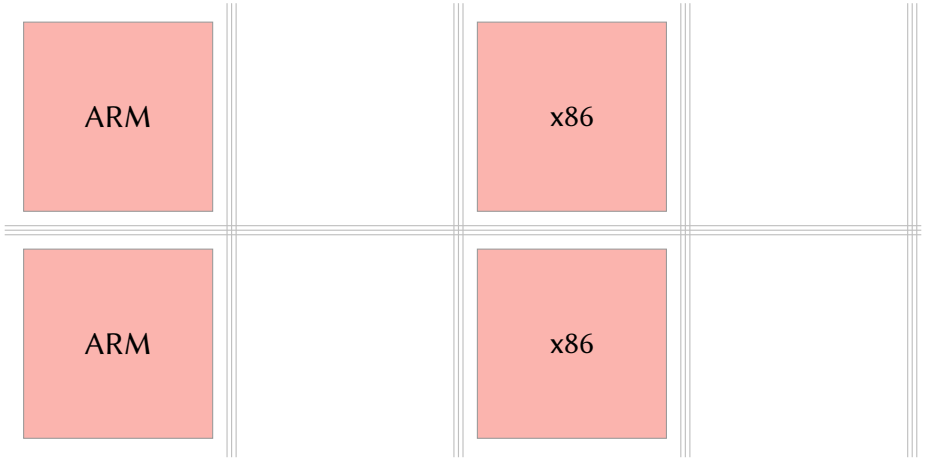
Future Platforms: Problems for Operating Systems



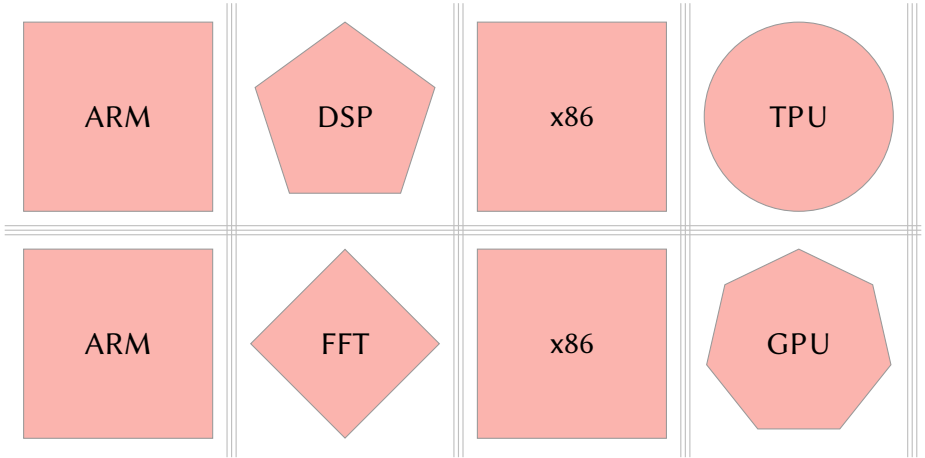
Future Platforms: Problems for Operating Systems



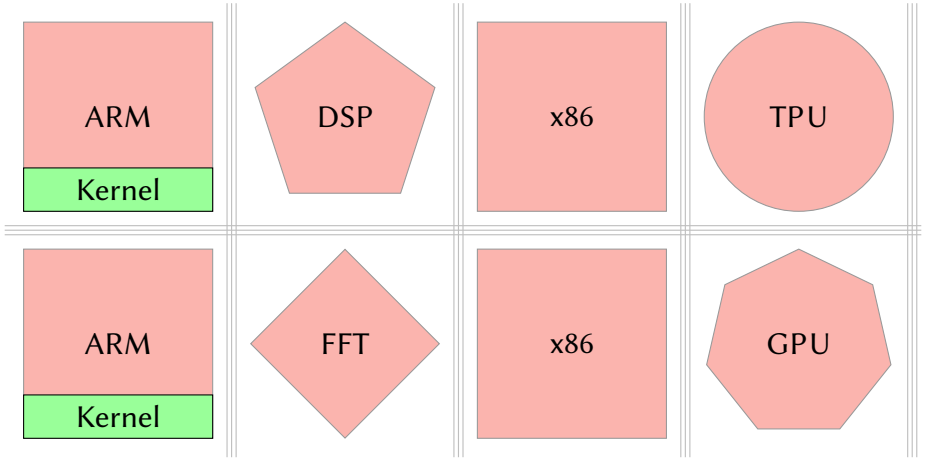
Future Platforms: Problems for Operating Systems



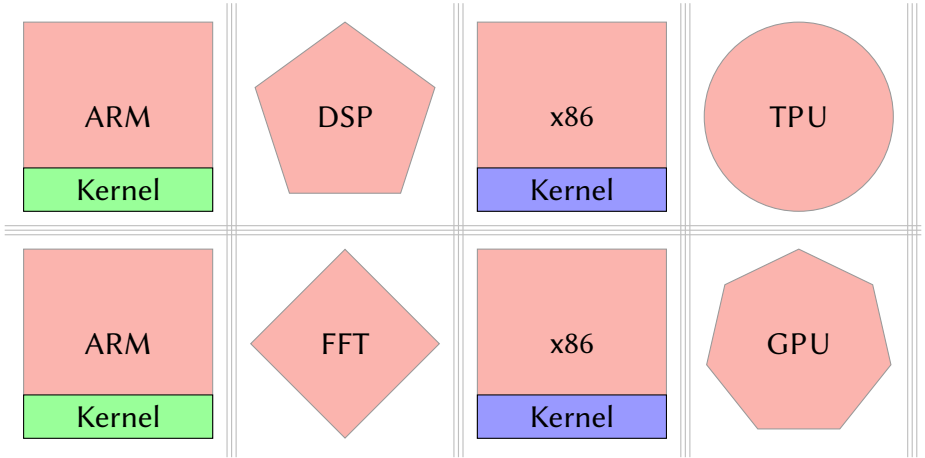
Future Platforms: Problems for Operating Systems



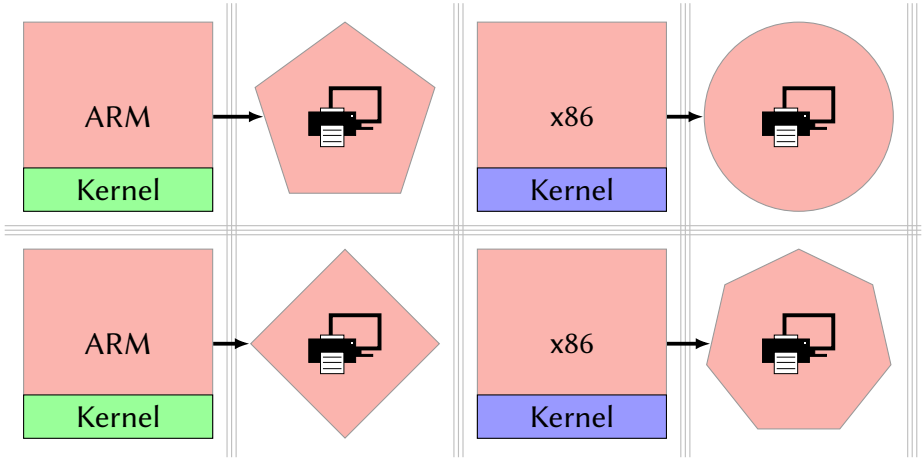
Future Platforms: Problems for Operating Systems



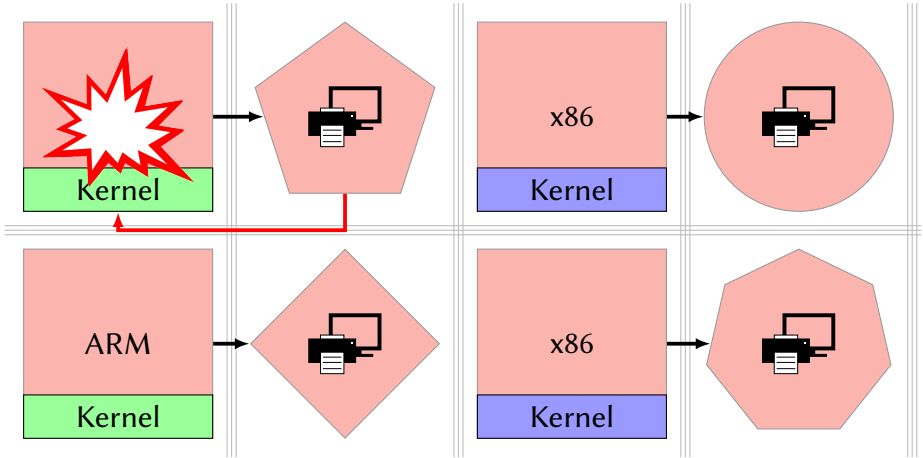
Future Platforms: Problems for Operating Systems



Future Platforms: Problems for Operating Systems



Future Platforms: Problems for Operating Systems



Related Work

Isolation of components

- DPU, NoC-MPU
- IOMMUs

Related Work

Isolation of components

- DPU, NoC-MPU
- IOMMUs

First-class handling of one specific accelerator

- GPUfs, GPUnet, PTask
- ReconOS, BORPH

Related Work

Isolation of components

- DPU, NoC-MPU
- IOMMUs

First-class handling of one specific accelerator

- GPUfs, GPUnet, PTask
- ReconOS, BORPH

OSes for heterogeneous systems

- Barrelfish
- Popcorn Linux, K2
- Helios

What If We Could Change Hardware?

Can we design a system that integrates all types of untrusted compute units as *first-class citizens*?

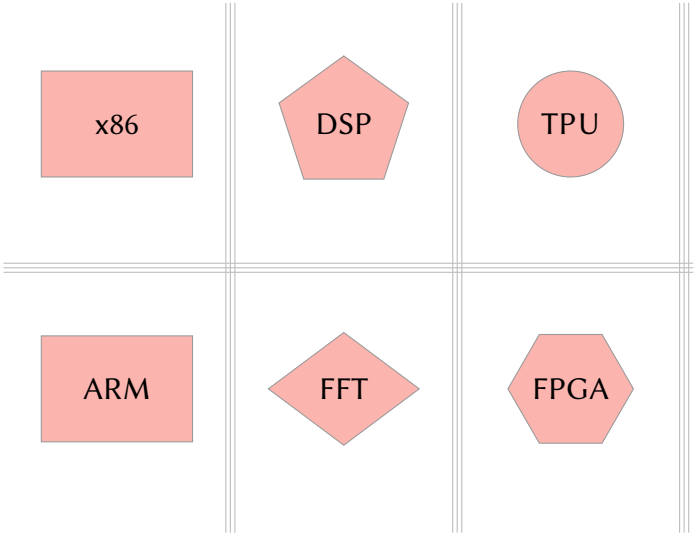
Goals for First-class Citizens

- Prevent harm by untrusted compute units (CUs)
- Access operating-system services by all CUs
- Direct communication between all CUs
- Context switching support for all CUs

Outline

Outline

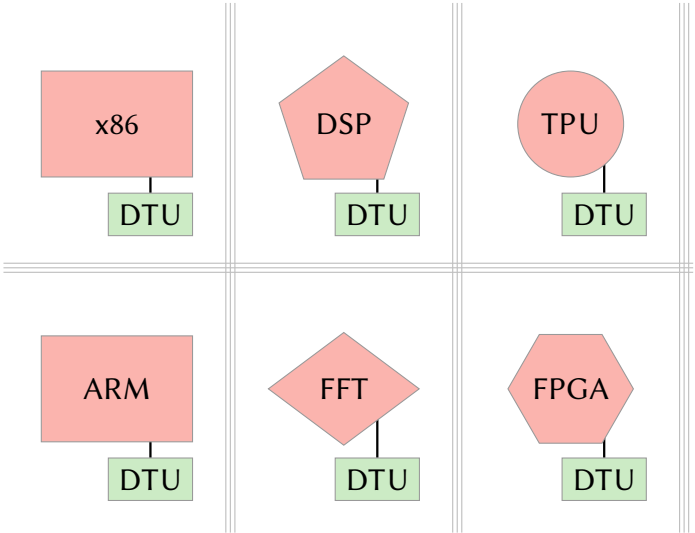
Hardware/Operating System Co-Design



Key Ideas:

- Minimize changes to existing components

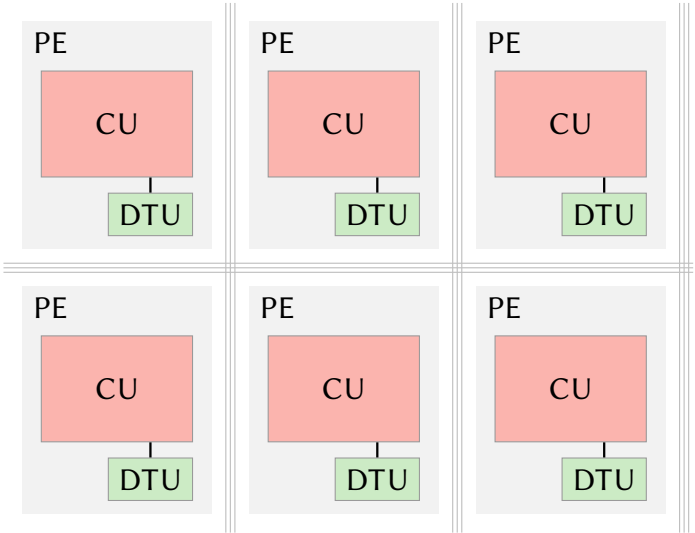
Hardware/Operating System Co-Design



Key Ideas:

