Microkernel Construction

Case Study: $M^3$

Nils Asmussen

May 28th 2020
Motivation

- Microkernel-based systems have proven valuable for several objectives
  - Security
  - Robustness
  - Real time
  - Flexibility

- Recently, new challenges are coming from the hardware side
  - Heterogeneous systems
  - Third-party components
  - Security issues of complex general-purpose cores
Heterogeneous Systems

- Demanded by performance and energy requirements
- Big challenge for OSes: single shared kernel on all cores does no longer work
- OSes need to be prepared for processing elements with different feature sets
Third-party Components

- Market pressure forces us to integrate third-party components
- We should not trust these components
- Currently, often no isolation between them
- Bug in such a component can compromise whole system (see Broadcom incident)
Security Issues of Complex General-purpose Cores

- 20 known attacks (and counting …)
- Allow to leak private data, sometimes bypassing all security measures of the core
- Mitigations exist, but these are complex and costly
- These security holes have been lurking in CPUs for many years
- Should we still trust these complex cores to properly enforce the isolation between different software components?
Microkernel-based System as Foundation

Diagram:
- Microkernel
  - Application
    - Service
  - Application
    - Service
  - Core
  - Core
  - Core
Microkernel-based System as Foundation

Application

Service

Management

Microkernel

Core

Core

Core
Microkernel-based System as Foundation

Application

Service

Management

Microkernel

Enforcement

Core

Core

Core
Microkernel-based System as Foundation

Application

Service

Management

Microkernel

Enforcement

Core

Core

Core

FPGA

TPU

GPU
Outline

1 The New System Architecture
2 Prototype Platforms
3 Isolation and Communication
4 Operating System Overview
5 Capabilities
6 OS Services and Accelerators
7 Evaluation
Core

GPU

TPU

Core

Core

FPGA

Key ideas:

TCU as new hardware component
Kernel on dedicated tile
Kernel manages, TCU enforces

Hardware/Operating System Co-Design
Hardware/Operating System Co-Design

Key ideas:
- TCU as new hardware component
- Kernel on dedicated tile
- Kernel manages, TCU enforces
Hardware/Operating System Co-Design

Key ideas:

- TCU as new hardware component
Hardware/Operating System Co-Design

Key ideas:

- TCU as new hardware component
Key ideas:
- TCU as new hardware component
- Kernel on dedicated tile
Hardware/Operating System Co-Design

Key ideas:
- TCU as new hardware component
- Kernel on dedicated tile
- Kernel manages, TCU enforces
Hardware/Operating System Co-Design

Hardware challenges:
- Heterogeneity:
  - Uniform interface

Untrusted HW comp.:
Protected by TCU

Side channels:
Physical isolation
Hardware/Operating System Co-Design

Hardware challenges:
- Heterogeneity:
  - Uniform interface
- Untrusted HW comp.:
  - Protected by TCU
Hardware/Operating System Co-Design

Hardware challenges:

- Heterogeneity:
  - Uniform interface
- Untrusted HW comp.:
  - Protected by TCU
- Side channels:
  - Physical isolation
Linux

- $M^3$ runs on Linux using it as a virtual machine
- A process simulates a tile, having two threads (CPU + TCU)
- TCUs 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
- Modular platform for computer architecture research
- Supports various ISAs (x86, ARM, Alpha, RISC-V, …)
- Provides detailed CPU and memory models
- Cycle-accurate simulation
- Added TCU model to gem5
- Added hardware accelerators
WIP: FPGA-based prototype
Outline

1. The New System Architecture
2. Prototype Platforms
3. Isolation and Communication
4. Operating System Overview
5. Capabilities
6. OS Services and Accelerators
7. Evaluation
TCU-based isolation:
- Additional protection layer
TCU-based isolation:
- Additional protection layer
TCU-based isolation:
- Additional protection layer
- Only kernel tile can establish communication channels
Isolation

TCU-based isolation:
- Additional protection layer
- Only kernel tile can establish communication channels
- User tiles can only use established channels
TCU provides *endpoints* to:

- Access memory (contiguous range, byte granular)
TCU provides endpoints to:

