M³ – Microkernel-based System for Heterogeneous Manycores

Nils Asmussen

MKC, 06/29/2017
Heterogeneous Systems
Heterogeneous Systems
Heterogeneous Systems

[Diagram showing various components and technologies related to heterogeneous systems]
Why?

- FPGA-based memcached 16x better in performance per watt than Atom CPU [1]
- Machine learning 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
The Problem for OSes
The Problem for OSes

- Intel Xeon
- ARM big
- Audio Decoder
- Intel Xeon Kernel
- ARM LITTLE

- DSP
- FPGA
- Kernel
The Problem for OSes
The Problem for OSes
Making Accelerators More First-Class

- File system access for GPUs [1]
- Network access for GPUs [2]
- Access to OS services from FPGAs [3,4]
- Computing directly on the SSD [5]

[1] GPUfs: integrating a file system with GPUs, ASPLOS'13
[2] GPUnet: Networking Abstractions for GPU Programs, OSDI'14
[3] ReconOS: An operating system approach for reconfigurable computing, MICRO'14
Can we design a system that treats all compute units (CU) as *first-class citizens* from the beginning?

1. Run untrusted code without causing harm
2. Access operating system services
3. Context switching support
4. Direct communication without involving CPU
Outline

1. Overall System Design
2. Prototype Platforms
3. Capabilities
4. OS Services
5. Context Switching
6. Evaluation
Outline

1. Overall System Design
2. Prototype Platforms
3. Capabilities
4. OS Services
5. Context Switching
6. Evaluation
My Approach – Hardware

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

My Approach – Hardware

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

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

Asmussen et al.: M3: A Hardware/OS Co-Design to Tame Heterogeneous Manycores, ASPLOS’16
Data Transfer Unit

- Supports memory access and message passing
- Provides a number of endpoints
- Each endpoint can be configured for:
  1. Accessing memory (contiguous range, byte granular)
  2. Receiving messages into a receive buffer
  3. Sending messages to a receiving endpoint
- Direct reply on received messages
- Configuration only by kernel, usage by application
- Credit system to prevent DoS attacks
OS Design

- **M³**: Microkernel-based system for het. manycores (or L4 ±1)
- Implemented from scratch
- Drivers, filesystems, ... are implemented on top
- Kernel manages permissions, using capabilities
- DTU enforces permissions (communication, memory access)
- Kernel is independent of other CUs in the system
M³ System Call
Outline

1. Overall System Design
2. Prototype Platforms
3. Capabilities
4. OS Services
5. Context Switching
6. Evaluation
Tomahawk 2 and 4

PEs have no OS support:
- No privileged mode
- No MMU, no caches, but SPM
- T2: simple DTU; T4: most features
M³ 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, SPARC, ...)
- Provides detailed CPU and memory models
- Cycle-accurate simulation
- We built a DTU for gem5
- We also added hardware accelerators
gem5 – Example Configuration
Overview

Overall System Design
Prototype Platforms
Capabilities
OS Services
Context Switching
Evaluation

Diagram:

- VPE 1
  - 0
  - 2
- VPE 2
  - 0
  - 2
  - 1

Kernel

VPE 1

VPE
SGate
RGate
VPE

VPE 2

VPE

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**: 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
  - → 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

Kernel: PE0

Receiver: PE1

Sender: PE2

DTU adds

configuration of endpoints to establish a channel
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 cap. and memory cap.
- 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);
### Executing ELF-Binaries

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

### Asynchronous Lambdas

```cpp
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";
});
```
Outline

1. Overall System Design
2. Prototype Platforms
3. Capabilities
4. OS Services
5. Context Switching
6. Evaluation
File Protocol

- File protocol is used for all file-like objects
- Simple for accelerators, yet flexible for software
- Software uses POSIX-like API on top of the protocol
- Server provides client access to data by configuring client’s memory endpoint
- Client accesses data via DTU, without involving others
- \texttt{req(in/out)} requests next input/output piece and implicitly commits previous piece
- \texttt{commit(nbytes)} commits \texttt{nbytes} of previous piece
- Receiving \texttt{resp(n, 0)} indicates EOF
Implementation: m3fs – Overview

- **m3fs** is an in-memory file system
- m3fs organizes the file’s data in extents
- 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

writer -> Pipe -> reader
Implementation: Pipe – Overview

- **writer**
- **Shared Memory**
- **msg passing**
- **reader**
- **pipeserv**
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
Accel. Example: Stream Processing

- Accelerator works on scratchpad memory
- Input data needs to be loaded into scratchpad
- Result needs to be stored elsewhere
Accel. Example: Stream Processing

- Accelerator works on scratchpad memory
- Input data needs to be loaded into scratchpad
- Result needs to be stored elsewhere
Shell Integration

- $M^3$ allows to use accelerators from the shell:
  preproc | accel1 | accel2 > output.dat
- Shell connects the EPs according to stdin/stdout
- Accelerators work autonomously afterwards
- Requires about 30 additional lines in the shell
Demo
# Outline

1. Overall System Design
2. Prototype Platforms
3. Capabilities
4. OS Services
5. Context Switching
6. Evaluation
Context Switching