- Minimize changes to existing components
- Add uniform interface

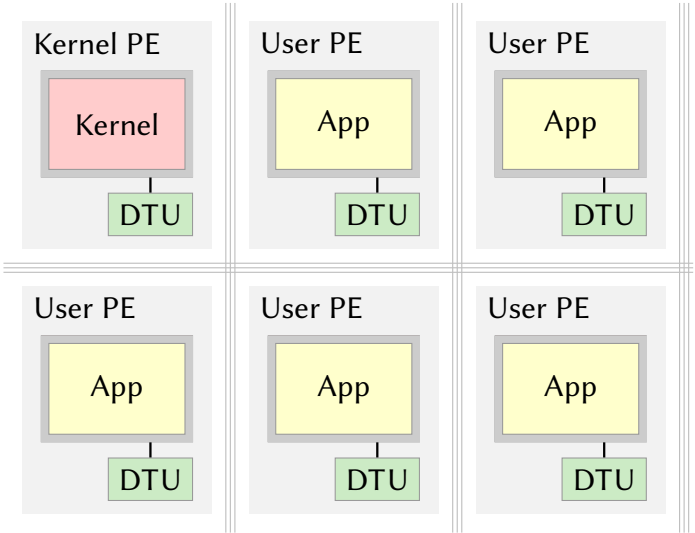
Hardware/Operating System Co-Design



Key Ideas:

- Minimize changes to existing components
- Add uniform interface

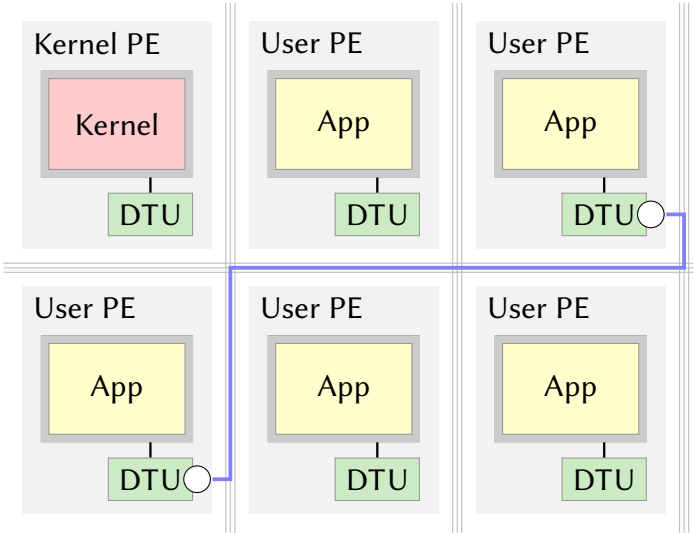
Hardware/Operating System Co-Design



Key Ideas:

- Minimize changes to existing components
- Add uniform interface
- Kernel controls user PEs remotely

Hardware/Operating System Co-Design

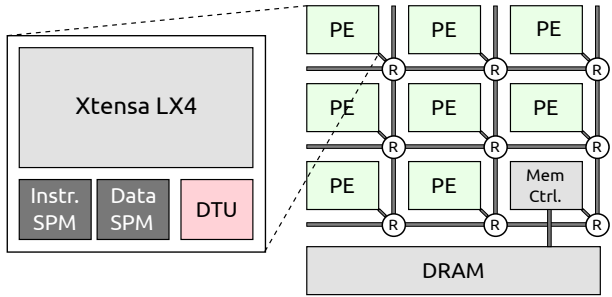


Key Ideas:

- Minimize changes to existing components
- Add uniform interface
- Kernel controls user PEs remotely
- Direct communication

Outline

Tomahawk



PEs have no OS support:

- No privileged mode
- No MMU, no caches, but SPM
- T2: simple DTU; T4: most features

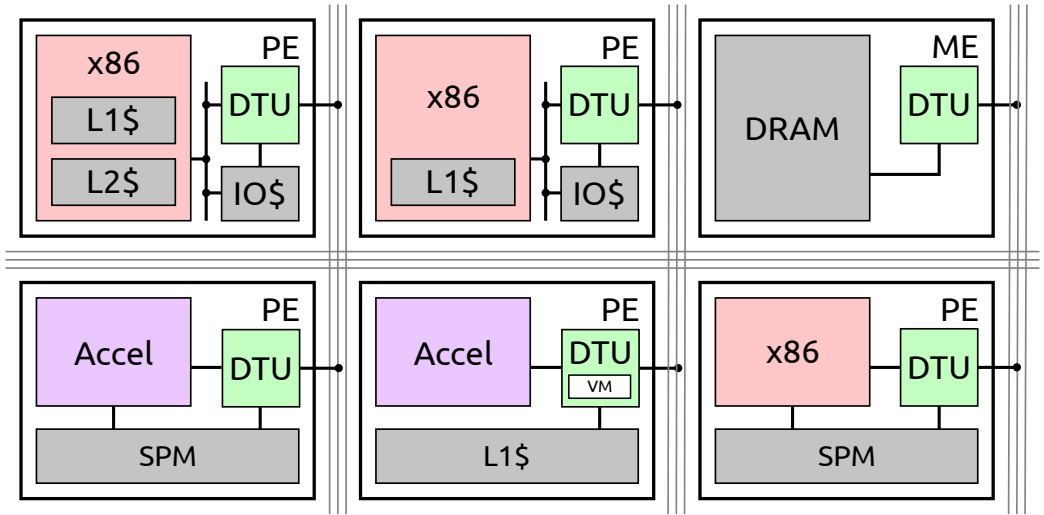
Linux

- M^3 runs on Linux using it as a virtual machine
- A process simulates a PE, having two threads (CPU + DTU)
- DTUs communicate over UNIX domain sockets
- No accuracy because
  - Programs are directly executed on host
  - Data transfers have huge overhead compared to HW
- Very useful for debugging and early prototyping

gem5

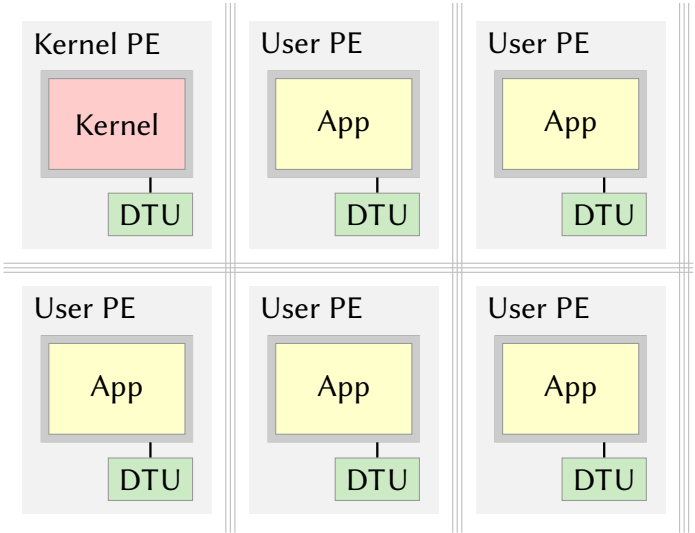
- Modular platform for computer architecture research
- Supports various ISAs (x86, ARM, Alpha, RISC-V, ...)
- Provides detailed CPU and memory models
- Cycle-accurate simulation
- Added DTU model to gem5
- Added hardware accelerators

gem5 – Example Configuration



Outline

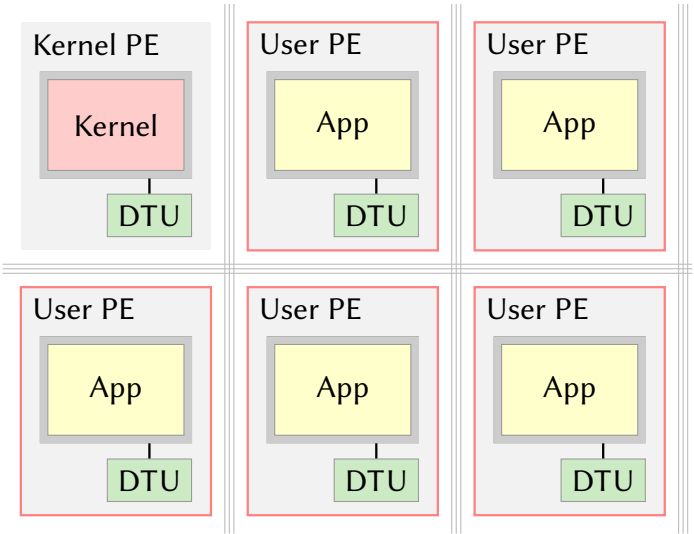
Isolation



DTU-based isolation:

- Additional protection layer

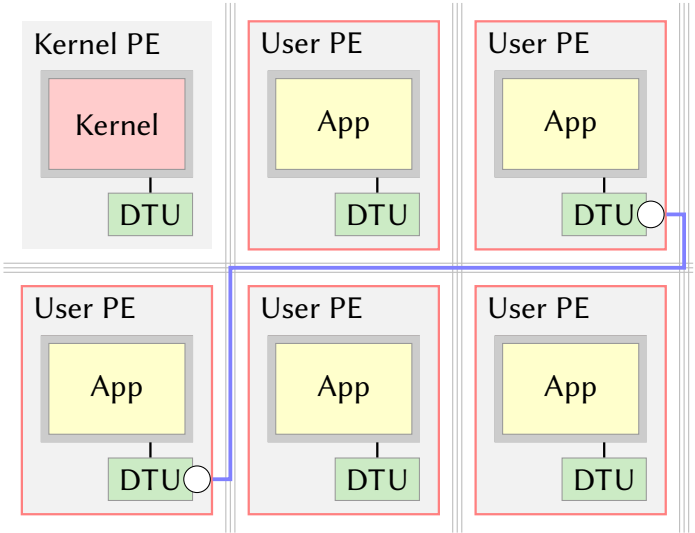
Isolation



DTU-based isolation:

- Additional protection layer

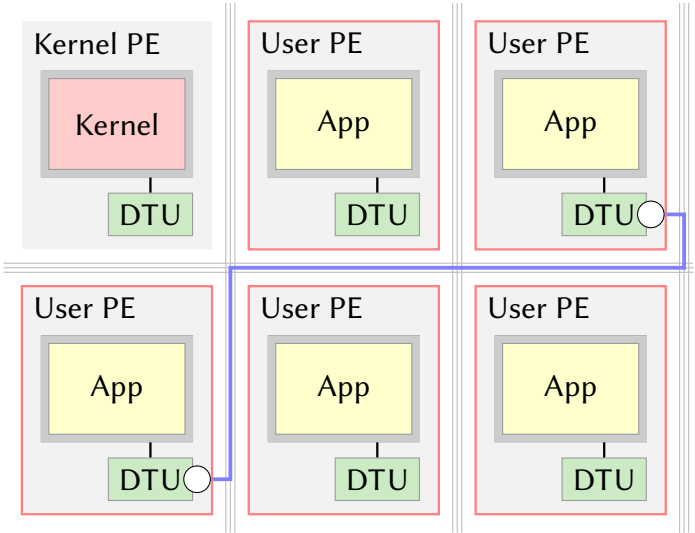
Isolation



DTU-based isolation:

- Additional protection layer
- Only kernel PE can establish communication channels

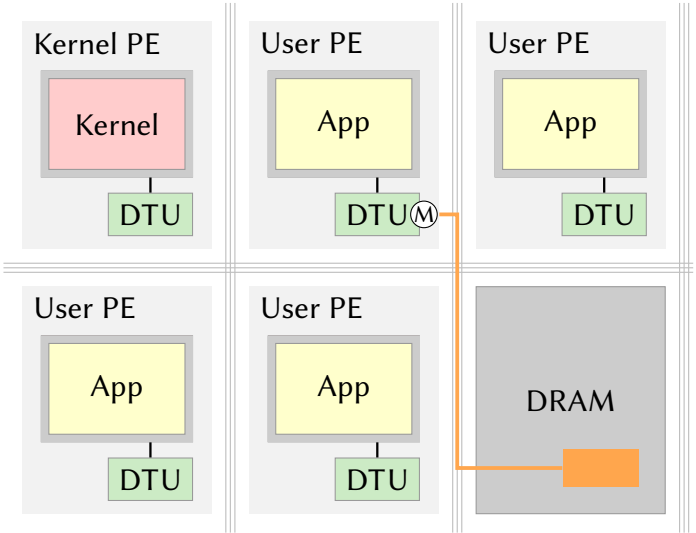
Isolation



DTU-based isolation:

- Additional protection layer
- Only kernel PE can establish communication channels
- User PEs can only use established channels

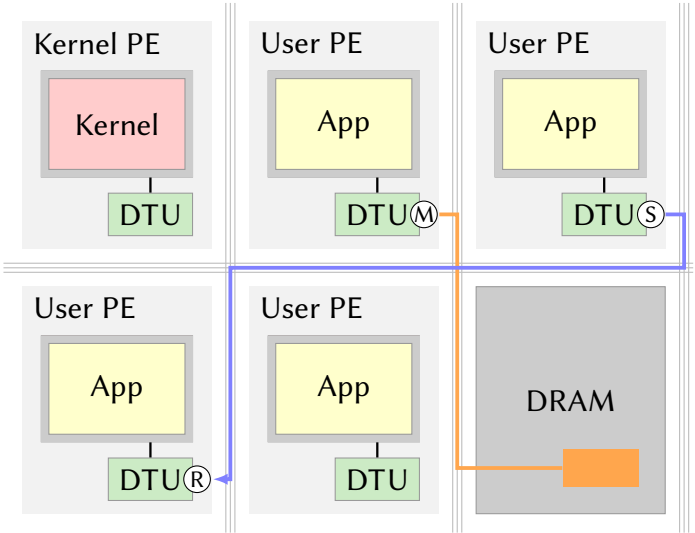
Communication



DTU provides *endpoints* to:

- Access memory (contiguous range, byte granular)

