M3 – Microkernel for Minimalist Manycores

Nils Asmussen

MKC, 07/09/2015
Outline

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

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

- Two architectural trends in last years:
  - More heterogeneity
  - More cores, but not all are usable at all times
- How can we integrate arbitrary cores as first-class citizens?
- Do the trends offer new trade-offs for the OS design?
High-level View
High-level View
High-level View
Data Transfer between Cores

Receiver PE

Sender PE

DTU

Mem

ringbuf

Core

DTU added by DTU

Mem

EP

header

data

local

label

remote

credits

remote

label

local

credits

added by DTU

channel
Memory Access
Outline

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

Data Transfer Unit (DTU)
- Instr. SPM
- Data SPM

NoC IF
- Xtensa LX4
- VDSP

Prototype Platforms
- PE
- PE
- DRAM

M3

Summary
Tomahawk 4

- Cycle-accurate simulator, based on SystemC
- Contains the described DTU
- Can instantiate an arbitrary number of PEs
- Each has 64K+64K of scratchpad memory for code+data
- Supports different cores (we have an FFT core)
- Has one DRAM of arbitrary size
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
\( \mu \)-kernel

- Runs on dedicated cores with no applications on top
- Applications run without kernel underneath
- Applications link against \texttt{libm3}
- Kernels responsibilities:
  - PEs: manages them in terms of VPEs (who is put where?)
  - Capabilities of VPEs
  - Establishes channels between DTU endpoints
  - Memory
  - Services
- Caps are protected by kernel, selectors managed by apps
- Normal IPC can’t be used because the kernel is not involved
The syscall `creategate` creates a gate for an endpoint.

- Requires a capability to the VPE, which endpoint is used.
- Typically it is the own VPE, but can be e.g. a child VPE.
- Binds a label to the gate for secure identification.
- Gate cap can be delegated to others.
- Gate cap needs to be programmed on an endpoint first.
- Kernel might defer it until the receiver is ready.
Memory

- The syscall `reqmem` can be used to request global memory
- Memory cap needs to be programmed on an endpoint first
  - → memory capabilities are the same as gate capabilities
- I.e. unlike L4, there are no different kinds of capabilities
- Syscall `derivemem` to create a subset of memory cap
There is a filesystem service running on some PE

The FS uses global memory as the inode and buffer cache

App goes to FS to know where file fragments are stored

FS hands out memory capabilities to the application

App reads/writes directly from/to global memory afterwards
Filesystem – API

- Like in L4Re, the mount tree is application-local
- File is the low level abstraction to read/write files
- FileStream sits on top, uses a buffer and supports formatting
Like in L4Re, the mount tree is application-local
File is the low level abstraction to read/write files
FStream sits on top, uses a buffer and supports formatting

Example

```cpp
VFS::mount("/", new M3FS("m3fs"));

FileRef file("/myfile", FILE_R);
while(file.read(buf, 512) > 0) {
    // ...
}
```
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 the VPE
- Can exchange capabilities with the VPE before starting it
- Syscall to wait until the program called `exit`
VPEs – Examples

Executing ELF-Binaries

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

**Executing ELF-Binaries**

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

**Asynchronous Lambdas**

```c
VPE vpe("test");
MemGate mem = MemGate::create_global(0x1000,
    MemGate::RW);
vpe.delegate(CapRngDesc(mem.sel(), 1));
vpe.run([&mem]() {
    mem.read_sync(buf, sizeof(buf));
    Serial::get() << "Done reading!\n";
});
```
M3 is a different kind of microkernel-based system

- $\mu$-kernel manages remote resources instead of local resources
- $\mu$-kernel and applications run on separate cores
- Applications are linked against a library to get abstractions
- Capabilities are used to manage permissions in the system
- Similar operations with caps, but still different