Design and Implementation of an Asymmetric Multiprocessor Environment Within an SMP System

Roberto Mijat, ARM
Santa Clara, October 2, 2007
Agenda

- Design Principle

- ARM11 MPCore™ overview
  - System’s considerations / optimizations

- Implementation
  - Work Plane
  - Data Plane
  - Communication Module

- Usage cases
Design principle

- **Primary CPU**
  - Single processor OS + application + driver for off-loaded API

- **Secondary CPUs**
  - Wait and/or process background work-tasks
  - Primary CPU driver abstracts shared RAM and provides message based API to data plane processor farm
ARM11 MPCore™ Processor

**PPA**

<table>
<thead>
<tr>
<th>90nm process</th>
<th>Speed Opt</th>
<th>Area Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Cells</td>
<td>Advantage-HS</td>
<td>Metro</td>
</tr>
<tr>
<td>Memories</td>
<td>Advantage</td>
<td>Metro</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>620</td>
<td>320</td>
</tr>
<tr>
<td>Area with cache (mm²)</td>
<td>2.54</td>
<td>1.46</td>
</tr>
<tr>
<td>Area without cache (mm²)</td>
<td>1.80</td>
<td>0.90</td>
</tr>
<tr>
<td>Cache size</td>
<td>16K/16K</td>
<td>16K/16K</td>
</tr>
<tr>
<td>Power with cache (mW/MHz)</td>
<td>0.43</td>
<td>0.23</td>
</tr>
<tr>
<td>Power w/o cache (mW/MHz)</td>
<td>0.37</td>
<td>0.18</td>
</tr>
</tbody>
</table>

* Either quoted from fully floor-planned layout/synthesis trials or scaled with respect to process and library performance
ARM11 MPCore Usage Models

SMP – Symmetric Multi-Processing

AMP – Asymmetric Multiprocessing

Shared resources

Operating System

CPU0

Private resources

CPU1

Private resources

Arm11 MPCore supports both models
ARM11 MPCore™ Processor

- Supports both AMP and SMP
  - Each CPU:
    - Boots independently
    - Share physical memory map
    - Elects whether to participate in coherency domain
  - This provides software portability and flexibility

- Provides advanced optimised coherence protocol
  - Advanced Snoop Control Unit
  - Optimized MESI implementation

- Provides integrated GIC
  - Highly configurable Interrupt Distributor/Interfaces
  - Inter-Processor Communication
The MESI coherency protocol

- MESI protocol is a write-invalidate cache protocol
  - When writing to a shared location, the related cache line is invalidated in all caches in L1 memory system

- Coherent cache line attributes: Every cache line is marked with one of the four following states (coded in two bits)
  - **Modified**
    - Coherent cache line is not up to date with main memory
  - **Exclusive**
    - Up to date and no other copies exist
  - **Shared**
    - Coherent cache line which is up to date with main memory
  - **Invalid**
    - This coherent cache line is not present in the cache
MPCore MESI optimizations

The key is to avoid main memory accesses

- **Duplicated Tag RAMs**
  - Stored in Snoop Control Unit
  - Check if requested data is in other MP11 caches without accessing them

- **Direct Data Intervention** (cache-2-cache transfer)
  - Copy **clean** data from one MP11 cache to another

- **Migratory Lines**
  - Move **dirty** data from one MP11 to another and skip MESI shared state.
  - Avoids writing to L2 and reading the data from external memory
Generic Interrupt Controller (GIC)

- **Distributor**
  - Detects and prioritizes interrupts
  - Used for:
    - configuring interrupt inputs
    - routing interrupts

- **CPU interfaces**
  - One per CPU
  - Used for:
    - acknowledging interrupts
    - masking interrupts
    - clearing interrupts
Inter-Processor communication

- Performed by means of **Inter Processor Interrupts** (IPIs)
  - Using reserved Interrupts 0 to 15
  - Software triggered by writing to a register

- Different broadcast modes
  - Sent Interrupt to a specific CPU
  - Send to all but self
  - Send to self

- Treated like normal (external) interrupts by Distributor

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBZ</td>
<td>CPU Target List</td>
<td>SBZ</td>
<td>Interrupt ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target list filter
The Development Environment

- **Hardware**
  - RealView Versatile Emulation Baseboard
  - CT11MPCore Core Tile
  - RealView ICE 3.1

- **Software**
  - RealView Development Suite 3.1
  - Linux 2.6.17
  - GCC 4.2.0
Implementation design

Single Multi-core Processor - ARM11 MPCore

Linux

CPU 0

control plane

SMP \mu\text{kernel}

CPU 1
CPU 2
CPU 3

/data device
driver

Linux CPU0 only

Shared buffer

SMP \mu\text{kernel} only

0 Mb

126 Mb

128 Mb

256 Mb

Physical Memory

NOT IN SCALE
The Control Plane (Linux)

- Linux kernel v2.6.17 from kernel.org
  - Applied MPCore support patches (now included with 2.6.21)

- Kernel built for 1 CPU but with SMP support
  - We need SCU and shared memory for communication buffer

- SMP Linux boots on CPU0
  - Unaware of any other CPUs
  - Only aware of 0-126Mb Physical RAM (out of 256Mb)
    - Using `mem=126M` in `bootargs`
The Data Plane (SMP ukernel)

- Light-weight bare-metal scheduler and threading library
  - NOT AN OS!