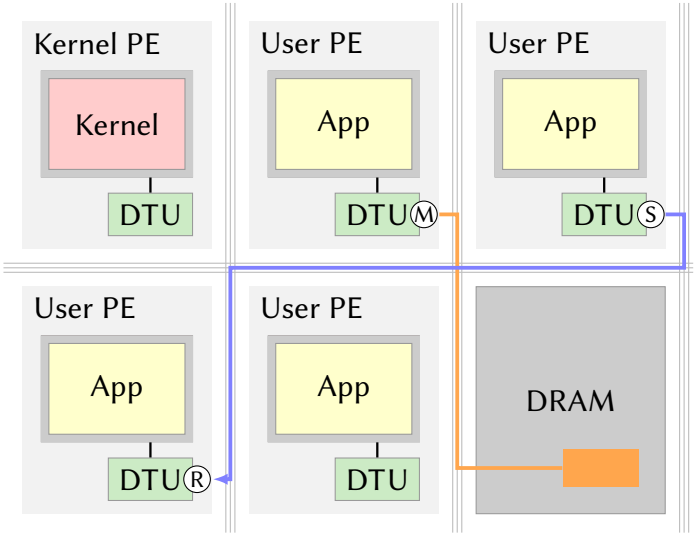
Communication



DTU provides *endpoints* to:

- Access memory (contiguous range, byte granular)
- Receive messages into a receive buffer
- Send messages to a receiving endpoint

Communication

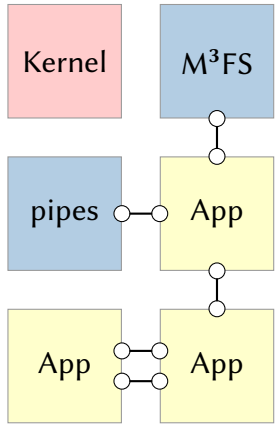


DTU provides *endpoints* to:

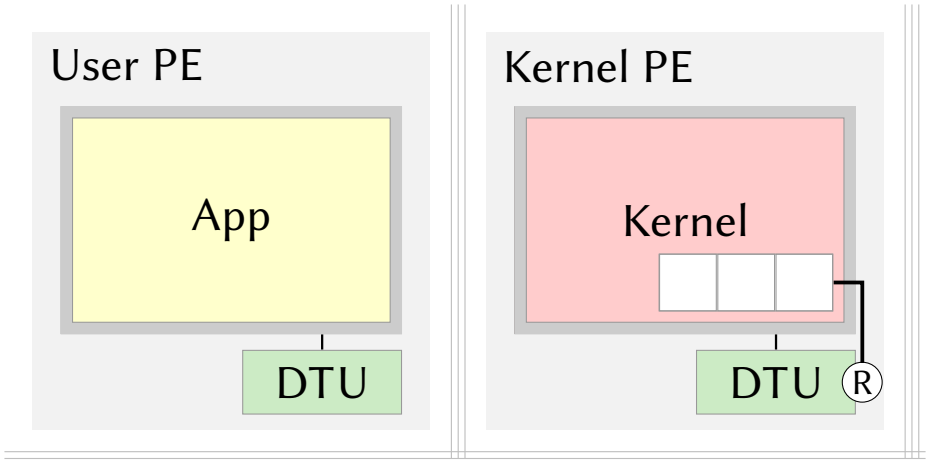
- Access memory (contiguous range, byte granular)
- Receive messages into a receive buffer
- Send messages to a receiving endpoint
- Replies for RPC

OS Design

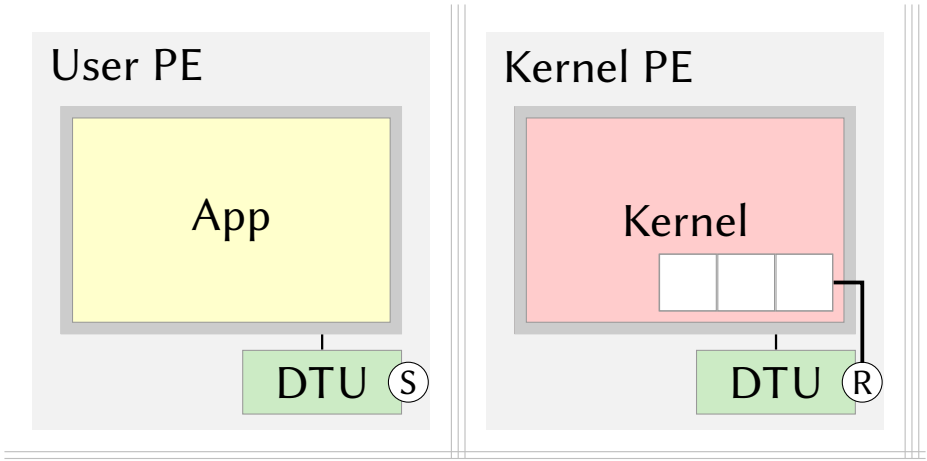
- M³: **M**icrokernel-based syste**m** for het. **m**anycores (or L4 ± 1)
- Implemented from scratch
- Drivers, filesystems, etc. implemented on user PEs
- Kernel manages permissions, using capabilities
- DTU enforces permissions (communication, memory access)
- Kernel is independent of other PEs



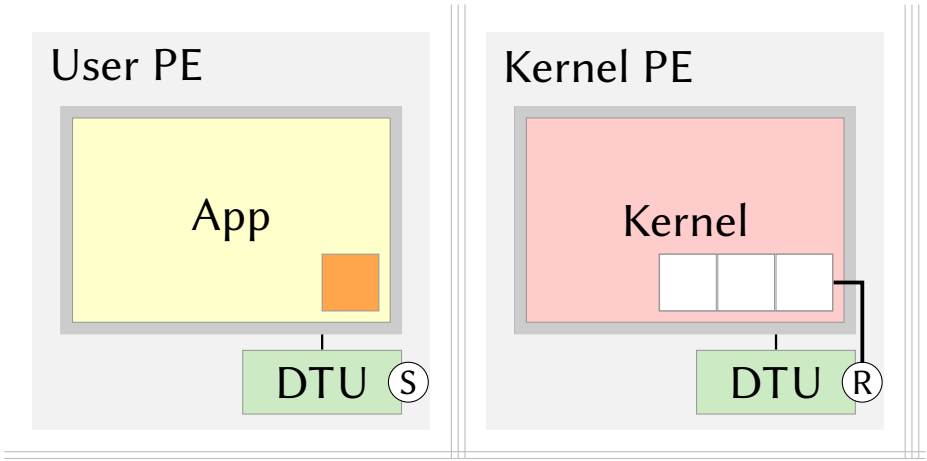
M³ System Call



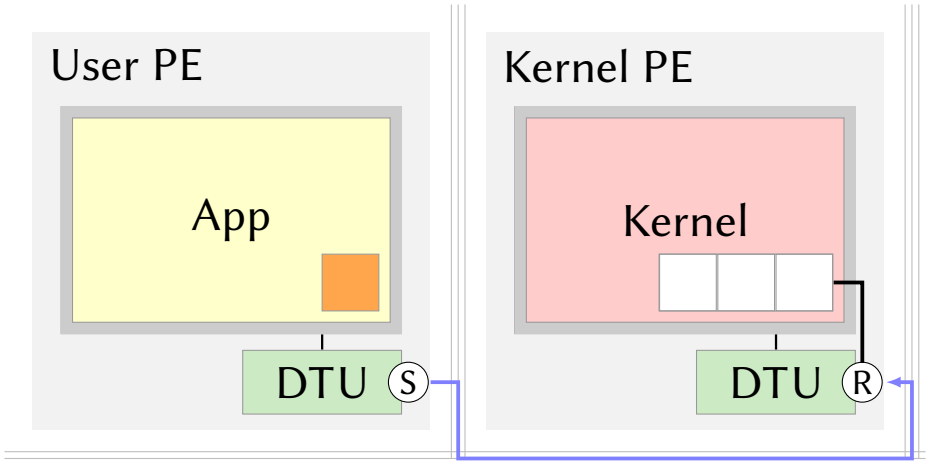
M³ System Call



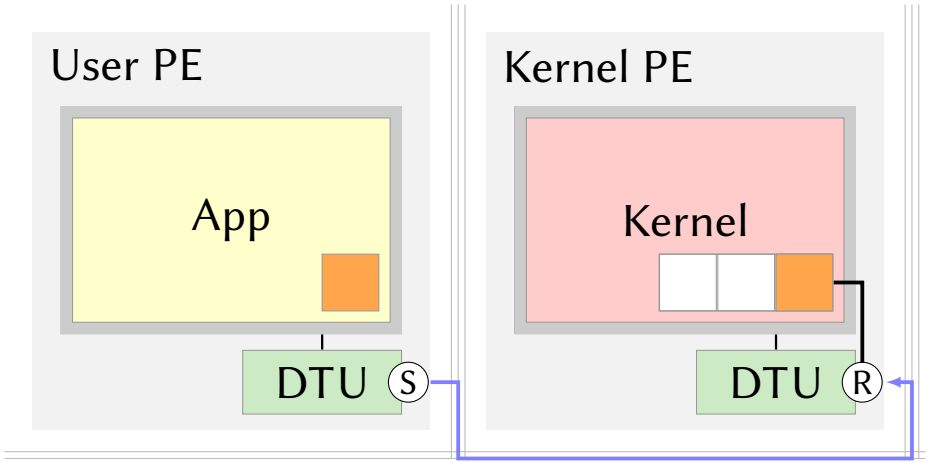
M³ System Call



M³ System Call

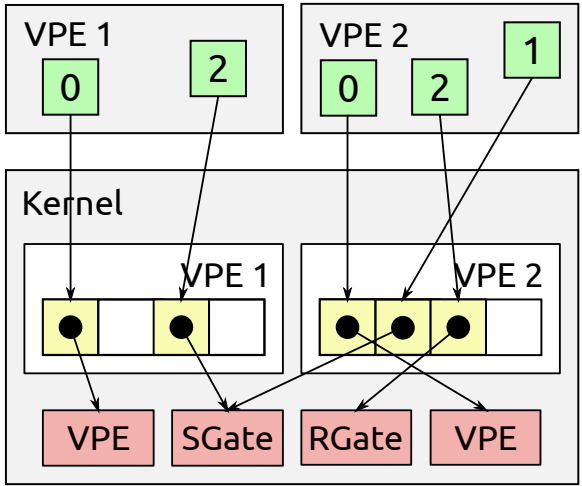


M³ System Call



Outline

Overview



Capabilities

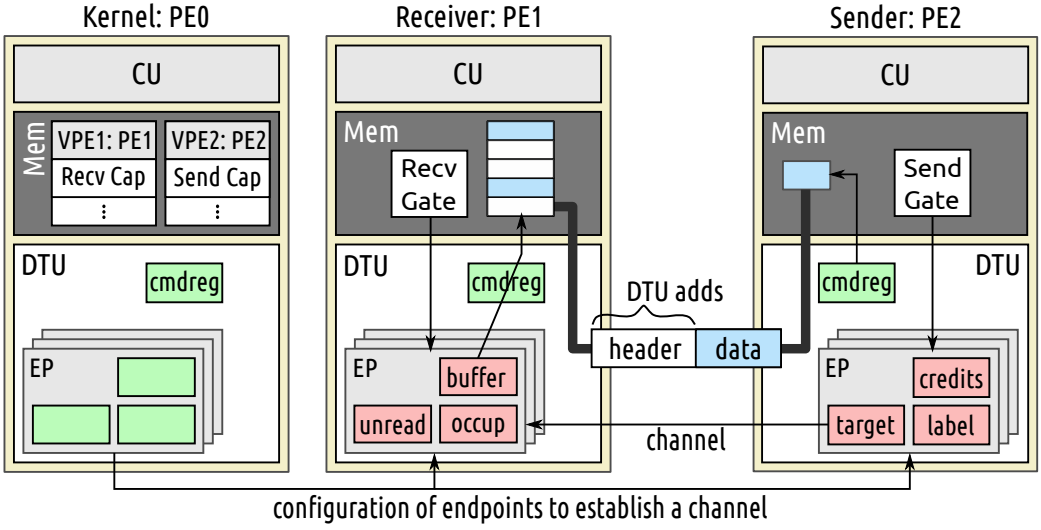
M^3 has the following capabilities:

- Send: send messages to a receive EP
- Receive: receive messages from send EPs
- Memory: access remote memory via DTU
- Mapping: access remote memory via load/store
- Service: create sessions
- Session: exchange caps with service
- Endpoint: configure EPs of own or foreign DTU
- VPE: use a PE

Capability Exchange

- Kernel provides syscalls to create, exchange, and revoke caps
- There are two ways to exchange caps:
  1. Directly with another VPE (typically, a child VPE)
  2. Over a session with a service
- The kernel offers two operations:
  1. Delegate: send capability to somebody else
  2. Obtain: receive capability from somebody else
- Difference to L4:
  - Applications communicate directly, without involving the kernel
  - $\rightarrow$ Capability exchange cannot be done during IPC
  - Special communication channel between kernel and servers
  - Kernel uses this channel to send exchange requests to server

Communication



Virtual PEs

- M³ kernel manages user PEs in terms of VPEs
- VPE is combination of a process and a thread
- VPE creation yields a VPE capability and memory capability
- Library provides primitives like `fork` and `exec`
- VPEs are used for *all* PEs:
  - Accelerators are not handled differently by the kernel
  - All VPEs can perform system calls
  - All VPEs can have time slices and priorities
  - …

VPEs – Examples

Executing ELF-Binaries

```
VPE vpe("test");
char *args[] = {"/bin/hello", "foo", "bar"};
vpe.exec(3, args);
```

VPEs – Examples

Executing ELF-Binaries

```
VPE vpe("test");
char *args[] = {"/bin/hello", "foo", "bar"};
vpe.exec(3, args);
```