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

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

1 The New System Architecture
2 Prototype Platforms
3 Isolation and Communication
4 Operating System Overview
5 Capabilities
6 OS Services and Accelerators
7 Evaluation
OS Design

- **M³**: Microkernel-based system for heterogeneous manycores (or L4 ± 1)
- Implemented from scratch in Rust and C++
- Drivers, filesystems, etc. implemented on user tiles
- Kernel manages permissions, using capabilities
- TCU enforces permissions (communication, memory access)
- Kernel is independent of other tiles
M³ System Call

User tile

App

TCU

Kernel tile

Kernel

TCU

R
M³ System Call
\(M^3\) System Call

User tile

Kernel tile

App

Kernel
$M^3$ System Call
M³ System Call

User tile

App

TCU

Kernel tile

Kernel

TCU
Outline

1 The New System Architecture
2 Prototype Platforms
3 Isolation and Communication
4 Operating System Overview
5 Capabilities
6 OS Services and Accelerators
7 Evaluation
Overview
Capabilities

$M^3$ capabilities:

- Send: send messages to a receive EP
- Receive: receive messages from send EPs
- Memory: access remote memory via TCU
- Service: create sessions
- Session: exchange caps with service
- Endpoint: configure EPs of own or foreign TCU
- VPE: use a processing element
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
  - → 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

TCU adds Core Mem buffer occupunread EP credits label target

configuration of endpoints to establish a channel

Receiver: VPE1  Sender: VPE2

Send Gate

Send Cap

Mem

EP

Recv Cap

Recv Gate

Mem

EP

unread

occup

cmdreg

Cmdreg

channel

header

data

Credits

Target

Label

TCU

TCU

TCU
Virtual PEs

- \( M^3 \) 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
  - …
VPEs – Examples

Executing ELF-Binaries

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

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

### Lambdas

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

1. The New System Architecture
2. Prototype Platforms
3. Isolation and Communication
4. Operating System Overview
5.Capabilities
6. OS Services and Accelerators
7. Evaluation
OS Service Access for all Processing Element Types

```
sh$ decode in.png | fft | mul | ifft > out.raw
```
OS Service Access for all Processing Element Types

Shell

```
sh$ decode in.png | fft | mul | ifft > out.raw
```
OS Service Access for all Processing Element Types

Shell

User program

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

Challenges:
- OS must provide generic protocols
- Accelerators need support for protocols
OS Service Access for all Processing Element Types

Shell

User program  Input file

sh$ decode in.png | fft | mul | ifft > out.raw
OS Service Access for all Processing Element Types

Shell

User program
Input file

Hardware accelerators for image processing

```
sh$ decode in.png | fft | mul | ifft > out.raw
```
OS Service Access for all Processing Element Types

Shell

User program
Input file

Hardware accelerators for image processing

Output file

User program: decode in.png | fft | mul | ifft > out.raw
OS Service Access for all Processing Element Types

Shell

User program

Input file

Hardware accelerators for image processing

Pipes and output redirect

Output file

sh$ decode in.png | fft | mul | ifft > out.raw
OS Service Access for all Processing Element Types

Shell

User program
Input file
Hardware accelerators for image processing

Pipes and output redirect

Challenges:

Shell

User program
Input file
Hardware accelerators for image processing

Challenges:

28 / 45
OS Service Access for all Processing Element Types

Shell

Pipes and output redirect

User program
Input file
Output file

Hardware accelerators for image processing

Challenges:
- OS must provide generic protocols
OS Service Access for all Processing Element Types

Shell
User program
Input file
Hardware accelerators for image processing
Output file
Pipes and output redirect

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

Challenges:
- OS must provide generic protocols
- Accelerators need support for protocols
Generic Protocols

Client

TCU

Server

TCU

DRAM

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 TCU
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 TCU
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 TCU
Implementation: $M^3$FS – Overview

- $M^3$FS organizes the file’s data in extents
- $M^3$FS 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, $M^3$FS 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: $M^3$FS – 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
- $M^3$FS 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

- **Shared Memory**
- **writer**
- **msg passing**
- **pipeserv**
- **reader**
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
Additions to Accelerator

Scratchpad memory (SPM)

Off-the-shelf accelerators

PE

Accelerator

Accelerator Support Module (ASM):
Interacts with TCU and accelerator
Implements file protocol for input and output channel
ASM assumes that endpoints are setup externally by software
Additions to Accelerator

