Driver development for Linux

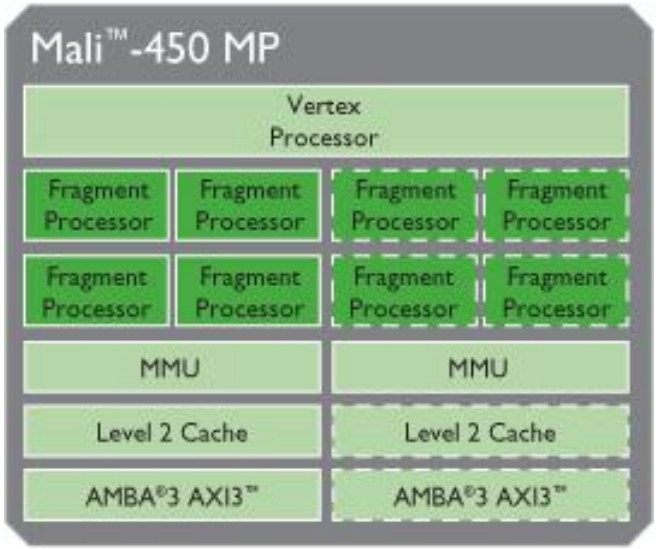
Guest lecture TDT4258

2013.04.18

Ørjan Eide and Mikael Valen-Sendstad

The kernel driver for Mali

- Driver for Mali-400
  - Mali consist of multiple components
    - Geometry
    - Pixel
    - MMU



- The driver is fairly big
  - 25000 code lines kernel driver
  - The user space OpenGL driver is bigger, with about half a million code lines

- http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php

How the Applications use Mali GPU

- OpenGL ES
  - Subset of OpenGL API designed for use on Embedded Systems
  - Now also for use in web browser through WebGl
- Basic flow:
  - Init: Add textures, vertex buffers, shaders etc
  - Set the render state:
    - Textures
    - Vertices
    - Input: vertex shader (translation matrix)
    - Input: pixel shader (lighting params)
    - Call DrawArray() to draw the object (or DrawElements())
  - Call eglSwapBuffers() to put the buffer on screen

How the Applications use Mali GPU

- Immediate mode rendering
  - Draw calls directly to HW
- Defered rendering
  - Batch up draw calls
  - Start GP and PP jobs
- Tiled rendering
  - Rendering completes 1 tile at the time
  - Reduces memory bandwith by caching:
    - Z-read
    - Pixel writes
    - Texture reads

Memory – CPU – GPU

- Computers: Dedicated graphics memory
- Embedded: Shared graphics memory
- Memory latency
- Should graphics memory be cached, by CPU and GPU ?
  - Mapping
  - Cache flushing – Clean, Invalidate
  - Prefetching

Driver architecture

- Most drivers can be built as a kernel module
  - Convenient when developing
  - Allows kernels to support devices without increasing the size of the kernel image
- Linux Driver Model (LDM)
  - Documentation in *Documentation/driver-model/*
  - The model separates the driver from the specification of the device
  - Automatically matches drivers to devices
    - Based on device IDs (for PCI, USB, …) or a device name
      - Drivers register a table of supported devices
    - Makes it possible to support hot-pluggable devices (e.g. USB)
  - Based on the bus structure in the machine

Partitioning: kernel vs user

- There is often a choice of where to put functionality
  - Whole drivers can run in user space
    - libusb makes it easy to create whole drivers for USB devices in user space
  - It is also possible to create whole applications in kernel space
    - But people rarely do that (for good reasons)
- Robustness and security
  - The consequences of a crash in kernel space can be very big
    - If a user space process crashes it doesn't impact the rest of the system
- Only the kernel can have the full system overview
  - Resource sharing (e.g.: memory or hardware access)
- Performance
  - System calls and context switches costs

Linux driver model

Device

```
struct platform_device mydevice = {
.name = ”devicename”,
.id = 0,
.resource = {
                .start = 0x1000,
                .end = 0x2000,
                .name = ”device registers”,
                . flags = IORESOURCE_MEM,
},
};

int  platform_init_func()
{
return platform_device_register(&mydevice);
}
```

Driver

```
int driver_probe(struct platform_device *pdev)
{
/*  Get resources from pdev.
    Probe the HW, check if the dirver is
compatible.
*/
}

struct platform_driver mydriver = {
.probe = &driver_probe,
.remove = &driver_remove,
.driver = {
                .name = ”devicename”,
                .owner = THIS_MODULE,
                .bus = &platform_bus_type,
                .pm = &mydriver_pm_ops,
},
}

int module_init(void)
{
platform_driver_register(&mydriver);
}
```

Debugging

Debugging

- Reproducing the problem will often be the hardest part
  - Problems that occur once every week, on a machine you can't access across town, is very hard to debug
  - Once you have a good reproducer, fixing the problem is usually relatively easy
- Bugs in kernel space can have non-obvious consequences
  - A simple dangling pointer dereference can cause very unpredictable results
- Without visibility of what the driver is doing it is hard to debug
  - There are many ways to gain some visibility
  - Log actions as they happen
    - Could be done simply with **printk**, or with more advanced logging
  - Use a debugger: kgdb, JTAG
  - Pre-emptive debugging built into the driver or kernel
    - Extra checks, asserts, ...