Asynchronous Lambdas

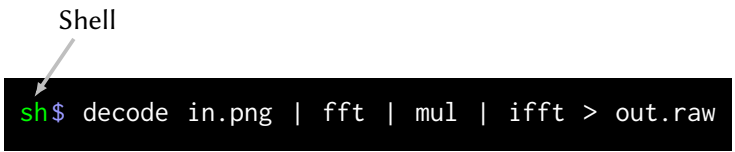
```
VPE vpe("test");
MemGate mem = MemGate::create_global(0x1000, RW);
vpe.delegate(CapRngDesc(mem.sel(), 1));
vpe.run_async([&mem]() {
    mem.read(buf, sizeof(buf));
});
```

Outline

OS Service Access for all CUs

```
sh$ decode in.png | fft | mul | ifft > out.raw
```

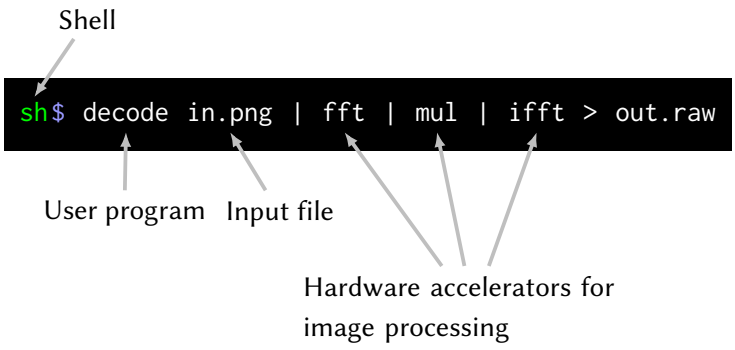
OS Service Access for all CUs



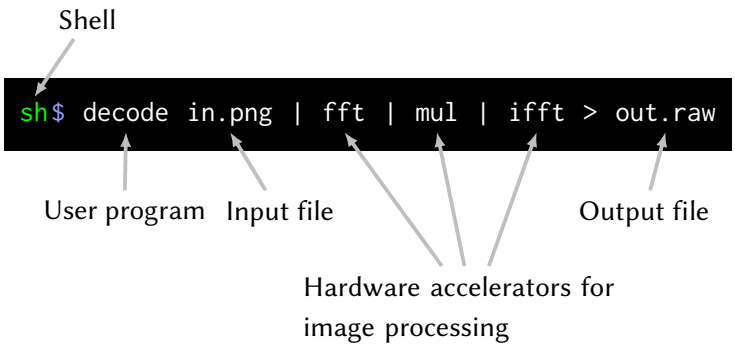
OS Service Access for all CUs

OS Service Access for all CUs

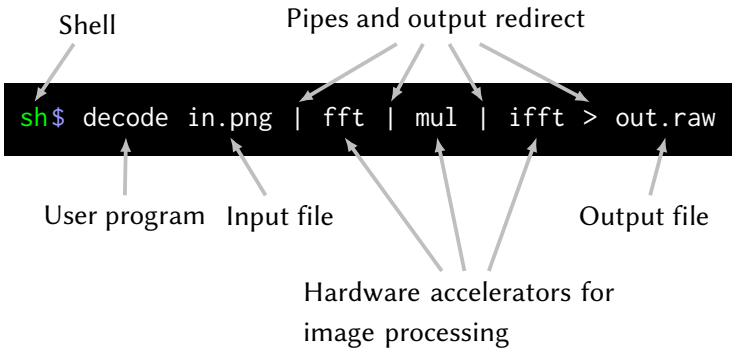
OS Service Access for all CUs



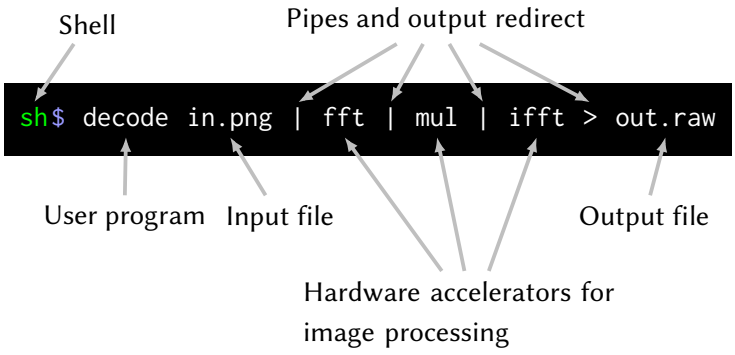
OS Service Access for all CUs



OS Service Access for all CUs

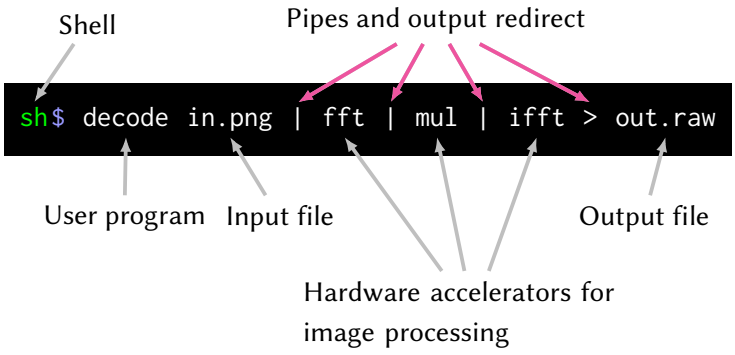


OS Service Access for all CUs



Challenges:

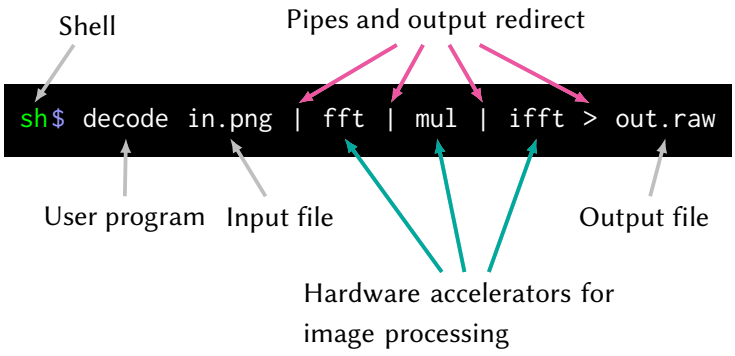
OS Service Access for all CUs



Challenges:

- OS must provide generic protocols

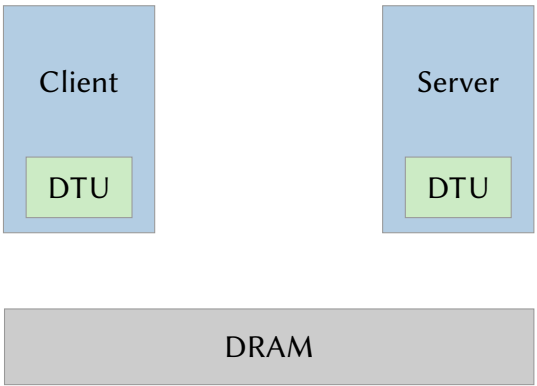
OS Service Access for all CUs



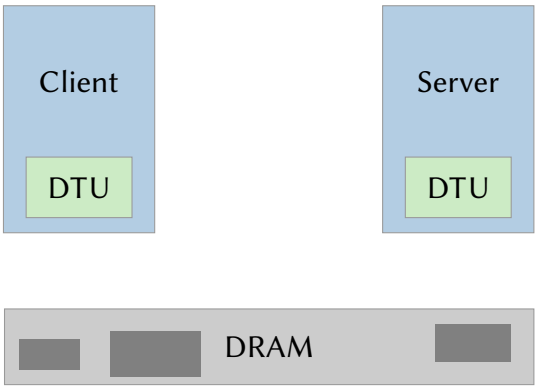
Challenges:

- OS must provide generic protocols
- Accelerators need support for protocols

Generic Protocols



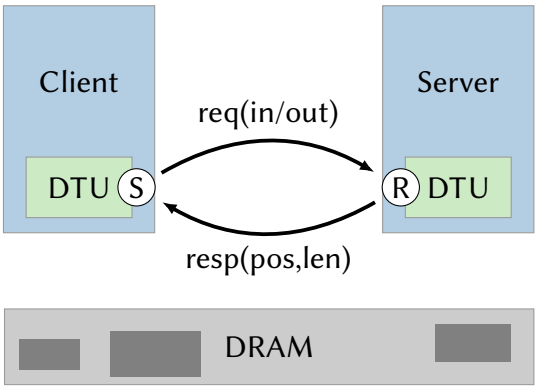
Generic Protocols



File protocol:

- Data in memory

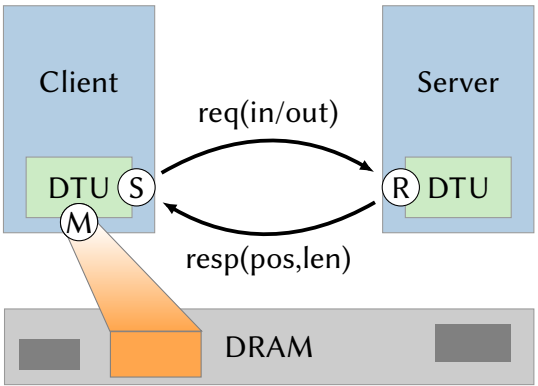
Generic Protocols



File protocol:

- Data in memory
- RPC between client and server
  - `req(in/out)` requests next piece, implicitly commits previous piece
  - `commit(nbytes)` commits `nbytes` of previous piece

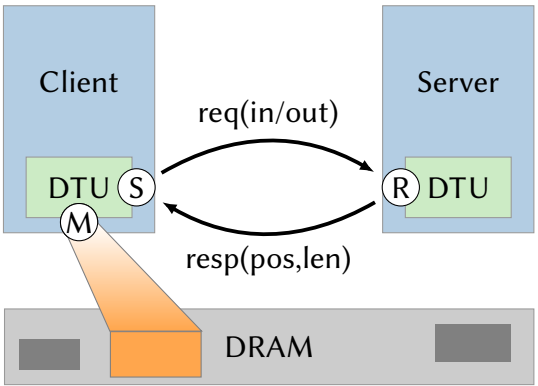
Generic Protocols



File protocol:

- Data in memory
- RPC between client and server
  - `req(in/out)` requests next piece, implicitly commits previous piece
  - `commit(nbytes)` commits `nbytes` of previous piece
- Server configures client's memory EP

Generic Protocols



File protocol:

- Data in memory
- RPC between client and server
  - `req(in/out)` requests next piece, implicitly commits previous piece
  - `commit(nbytes)` commits `nbytes` of previous piece
- Server configures client's memory EP
- Client accesses data via DTU

Implementation: M³FS – Overview

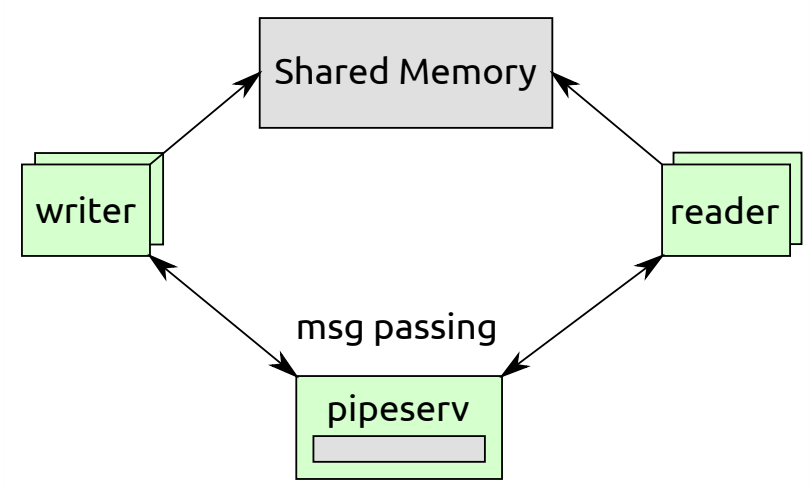
- M³FS organizes the file's data in extents
- M³FS can be used with a memory and disk backend
  - With memory backend, FS image is a contiguous region in DRAM
  - Clients get access to parts of the image
  - With disk backend, M³FS uses a buffer cache in DRAM
  - Clients get access to parts of buffer cache
- Two types of sessions: *metadata session*, *file session*
- Metadata session is created first, allows `stat`, open, ...
- open creates a new file session
- Both sessions can be cloned to provide other VPEs access

Implementation: M³FS – File Protocol

- The file session implements the file protocol (plus seeking)
- File session holds file position and advances it on read/write
- `req(in/out)` request next extent
- M³FS configures client's EP for this extent
- Appending reserves new space, invisible to other clients
- `commit(nbytes)` commits a previous append

Implementation: Pipe – Overview

Implementation: Pipe – Overview



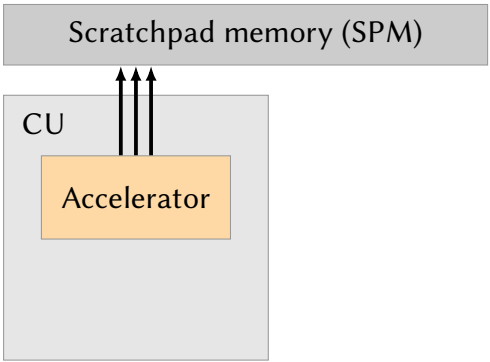
Implementation: Pipe

- Two types of sessions: *pipe session*, *channel session*
- Pipe session represents whole pipe, allows to create channels
- Channel session implements file protocol
- Channel session can be cloned
- Server configures client's EP just once at the beginning
- `req(in/out)` request access to next data
- `commit(nbytes)` commits previous request

