M³ – Microkernel-based System for Heterogeneous Manycores

Nils Asmussen

MKC, 06/29/2017
Outline

1. Introduction
2. Data Transfer Unit
3. Prototype Platforms
4. M³
5. Summary
Motivation

Intel Knights Landing
Motivation

Intel Knights Landing

Qualcomm Snapdragon 810
What’s the Problem?

- **DSP**
- **ARM big**
- **Audio Decoder**
- **Intel Xeon**
- **FPGA**
- **ARM LITTLE**
- **Intel Xeon**
- **DSP**

**Intel Xeon**

**ARM**

**FPGA**

**Audio**

**Decoder**

**Intel Xeon**

**ARM LITTLE**
What’s the Problem?

- DSP
- ARM big
- Audio Decoder
- Intel Xeon Kernel
- Intel Xeon Kernel
- FPGA
- ARM LITTLE
What’s the Problem?
What’s the Problem?

- ARM big
  - Kernel
- Intel Xeon
  - Kernel
- ARM LITTLE
  - Kernel
- Intel Xeon
  - Kernel
The Goal

Treat all compute units (CU) as first-class citizens:

1. Run untrusted code without causing harm
2. Access operating system services
3. Interact as the master with other CUs
First-class Citizenchip as Enabler

- Pipe communication between arbitrary CUs
- Use parallelism on GPUs for FS operations
- Direct access to accelerators from the net
- …
My Approach

- DSP
- ARM big
- Audio Decoder
- Intel Xeon
- Intel Xeon
- FPGA
- DSP
- ARM LITTLE
My Approach
My Approach

- **Intel Xeon**
- **ARM big**
- **Audio Decoder**
- **Intel Xeon**
- **FPGA**
- **DSP**
- **ARM LITTLE**
My Approach
My Approach

- PE
- App
- Mem
- DTU
- PE
- Kernel
- Mem
- DTU
- PE
- App
- Mem
- DTU
- PE
- App
- Mem
- DTU
- PE
- App
- Mem
- DTU
- PE
- App
- Mem
- DTU
- PE
- App
- Mem
- DTU
- PE
- App
- Mem
- DTU
My Approach
Outline

1. Introduction
2. Data Transfer Unit
3. Prototype Platforms
4. M^3
5. Summary
Data Transfer Unit

- Supports memory access and message passing
- Provides a number of endpoints
- Each endpoint can be configured for:
  1. Accessing a piece of memory (contiguous range, byte granular)
  2. Receiving messages into a ringbuffer
  3. Sending messages to a receiving endpoint
- Configuration only by kernel, usage by application
- Direct reply on received messages
- Credits are used to prevent DoS attacks
Data Transfer Unit – Example
Data Transfer Unit – Example
Data Transfer Unit – Example
Tomahawk 2 and 4

- Cores attached to NoC with DTU
- No privileged mode
- No MMU, no caches, but SPM
- T2: simple DTU; T4: most features
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-system architecture research
- Supports various ISAs (x86, ARM, Alpha, SPARC, ...)
- Cycle-accurate simulation
- Has an out-of-order core model
- We built a DTU for gem5
- Support for caches and virtual memory
- We also added hardware accelerators
gem5 – Example Configuration
Outline

1. Introduction
2. Data Transfer Unit
3. Prototype Platforms
4. M³
5. Summary
M$^3$ uses ideas from L4, but implemented from scratch
Provides mechanisms for PEs, memory and communication
Drivers, filesystems, . . . are implemented as services
Kernel manages permissions, using capabilities
DTU enforces permissions (communication, memory access)
Kernel has no ISA-specific code
M³ 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
- **VPE**: execute code on 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
Communication Overview

Kernel: PE0

Receiver: PE1

Sender: PE2

Configuration of endpoints to establish a channel
Communication

- System call to create send cap for specific EP and VPE
- Typically created for own VPE
- Binds a label to the send cap for secure identification
- Can then be delegated to someone else
- To use a send cap, an DTU EP needs to be configured
- The kernel provides a syscall for that purpose
- This indirection allows EP multiplexing and blocking
Memory

- Syscall `reqmem` can be used to request remote memory
- Like with send caps, an EP needs to be configured to access it
- Afterwards, the DTU can be instructed to read/write
- Behaves like RDMA (no SW involved)
- Syscall `derivemem` to create a subset of mem cap
VPEs

- Syscall `createvpe` to create a new VPE
- The kernel assigns the VPE to a suitable PE, if possible
- App gets VPE cap and memory cap for the whole PE memory
- Syscall to start it and wait for its completion (`exit`)
- Can exchange capabilities with the VPE
  - before starting it: e.g., delegate inputs
  - after it’s finished: e.g., obtain results
VPEs – Examples

### Executing ELF-Binaries

```c
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));
    cout << "Done reading!\n";
});
Filesystem: *m3fs*

- Filesystem service is implemented outside of kernel
- m3fs is (currently) an in-memory filesystem
Filesystem: *m3fs*

- Filesystem service is implemented outside of kernel
- `m3fs` is (currently) an in-memory filesystem
Filesystem: \textit{m3fs}

- Filesystem service is implemented outside of kernel
- \textit{m3fs} is (currently) an in-memory filesystem

![Diagram showing the relationship between PE1, PE2, and PE3 with m3fs, App, and kernel]
Filesystem: *m3fs*

- Filesystem service is implemented outside of kernel
- m3fs is (currently) an in-memory filesystem
Filesystem: *m3fs*

- Filesystem service is implemented outside of kernel
- m3fs is (currently) an in-memory filesystem

![Diagram of m3fs architecture]

PE1

m3fs

PE2

App

PE3

kernel

Mem

FS

read/write
Pipes: Direct Pipe

writer → Direct Pipe → reader
Pipes: Direct Pipe

Shared Memory

writer

msg passing

reader
Pipes: Indirect Pipe

Diagram showing the flow of data between writers and readers through an indirect pipe.
Pipes: Indirect Pipe

Shared Memory

writer

msg passing

service

reader
Files on Top of Capabilities

- libm3 provides abstractions to work with files, pipes, …
- Every VPE has a mountspace and file descriptors
- These can be inherited to child VPEs
- Pure convenience: security is based on capabilities

Example

VPE vpe("my vpe");
vpe.fds()->set(STDIN_FD, pipe_fd);
vpe.fds()->set(STDOUT_FD, VPE::self().fds()->get(STDOUT_FD));
vpe.obtain_fds();
vpe.run_async([] { 
    File *in = VPE::self().fds()->get(STDIN_FD);
    // ...
});
Demo
Ongoing Work

- Multiple instances of the kernel/services (by Matthias Hille)
- Port of the musl C library (by Sherif Shehabaldin)
- Integration of a NIC (by Georg Kotheimer)
- Integration of an IDE controller (by Lukas Landgraf)
DTU+M³ is a new point in the design space for systems
Supports untrusted code and OS services on all PEs
Supports direct communication between all PEs
DTU abstracts heterogeneity and provides common interface
M³ kernel controls DTUs remotely
M³ is available at https://github.com/TUD-OS/M3