Kernel
  CtxSw
CU: ARM
  DTU
CU: Accel
  DTU
Context Switching

interrupt request
Context Switching

[Diagram showing a system design with nodes labeled App, Kernel, CU: x86, CU: ARM, CU: Accel, DTU, RCTMux, and CtxSw. The diagram illustrates the flow of interrupt request through the system.]
Communication with Suspended VPEs

- If a VPE is suspended, communication channels stay valid.
- Each DTU knows the ID of the currently running VPE.
- Messages contain the target VPE ID.
- If these do not match, DTU responds with an error.
- In this case, the sender lets the kernel *forward* the message.
- Kernel will resume the VPE and afterwards transmit the message on behalf of the sender.
Computing vs. Idling

- How does the kernel know what VPEs are doing?
- VPEs communicate directly, without involving the kernel and wait for the next msg via DTU
- The kernel asks VPEs to report idling, if other VPEs are ready
- As soon as a VPE starts to idle, it checks whether it should report that
- If so, the VPE waits for the time chosen by the kernel and performs a system call afterwards
Experimental Setup

- Evaluation platform is gem5
- Each general-purpose PE has x86_64 core @ 3GHz, 32+32 KiB L1 cache, 256 KiB L2 cache
- Accelerator PEs are clocked with 1GHz
- DRAM (DDR3_1600_8x8) clocked with 1GHz
- Short running, but representative benchmarks
Accelerator Chaining – Variants

Overall System Design
Prototype Platforms
Capabilities
OS Services
Context Switching
Evaluation

Assisted
Accelerator Chaining – Variants

**Assisted**

- Input
- OS
- Logic
- DMA
- Accel
- SPM
- Output

**Autonomous**

- Input
- Shell
- DTU
- Logic
- Accel
- SPM
- Output
Accelerator Chaining – Results

- Assisted
- Autonomous

Time (ms):

- 4GB
- 2GB
- 1GB
- 0.5GB

1 Accel.
2 Accel.
4 Accel.
8 Accel.
Accelerator Chaining – Results

- Assisted
- Autonomous

Time (ms)

- 4GB
- 2GB
- 1GB
- 0.5GB

1 Accel., 2 Accel., 4 Accel., 8 Accel.
Accelerator Chaining – Results

CPU time (rel)

Assisted
Autonomous

4GB 2GB 1GB
0.5GB
1 Accel.
2 Accel.
4 Accel.
8 Accel.

0.0 0.5 1.0

1 Accel. 2 Accel. 4 Accel. 8 Accel.
Accelerator Chaining – Results

- Assisted
- Autonomous

<table>
<thead>
<tr>
<th>Accelerator Count</th>
<th>1 Accel.</th>
<th>2 Accel.</th>
<th>4 Accel.</th>
<th>8 Accel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Size</td>
<td>4GB</td>
<td>2GB</td>
<td>1GB</td>
<td>0.5GB</td>
</tr>
<tr>
<td>CPU Time (rel)</td>
<td>1.0</td>
<td>0.7</td>
<td>0.5</td>
<td>0.3</td>
</tr>
</tbody>
</table>

- 1 Accel.: 4GB, 2GB, 1GB, 0.5GB
- 2 Accel.: 4GB, 2GB, 1GB, 0.5GB
- 4 Accel.: 4GB, 2GB, 1GB, 0.5GB
- 8 Accel.: 4GB, 2GB, 1GB, 0.5GB
Application Performance

- Comparison to Linux 4.10, using tmpfs
- Traced obtained on Linux and replayed on $M^3$
- $M^3$: 3 user PEs; Linux: 1 core (same config)
Application Performance

- Comparison to Linux 4.10, using tmpfs
- Traced obtained on Linux and replayed on M³
- M³: 3 user PEs; Linux: 1 core (same config)
Application Performance

- Comparison to Linux 4.10, using tmpfs
- Traced obtained on Linux and replayed on M³
- M³: 3 user PEs; Linux: 1 core (same config)
PE Sharing

- 3 user PEs: pager, m3fs, app (baseline)
- 2 user PEs: pager+m3fs, app
- 1 user PEs: pager+m3fs+app

![Graph showing relative runtime for different configurations]
PE Sharing

- 3 user PEs: pager, m3fs, app (baseline)
- 2 user PEs: pager+m3fs, app
- 1 user PEs: pager+m3fs+app

<table>
<thead>
<tr>
<th>Command</th>
<th>M³ (3 uPEs)</th>
<th>M³ (2 uPEs)</th>
<th>M³ (1 uPE)</th>
<th>Linux (1 core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tar</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>untar</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>shasum</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>sort</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>find</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SQLite</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LevelDB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ongoing Work

- Multiple instances of the kernel/services (by Matthias Hille)
- Improved network support (by Georg Kotheimer)
- Extension of m3fs for storage devices (by Sebastian Reimers)
Conclusion

- M³ uses a hardware/software co-design
- DTU introduces common interface for all CUs
- Allows to treat all CUs as first-class citizens
- Access to OS services for all CUs
- M³ uses the same concepts for all CUs
- Allows simple management of complex systems