File Multiplexing

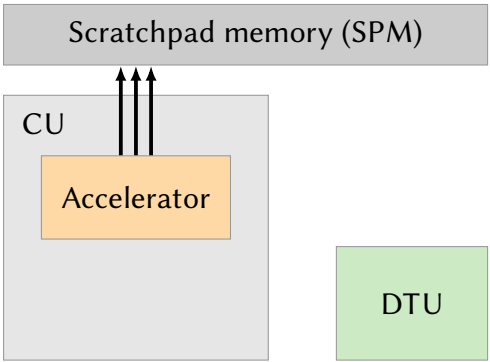
- File protocol maps directly to EPs (limited resource)
- Number of open files shouldn't be limited (that much)
- libm3 dedicates at most 4 EPs to files and multiplexes them
- Multiplexing requires:
  1. `commit(nbytes)` to commit read/written data
  2. revocation of EP capability (old server)
  3. delegation of EP capability (new server)
  4. next `read`/`write` will contact server again
- Fortunately, file multiplexing does almost never happen

Additions to Accelerator



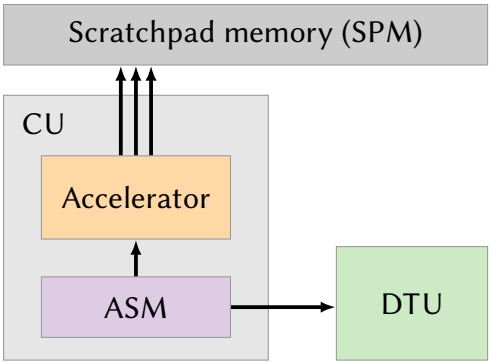
Off-the-shelf accelerators

Additions to Accelerator



Off-the-shelf accelerators

Additions to Accelerator

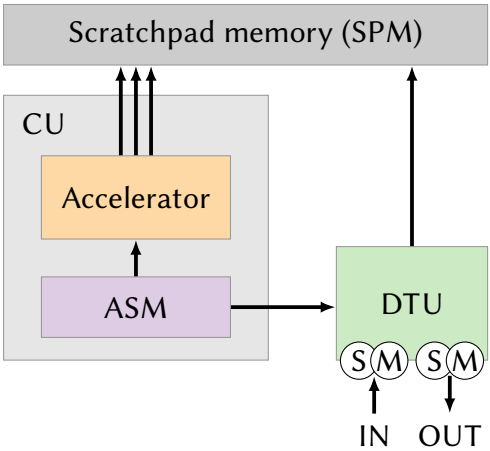


Off-the-shelf accelerators

Accelerator Support Module (ASM):

- Interacts with DTU and accelerator

Additions to Accelerator

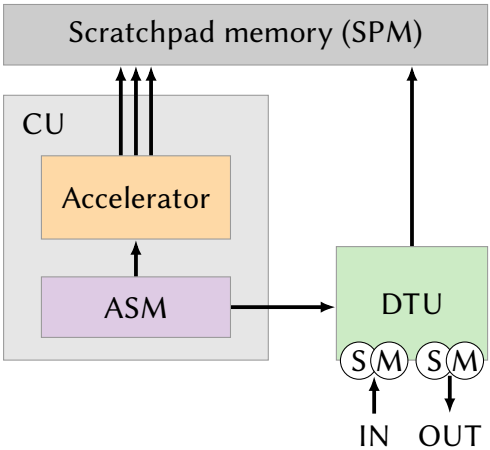


Off-the-shelf accelerators

Accelerator Support Module (ASM):

- Interacts with DTU and accelerator
- Implements file protocol for input and output channel

Additions to Accelerator



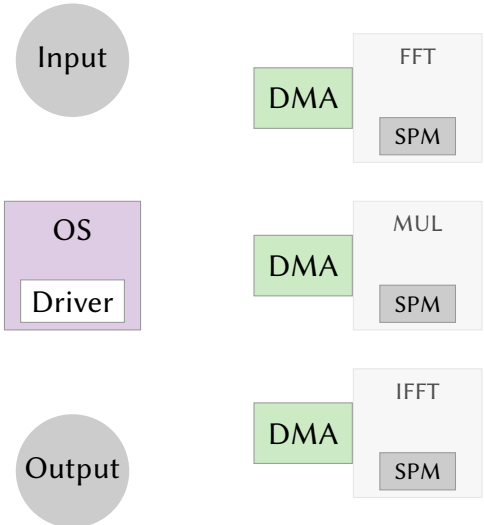
Off-the-shelf accelerators

Accelerator Support Module (ASM):

- Interacts with DTU and accelerator
- Implements file protocol for input and output channel
- ASM assumes that endpoints are setup externally by software

Demo

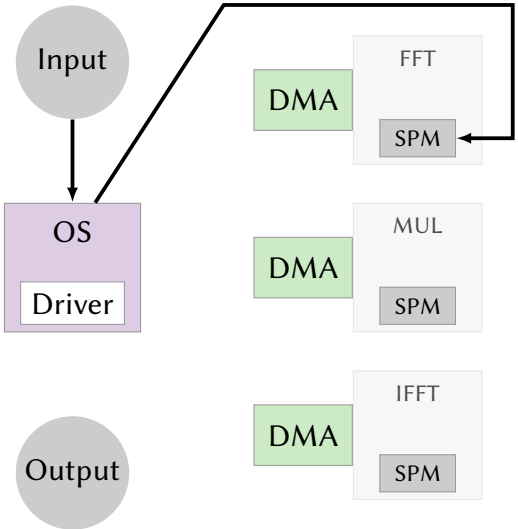
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

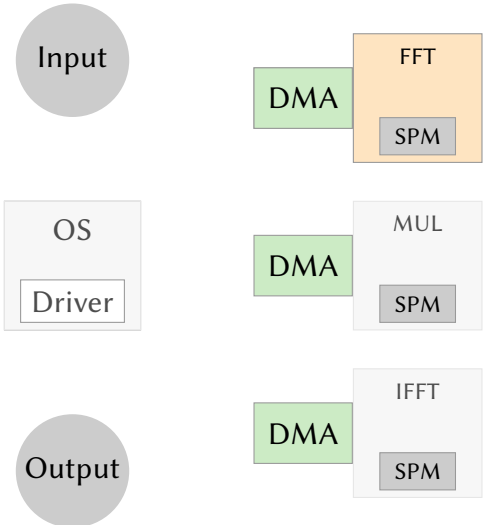
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

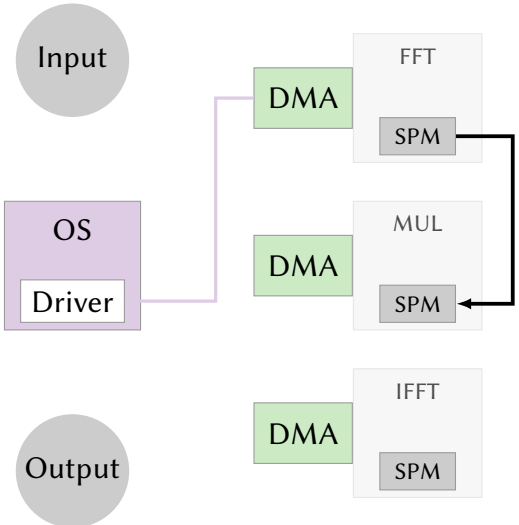
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

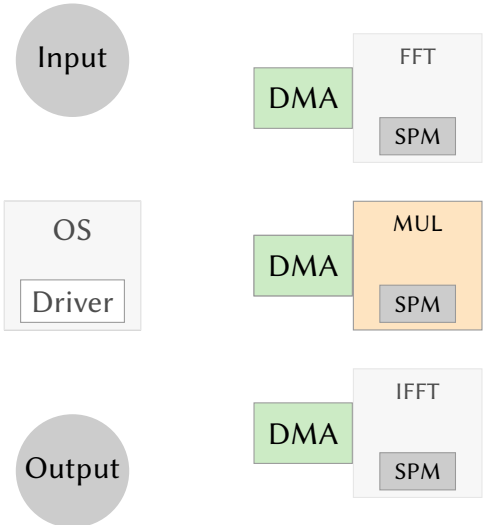
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

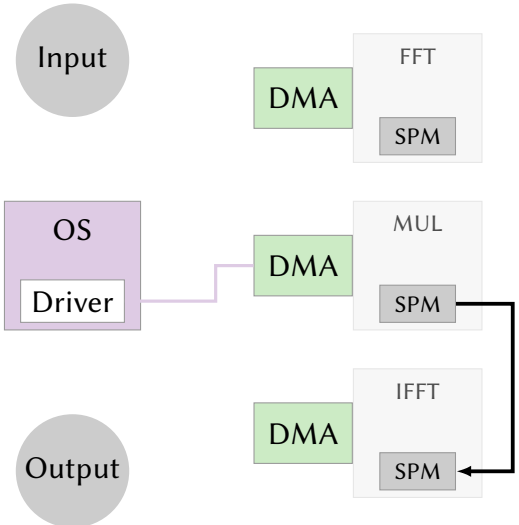
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

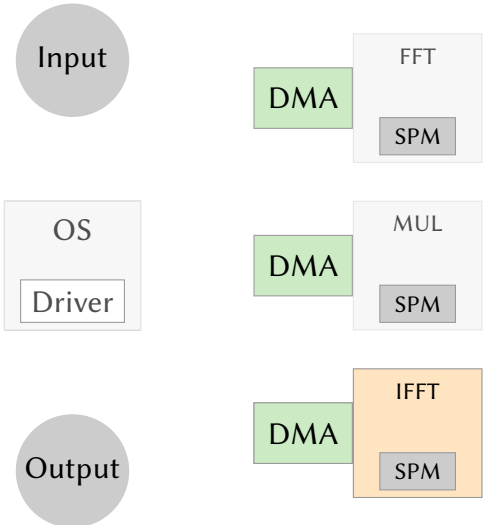
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

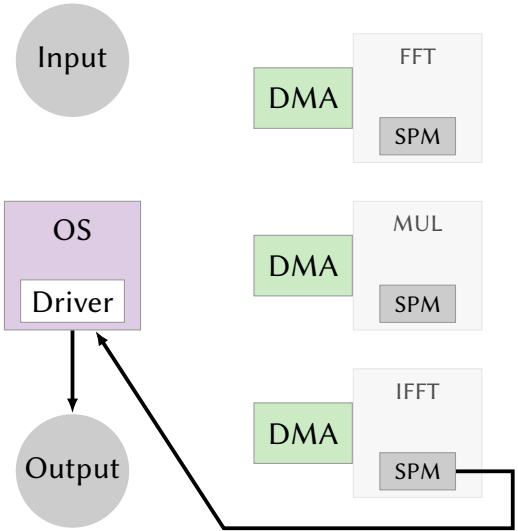
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

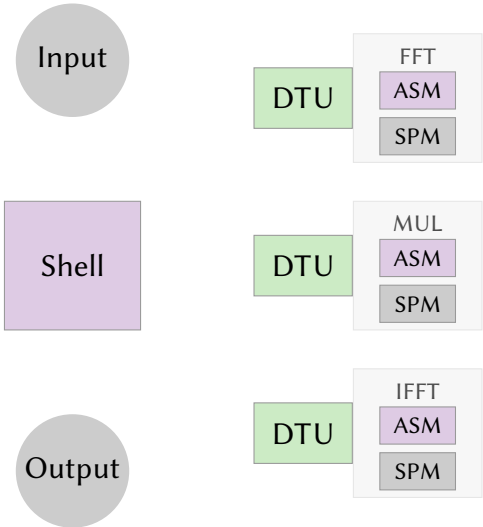
Accelerator Chains: Assisted by OS



OS-assisted accelerator chains:

- OS drives copy-in/copy-out of accelerator SPMs
- Only simple DMA needed
- Like in traditional systems, high CPU overhead for OS

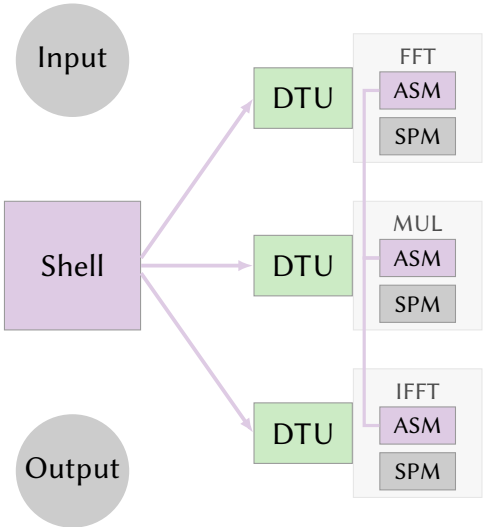
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

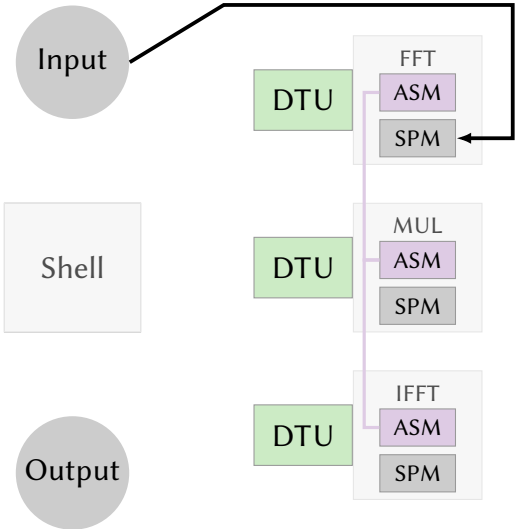
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

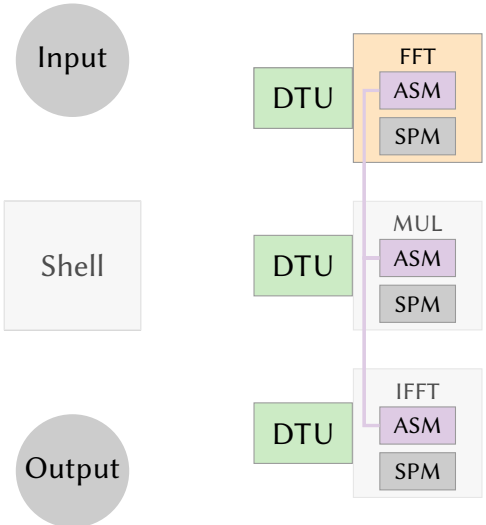
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