Typical debugging workflow

1. **Reproduce problem**
2. **Gather information**
3. **Try to understand problem**
    1. **If more information is needed:**
        1. **Increase visibility of what is happening**
        2. **Goto 1.**
4. **Fix problem**
5. **Verify fix**
6. **Create test**
    - **Catches error if reintroduced**

Debugging: Debuggers

- Hardware debuggers: JTAG
  - It can be hard to gain visibility of software through software
    - The debugging/tracing can interfere with the driver
      - Quite often the problem disappears as soon as you attach gdb
  - A HW debugger lets you stop the CPU and inspect memory and registers
  - Can also debug interrupt handlers and regions of code running with interrupts disabled

- gdb for the Linux kernel: kgdb
  - Connects to a running kernel over serial port or network
  - Can't do everything you can do with a HW debugger
    - But, you don't need specialized (possibly expensive) hardware

Debugging: Linux

- There are a lot of great debugging aids built into the Linux kernel
- Check «Kernel hacking» in menuconfig
  - Many of these functions helps detection problems early
- Using a kernel with most of these debugging aids enabled can be good during development
  - It will be slower than a ”normal” kernel
    - But it is worth it to discover problems early
  - Some of the debugging aids have only a very small performance impact and can be enabled even for production kernel builds

Debugging: Linux

- kmemleak
  - Tracks all memory allocations and discovers memory leaks
  - *Documentation/kmemleak.txt*

- The built-in Linux lock debugging is great
  - It can be quite hard to do locking correctly
    - Errors can lead to hanging processes, or whole systems
  - CONFIG_DEBUG_MUTEXES, CONFIG_DEBUG_SPINLOCK
    - Does some basic checks for locks
  - CONFIG_PROVE_LOCKING
    - Proves that all observed locking behavior is safe
      - Generates a warning if a deadlock could possibly occur
  - And many more: CONFIG_DEBUG_ATOMIC_SLEEP, CONFIG_DEBUG_RT_MUTEXES, CONFIG_DEBUG_LOCK_ALLOC, ...

Debugging: Linux

- Ftrace/Tracepoints
  - *Documentation/trace/ftrace.txt*
  - *Documentation/trace/tracepoints.txt*
  - Traces events and function calls in the kernel
  - A driver can emit events for its own events

```
$ cd /sys/kernel/debug/tracing
$ echo 1 > /sys/kernel/debug/tracing/tracing_enabled
$ cat /sys/kernel/debug/tracing/trace > /tmp/trace.txt
$ echo 0 > /sys/kernel/debug/tracing/tracing_enabled
```

```
# tracer: function_graph
#
#     TIME        CPU  DURATION                  FUNCTION CALLS
#      |          |    |    |                     |   |   |   |
 3)   0.105 us    |                } /* rcu_enter_nohz */
 3)   1.083 us    |              } /* rcu_irq_exit */
 3)   0.066 us    |              idle_cpu();
 3)               |            tick_nohz_stop_sched_tick() {
 3)               |              ktime_get() {
 3)   1.125 us    |                read_hpet();
 3)   1.626 us    |              }
 3)   0.066 us    |              update_ts_time_stats();
 3)   0.066 us    |              timekeeping_max_deferment();
 3)   0.171 us    |              rcu_needs_cpu();
 3)   0.069 us    |              printk_needs_cpu();
 3)               |            get_next_timer_interrupt() {
 3)   0.096 us    |                _raw_spin_lock();
 3)               |              hrtimer_get_next_event() {
 3)   0.105 us    |                _raw_spin_lock_irqsave();
 3)   0.078 us    |                _raw_spin_unlock_irqrestore();
 3)   1.095 us    |              }
 3)   2.181 us    |            }
 3)               |            hrtimer_start() {
 3)               |              __hrtimer_start_range_ns() {
 3)               |                lock_hrtimer_base() {
 3)   0.099 us    |                  _raw_spin_lock_irqsave();
 3)   0.621 us    |                }
 3)   0.114 us    |                __remove_hrtimer();
 3)   0.201 us    |                enqueue_hrtimer();
 3)   0.078 us    |                _raw_spin_unlock_irqrestore();
 3)   2.841 us    |              }
 3)   3.417 us    |            }
 3) + 10.893 us   |            }
 3) ! 110.298 us  |          } /* irq_exit */
 3) ! 125.199 us  |        } /* smp_call_function_single_interrupt */
```

ARM DS-5

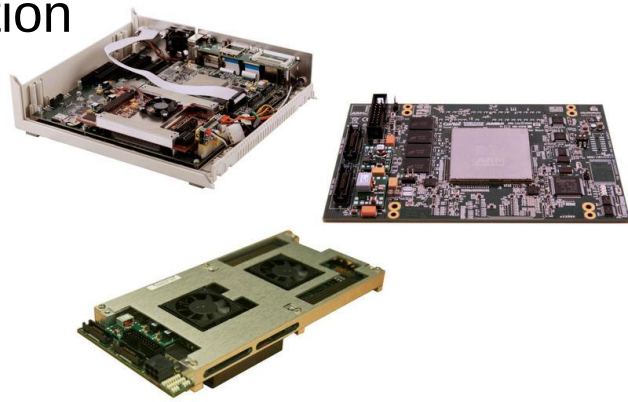
- IDE with built in debugging and trace features
  - Supports HW debugger, DSTREAM
  - And remote debug similar to kgdb

- Streamline
  - Remote debug and trace
  - Trace based on Linux ftrace and tracepoints
  - Uses the Linux perf framework to gather CPU performance counters

- http://www.arm.com/products/tools/software-tools/ds-5/index.php

Driver development without hardware

- Prototype in FPGA
  - Can be used to develop driver before HW is ready
  - Driver development can feed back into HW development
  - Probably slower than real HW
  - Software can contribute to hardware verification



- Software model
  - E.g. ARM Fast Models
    - http://www.arm.com/products/tools/models/fast-models/index.php
  - Can start driver development before even FPGA is available

Questions?