Scratchpad memory (SPM)

Off-the-shelf accelerators

PE

Accelerator

TCU

Accelerator Support Module (ASM):
Interacts with TCU and accelerator
Implements file protocol for input and output channel
ASM assumes that endpoints are setup externally by software
Additions to Accelerator

Off-the-shelf accelerators

Accelerator Support Module (ASM):
- Interacts with TCU and accelerator
Additions to Accelerator

- Scratchpad memory (SPM)
- PE
- Accelerator
- ASM
- TCU

Off-the-shelf accelerators

Accelerator Support Module (ASM):
- Interacts with TCU and accelerator
- Implements file protocol for input and output channel
Additions to Accelerator

Scratchpad memory (SPM)

Off-the-shelf accelerators

Accelerator Support Module (ASM):
- Interacts with TCU and accelerator
- Implements file protocol for input and output channel
- ASM assumes that endpoints are setup externally by software

PE

Accelerator

ASM

TCU

IN  OUT
Demo
Outline

1. The New System Architecture
2. Prototype Platforms
3. Isolation and Communication
4. Operating System Overview
5. Capabilities
6. OS Services and Accelerators
7. Evaluation
Assisted vs. Autonomous

Diagram showing the flow of data from Input to Output through DMA, SPM, FFT, MUL, IFFT, and Driver.
Assisted vs. Autonomous
Assisted vs. Autonomous

Input

OS
Driver

Input

OS
Driver

Output

Input

Output

FFT
SPM
MUL
SPM
IFFT
SPM

FFT
ASM
SPM
MUL
ASM
SPM
IFFT
ASM
SPM
Assisted vs. Autonomous

Diagram showing the flow of data and operations in assisted and autonomous systems.

- Input
- Output
- DMA
- FFT
- SPM
- MUL
- IFFT
- DTU
- OS
- Driver
- ASM

Comparison between assisted and autonomous systems, highlighting differences in data flow and processing.
Accelerator Chains: Evaluation

FFT  MUL  IFFT

1..4 chains
Accelerator Chains: Evaluation

Input $\rightarrow$ VPE $\rightarrow$ VPE $\rightarrow$ VPE $\rightarrow$ Output

FFT $\rightarrow$ MUL $\rightarrow$ IFFT
Accelerator Chains: Evaluation

1..4 chains
Accelerator Chains: Results

Time (ms)

<table>
<thead>
<tr>
<th># of parallel chains</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

CPU Load

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
</tr>
<tr>
<td>0.2</td>
</tr>
<tr>
<td>0.4</td>
</tr>
<tr>
<td>0.6</td>
</tr>
<tr>
<td>0.8</td>
</tr>
<tr>
<td>1.0</td>
</tr>
</tbody>
</table>

# of parallel chains

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
</tr>
<tr>
<td>45</td>
</tr>
</tbody>
</table>

Assisted

Autonomous
Accelerator Chains: Results

![Bar chart showing time (ms) vs. # of parallel chains for Assisted and Autonomous modes.](chart.png)
Accelerator Chains: Results

![Graph 1: Time (ms) vs. # of parallel chains](image1)

![Graph 2: CPU load vs. # of parallel chains](image2)
Accelerator Chains: Results (PCIe-like Latency)

![Graph showing time and CPU load for different numbers of parallel chains. The graph compares Assisted and Autonomous modes.](image)
Accelerator Chains: Results (PCIe-like Latency)

- **Time (ms)** vs **# of parallel chains**
  - Assisted (red)
  - Autonomous (blue)

- **CPU load** vs **# of parallel chains**
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

M³: 1+3 cores

Linux: 1 core
Linux Application Workloads

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

$M^3$: 1+3 cores

Linux: 1 core
Ongoing Work at the Barkhausen Insitut

- Tile sharing among multiple applications
- Hardware implementation (FPGA and silicon)
- Connected devices with remote attestation
Conclusion

- $M^3$ uses a hardware/operating-system co-design
- TCU introduces common interface for all processing elements (PEs)
- Allows to integrate all (untrusted) PEs as first-class citizens
- Access to OS services for all PEs
- 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