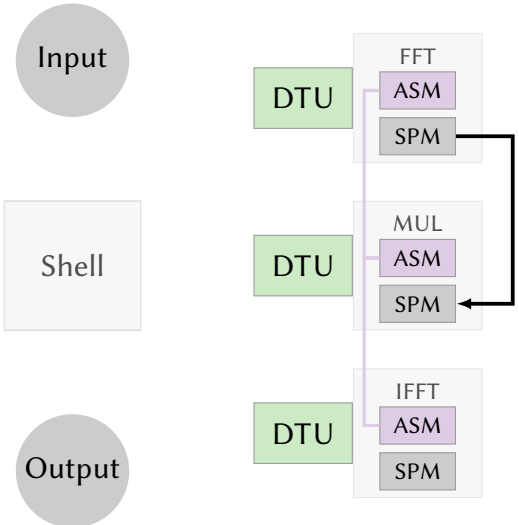
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

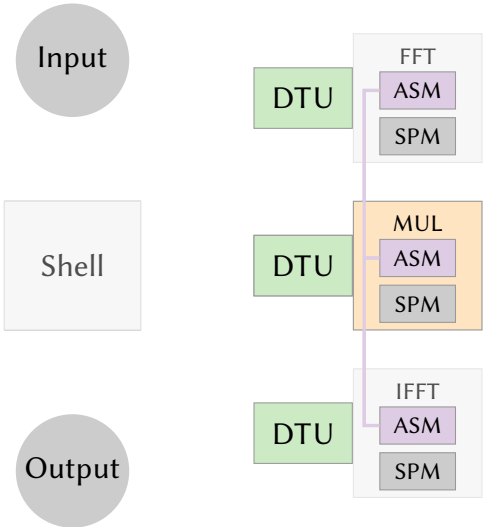
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

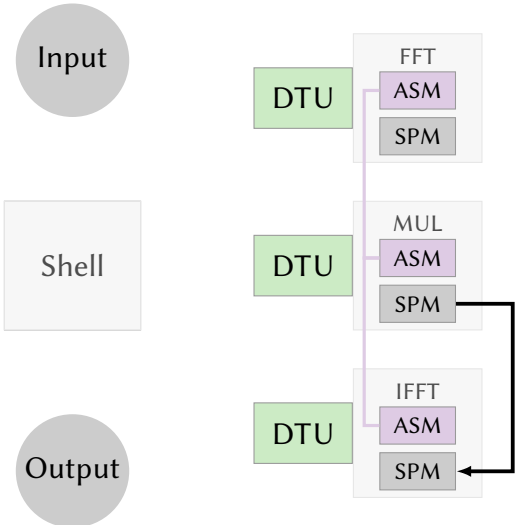
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

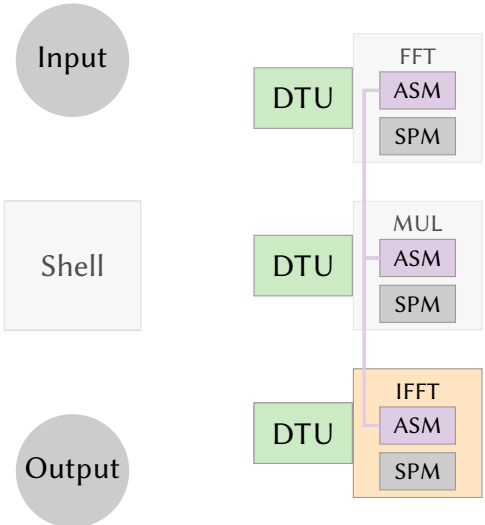
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

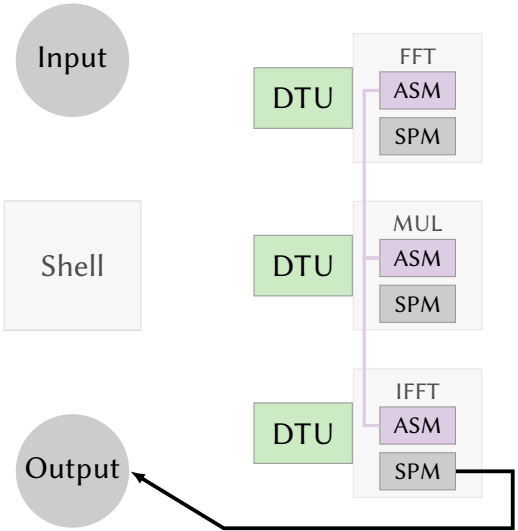
Accelerator Chains: Fully Autonomous



Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

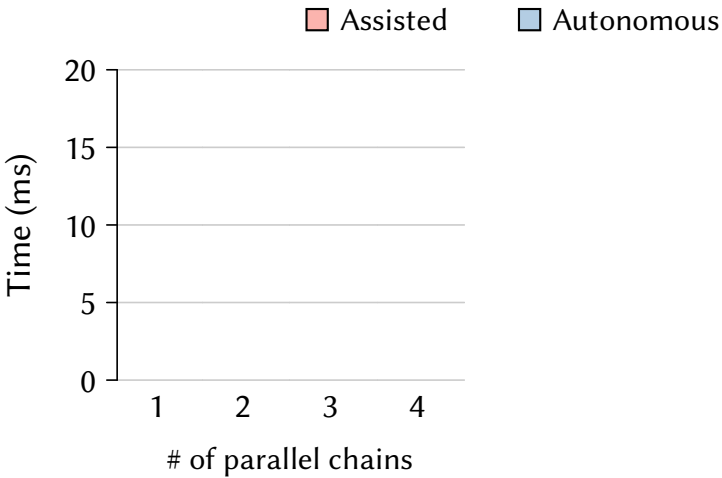
Accelerator Chains: Fully Autonomous



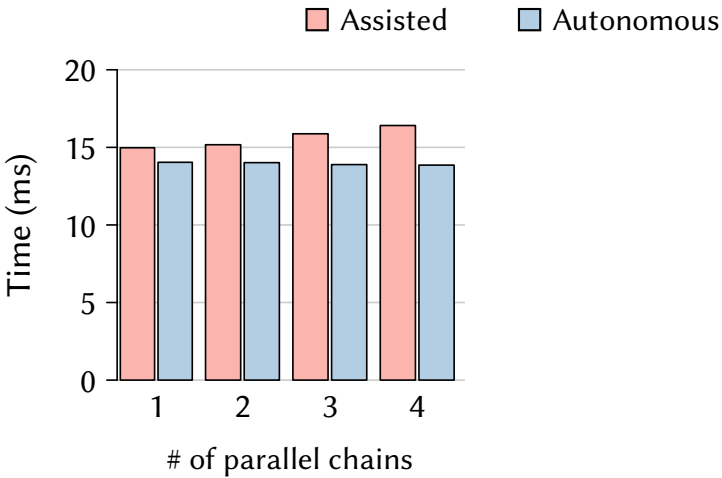
Autonomous accelerator chains:

- Shell configures all endpoints
- ASMs of accelerators drive DTUs to transfer data autonomously
- Fully offloaded,
  almost no CPU overhead for OS

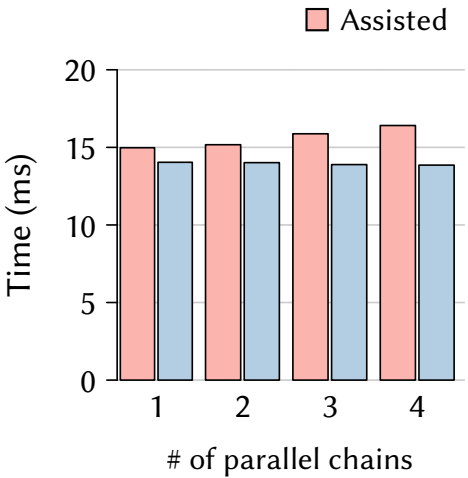
Accelerator Chains: Results

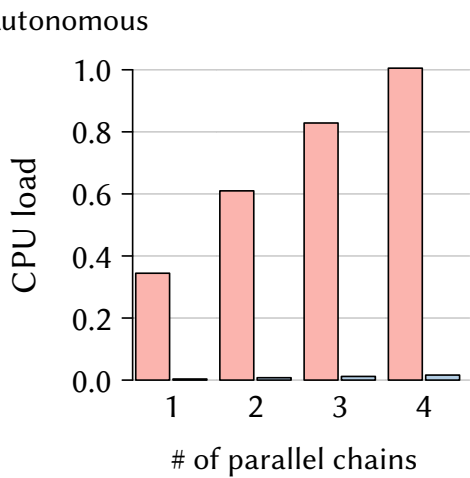


Accelerator Chains: Results

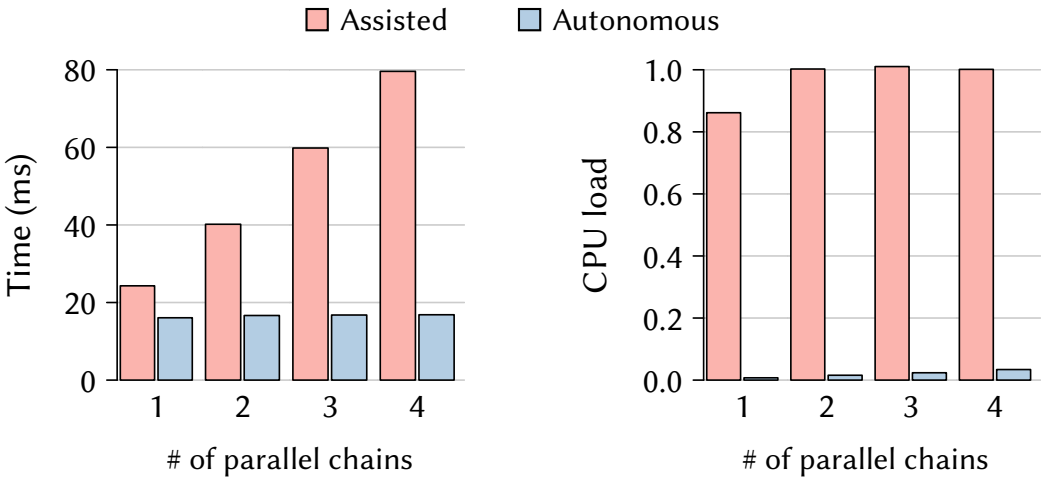


Accelerator Chains: Results



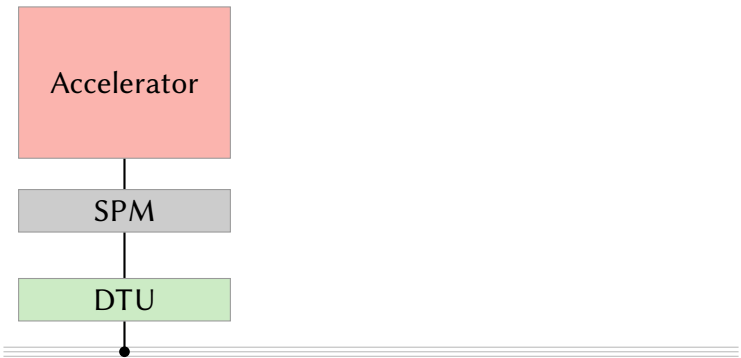


Accelerator Chains: Results (PCIe-like Latency)



Outline

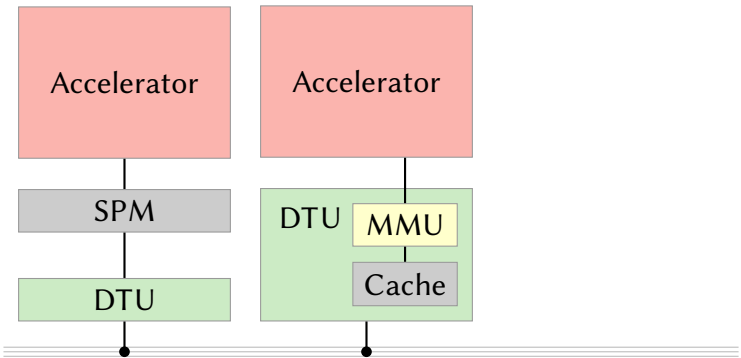
Virtual Memory – Overview



Different PE types:

- No MMU, SPM instead of caches

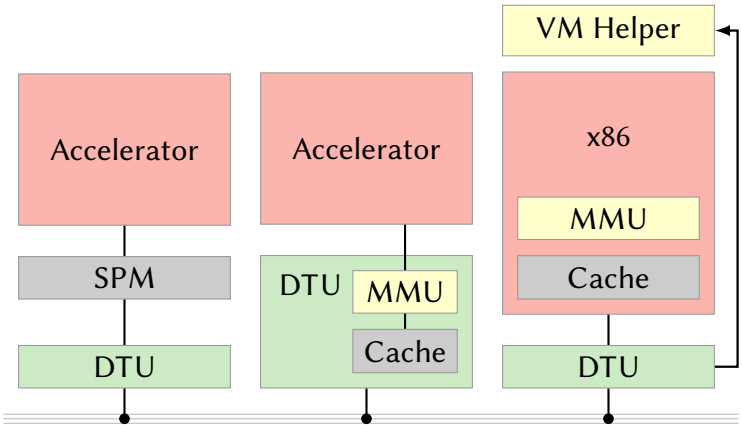
Virtual Memory – Overview



Different PE types:

- No MMU, SPM instead of caches
- MMU+caches provided by DTU

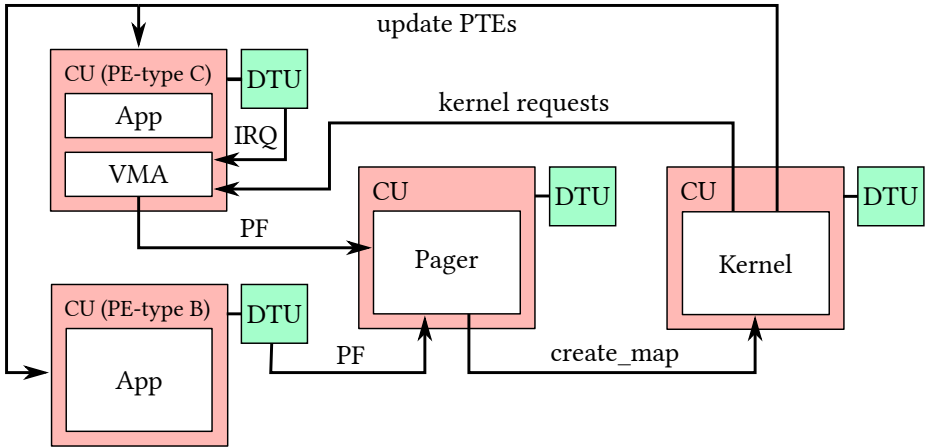
Virtual Memory – Overview



Different PE types:

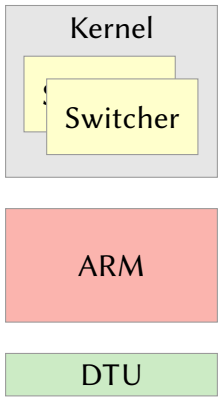
- No MMU, SPM instead of caches
- MMU+caches provided by DTU
- Reuse existing MMU+caches of CU

Page Fault Handling



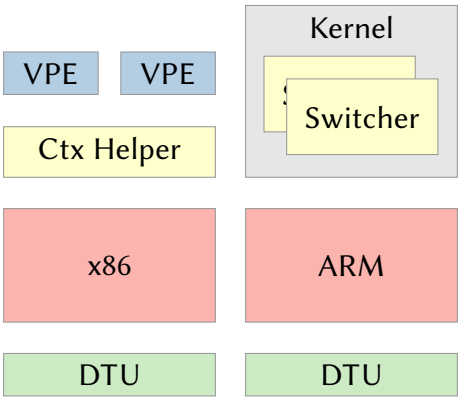
Outline

Context Switching – Overview



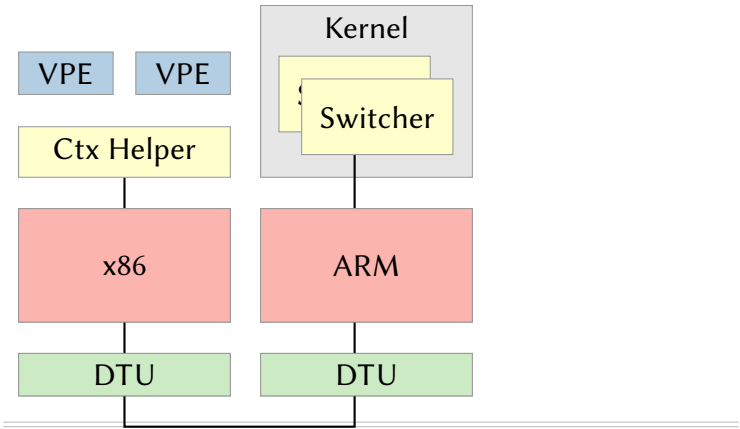
- Kernel handles complex part
  - Schedules and migrates VPEs
  - Initiates context switches

Context Switching – Overview



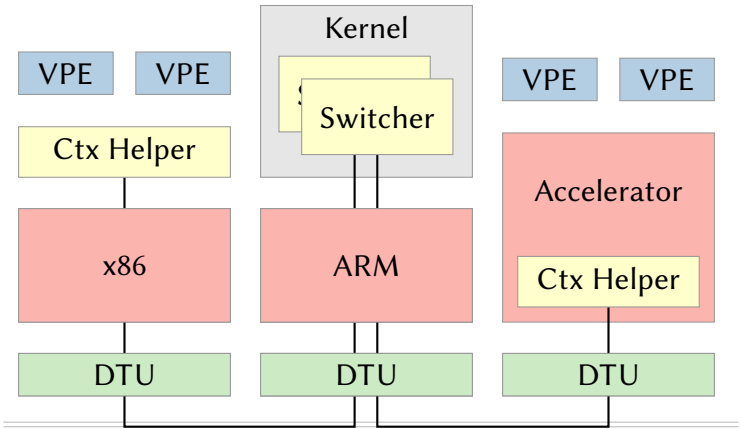
- Kernel handles complex part
  - Schedules and migrates VPEs
  - Initiates context switches
- Helper on user PEs implements save/restore
  - General purpose PEs: Software helper

Context Switching – Overview



- Kernel handles complex part
  - Schedules and migrates VPEs
  - Initiates context switches
- Helper on user PEs implements save/restore
  - General purpose PEs: Software helper

Context Switching – Overview



- Kernel handles complex part
  - Schedules and migrates VPEs
  - Initiates context switches
- Helper on user PEs implements save/restore
  - General purpose PEs: Software helper
  - Accelerator PEs: Helper implemented in hardware as part of ASM

Context Switching with Direct Communication

- How to determine whether recipient is running?
  - DTU knows running VPE and recipient of communication
  - DTU reports error if recipient is not running

Context Switching with Direct Communication

- How to determine whether recipient is running?
  - DTU knows running VPE and recipient of communication
  - DTU reports error if recipient is not running
- How to deliver the message if recipient is not running?
  - Message is forwarded via the kernel
  - Kernel schedules recipient and delivers message

Context Switching with Direct Communication

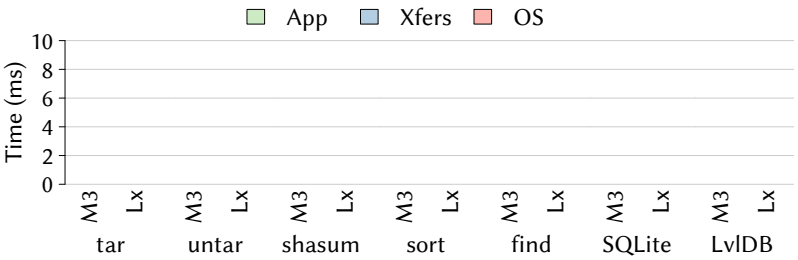
- How to determine whether recipient is running?
  - DTU knows running VPE and recipient of communication
  - DTU reports error if recipient is not running
- How to deliver the message if recipient is not running?
  - Message is forwarded via the kernel
  - Kernel schedules recipient and delivers message
- How does the kernel know what VPEs are doing?
  - Activities send idle notification
  - Only if compatible VPE is ready

Outline

Experimental Setup

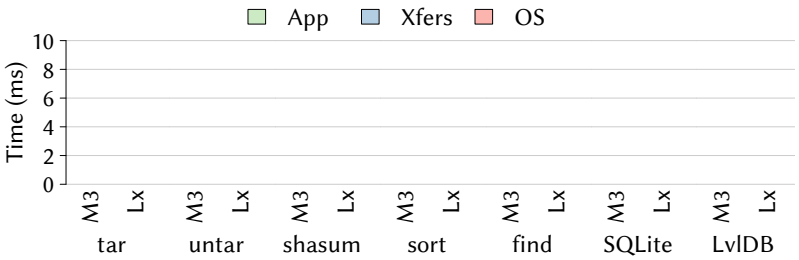
- Evaluation platform is gem5
- Each general-purpose PE has out-of-order x86-64 core @ 3GHz, 32+32 KiB L1 cache, 256 KiB L2 cache
- Accelerator PEs are clocked with 1GHz
- DRAM clocked with 1GHz
- Short running, but representative benchmarks

Linux Application Workloads

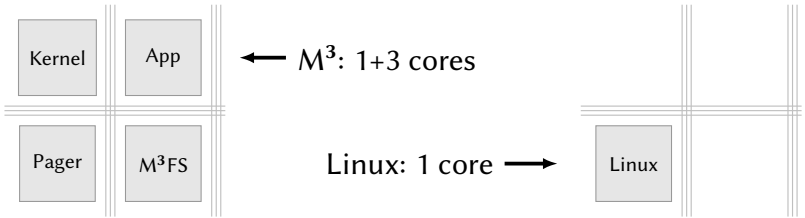


- M^3 vs. Linux 4.10
- Traced on Linux, replayed on M^3
- M^3FS vs. Linux tmpfs

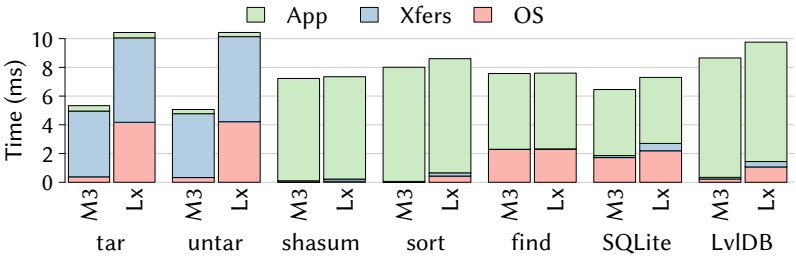
Linux Application Workloads



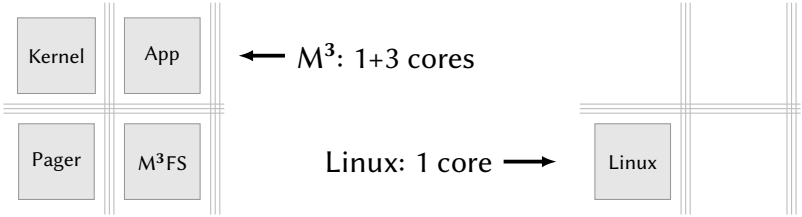
- M³ vs. Linux 4.10
- Traced on Linux, replayed on M³
- M³FS vs. Linux tmpfs



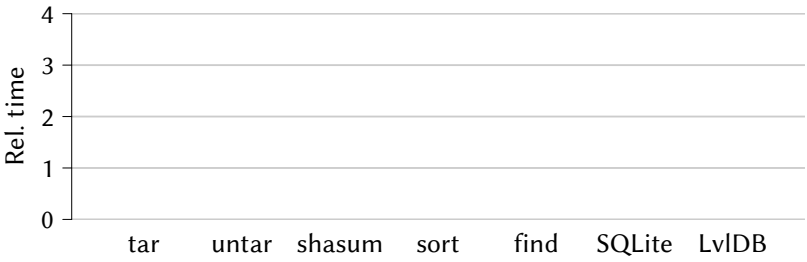
Linux Application Workloads



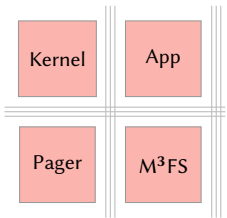
- M³ vs. Linux 4.10
- Traced on Linux, replayed on M³
- M³FS vs. Linux tmpfs

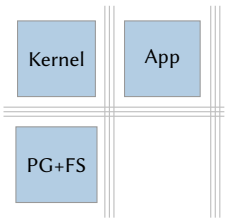


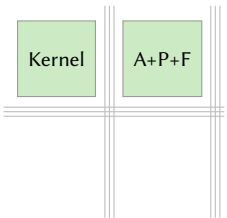
PE Sharing

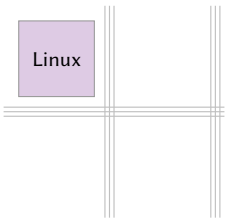


- M³ vs. Linux 4.10
- M³ shares user PEs in different ways
- Baseline is 1+3 PEs

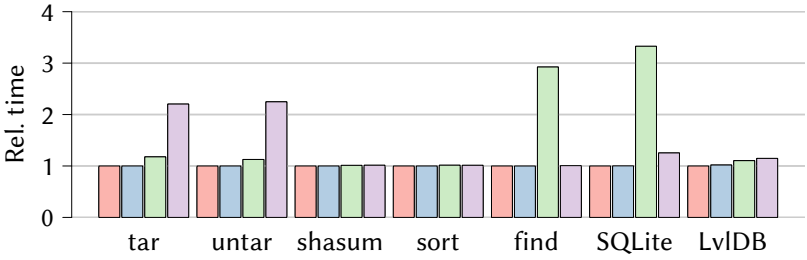




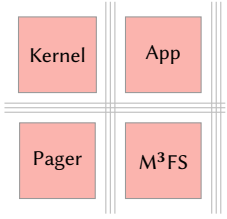


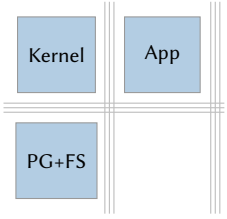


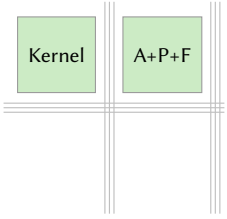
PE Sharing

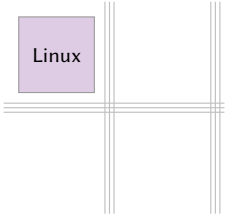


- M^3 vs. Linux 4.10
- M^3 shares user PEs in different ways
- Baseline is 1+3 PEs



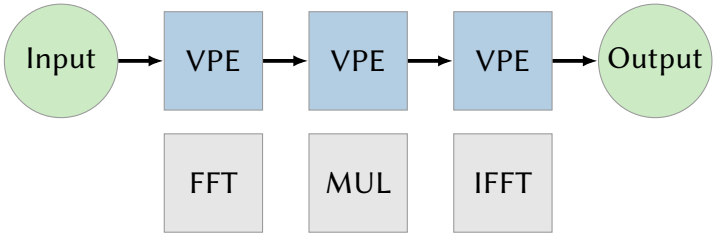




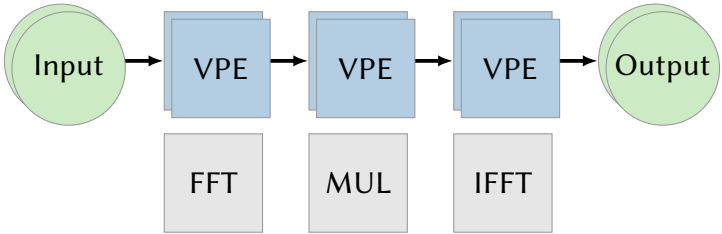


Accelerator Sharing

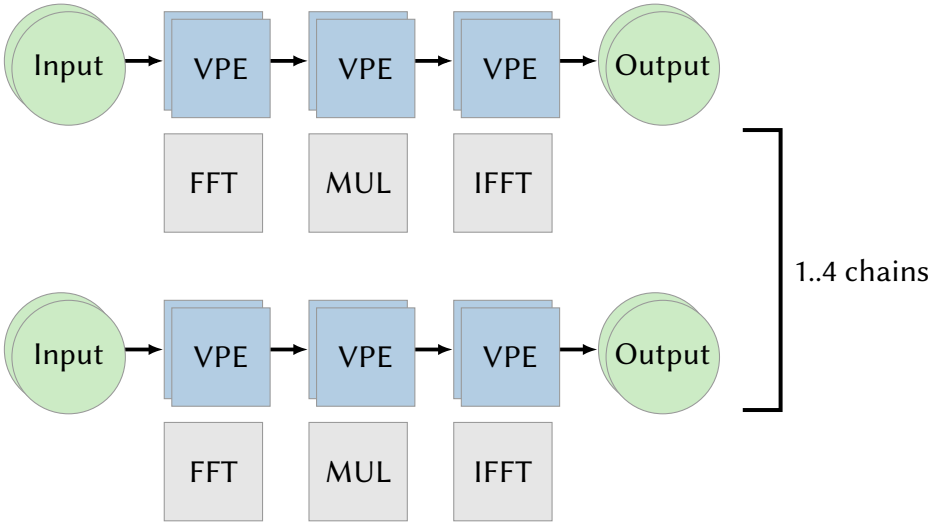
Accelerator Sharing



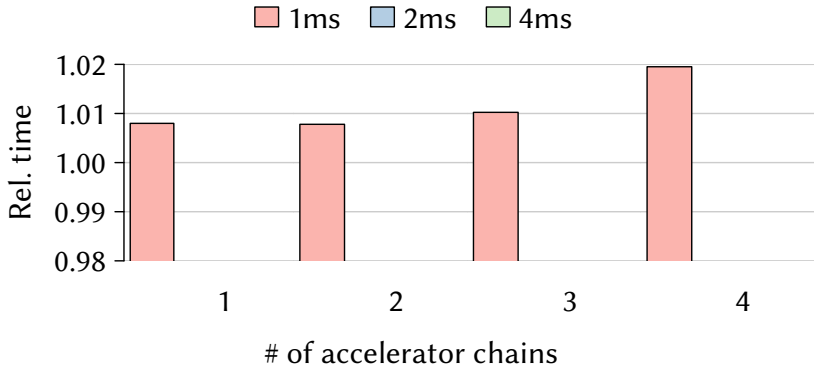
Accelerator Sharing



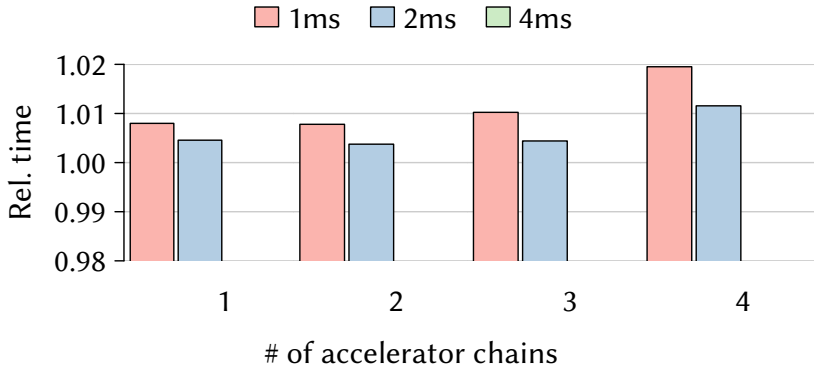
Accelerator Sharing



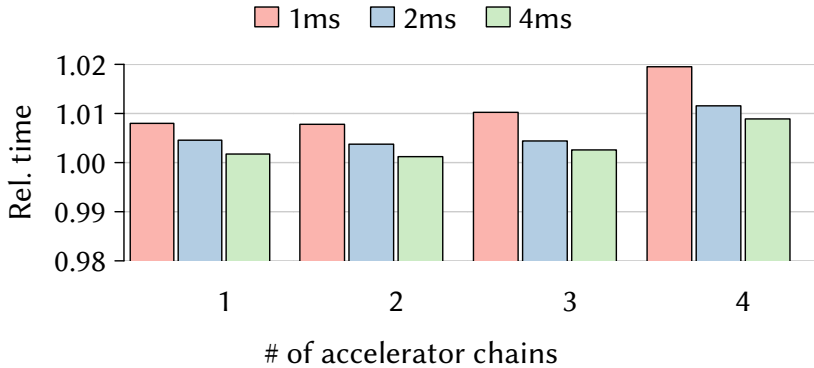
Accelerator Sharing



Accelerator Sharing



Accelerator Sharing



Evaluation Summary

- Comparable application performance
- Superior performance for data-intensive applications
- Accelerators can run autonomously, causing almost no CPU load
- Accelerators can be shared with minimum overhead

Future Work

- Scaling to larger systems pursued by Matthias Hille
  (runs 512 applications with a parallel efficiency of 75%, using 11% for the OS [1])
- Core-local context switching and IPC
- Other accelerators: FPGAs, GPUs, ...

[1] SemperOS: Distributed Capability System, USENIX ATC'19

Conclusion

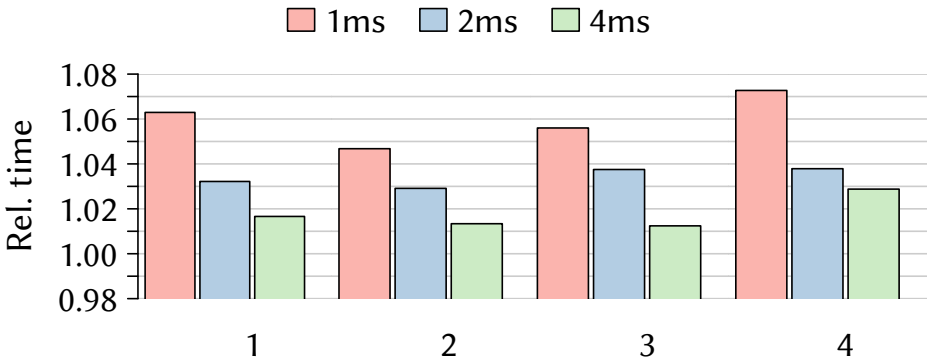
- M^3 uses a hardware/operating-system co-design
- DTU introduces common interface for all CUs
- Allows to integrate all (untrusted) CUs as first-class citizens
- Access to OS services for all CUs
- M^3 uses the same concepts for all CUs
- Allows simple management of complex systems

More Information

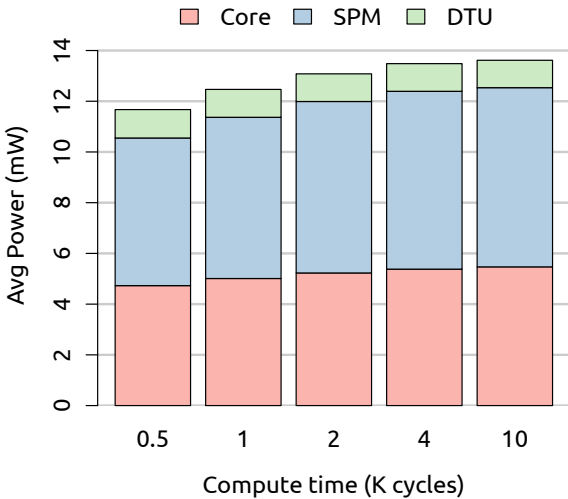
- **M³: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores**
  Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis
  ASPLOS 2016
- **M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication**
  Nils Asmussen, Michael Roitzsch, and Hermann Härtig
  USENIX ATC 2019
- **SemperOS: Distributed Capability System**
  Matthias Hille, Nils Asmussen, Pramod Bhatotia, and Hermann Härtig
  USENIX ATC 2019

Backup Slides

Accelerator Sharing (PCIe)



DTU Power Consumption



DTU Size

Module	**Area** (mm^2)
PE	0.476 864
├ SPM (32 KiB)	├ 0.211 805
├ Xtensa LX5	├ 0.185 259
└ DTU	└ 0.0798
├ Memory (8 EPs)	├ 0.060 991
└ Logic	└ 0.018 809

Comparison:

- Single Xtensa core has
  $\sim$ 50000 gates
- Single x86 core (haswell) has
  $\sim$ 100 Million gates

Software Complexity

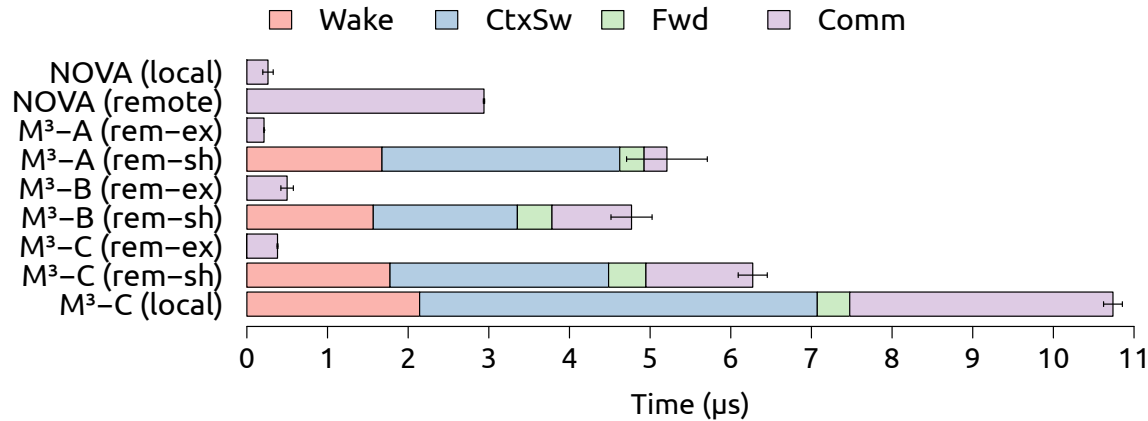
Component	SLOC
Kernel	5398
├ PEs & VPEs	1839
├ Syscalls	970
├ Capabilities	676
├ Remote control	658
└ Memory	551
Pager	852
M^3FS	1794
Pipe server	619

(a) Complexity of kernel and servers

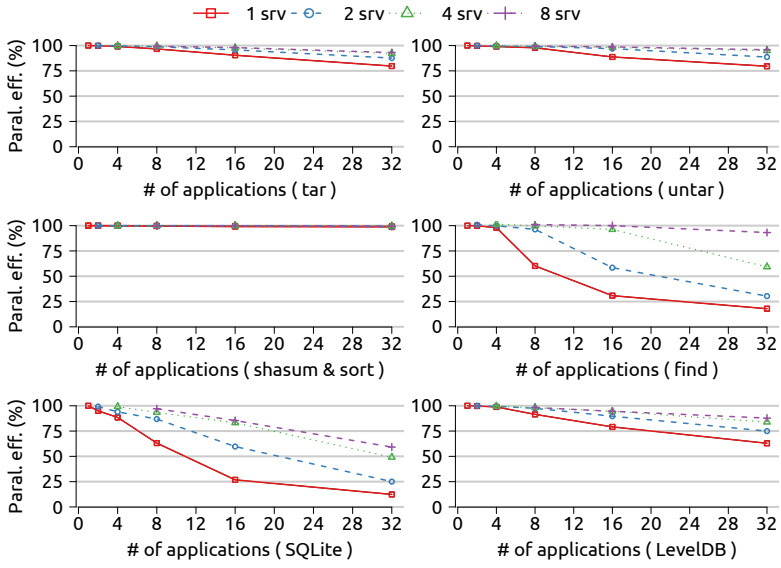
Component	SLOC
CU-specific helper	1049
├ x86-64-specific code	362
├ ARMv7-specific code	149
├ RCTMux	176
└ VMA (x86-64)	301
Base libraries	5204
├ x86-64-specific code	209
└ ARMv7-specific code	178
libm3	5843

(b) Complexity of support components

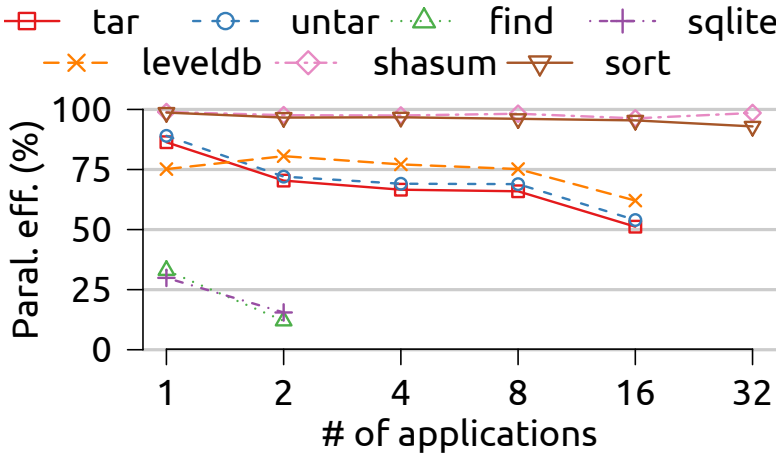
Context Switching Microbenchmark



Scalability with Dedicated OS Service PEs



Scalability with PE Sharing



Stream Processing ASM