- Initialise CPUs partaking in SMP subsystem

- Implements subset of POSIX Threads
  - Providing well known programming abstraction

- Simple round-robin pre-emptive scheduling algorithm

- No optimization (created for demonstrative purposes only)
SMP System initialization (boot)

Primary CPU
- CPU1

Secondary CPUs
- CPU2
- CPU3

Common Reset handler

Core Init

Timer/Interrupts

Enable IPIs for scheduling

Operating-System initializes here

C/C++ Library

Thread Library

SMP scheduler ready

Create threads main()
Memory Layout

- RealView Versatile EB
  - 256 Mb RAM
- Linux loaded at 0x00008000
  - mem = 126M
- ukernel loaded at 0x08008000

typedef volatile struct
{
    int lock;
    int num_packets;
    char *base;
    unsigned int head;
    unsigned int tail;
    unsigned int size;
    unsigned int available;
    /* more */
} control_data_buffer_t;
The communication driver

- **Linux Kernel Module**
  - `/dev/ukernel abstracts the communication buffer`
  - Implements `open()`, `close()`, `write()`, `read()` and `ioctl()`s

- **Loading module**
  - Boots `ukernel` on data plane (via secondary boot register)
  - Maps in **shared** communication buffer (126-128Mb)
    - We benefit from DDI and cache line migration

- **Unloading module**
  - Exits `ukernel` on data plane
    - Sending to WFI – power saving
  - Resets and un-maps communication buffer
Loading the module (1)

# Clean up
if [ -e $COMMS_FILE ]; then
    rm $COMMS_FILE
    if [ "$?" -ne "0" ]; then
        echo -e "ERROR: Failed to remove $COMMS_FILE !\n"
        exit 1
    fi
fi

# Create /dev character driver file for the communication module
mknod -m a+rw $COMMS_FILE c $PRIMARY_NUM $SECONDARY_NUM
if [ "$?" -ne "0" ]; then
    echo -e "ERROR: Failed to change permissions for $COMMS_FILE !\n"
    exit 1
fi

# Install the module
insmod $MODULE_NAME
if [ "$?" -ne "0" ]; then
    echo -e "ERROR: Failed to install the $MODULE_NAME module !\n"
    exit 1
fi

exit 0

COMMS_FILE=/dev/ukernel
PRIMARY_NUM=42
SECONDARY_NUM=0
MODULE_NAME=ukernel.ko
One Page of a Document

Loading the module (2)

```c
static int __init ukernel_mod_init(void)
{
    writel((ukernel_base_pa + entrypoint_offset),
            (IO_ADDRESS(VERSATILE_HDR_BASE) + VERSATILE_HDR_FLAGSS_OFFSET));

    comms_base_va = __ioremap_pfn(comms_base_pa >> PAGE_SHIFT, 0, shared_size,
                                  L_PTE_CACHEABLE | L_PTE_BUFFERABLE | L_PTE_SHARED);

    memset_io(comms_base_va, 0, shared_size);

    [...] /* Initialise control data structures etc ... */

    ipi_broadcast_all_but_self(WAKE_WORK_PLANE_IPI);

    /* Register the device driver - only after we initialised everything */
    result = register_chrdev(MAJOR_NUMBER, DRIVER_NAME, &device_fops);

    [...] /* Handle failure to register */

    return 0;
}
```
# Uninstall the module
rmmod $MODULE_NAME
if [ "$?" -ne "0" ]; then
    echo -e "ERROR: Failed to uninstall the $MODULE_NAME module !\n"
    exit 1
fi

# Clean up
if [ -e $COMMS_FILE ]; then
    rm $COMMS_FILE
    if [ "$?" -ne "0" ]; then
        echo -e "ERROR: Failed to remove $COMMS_FILE !\n"
        exit 1
    fi
fi

echo "Success.\n"
exit 0
Unloading the module (2)

```c
static void __exit ukernel_mod_exit(void)
{
    /* Unmap the shared region */
    iounmap(comms_base_va);

    /* Unregister the device driver */
    result = unregister_chrdev(MAJOR_NUMBER, DRIVER_NAME);

    if (result < 0)
    {
        printk(KERN_ERR "Failed to unregister driver (result = %d).\n", result);
        return;
    }

    printk(KERN_INFO "Exiting module.\n");

    /* Reset the other CPUs - Send them to _wfi() */
    ipi_broadcast_all_but_self(QUIT_WORK_PLANE_IPI);
}
```
write() file operation

static ssize_t device_write(struct file *file, char *buf, size_t count, loff_t *offset)

- Called when a user program calls write()
- Writes to the communications buffer up to count bytes
- Returns number of bytes successfully written

```c
[...]
ret = write(dev_fd, buffer, PACKET_SIZE);
[...]
```

OR

```
cat data_file.dat > /dev/ukernel
```
read() file operation

static ssize_t device_read(struct file *file,
                          char *buf,
                          size_t count,
                          loff_t *offset)

- Called when a user program calls read()
- Retrieves from the communications buffer up to count bytes
- Returns number of bytes successfully read

[...] /* Try to read the packet. If buffer empty, keep trying */
while ( (ret = read(dev_fd, buffer, PACKET_SIZE)) == 0)
{
    printf(".");
    sleep(1);
}
[...]

OR

cat /dev/ukernel > data_file.dat
Control operations

static int device_ioctl(struct inode *inode, 
            struct file *file, 
            unsigned int cmd, 
            unsigned long arg)

- Used to send control command signals to the work plane

<table>
<thead>
<tr>
<th>CONTROL COMMAND</th>
<th>TYPE</th>
<th>DATA</th>
<th>SEQ NUM</th>
<th>IPIs</th>
</tr>
</thead>
<tbody>
<tr>
<td>UKERNEL_SET_NUM_THREADS</td>
<td>_IOW</td>
<td>int</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>UKERNEL_SET_CONSUMER_ID</td>
<td>_IOW</td>
<td>int</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>UKERNEL_GET_CONS_DESCR</td>
<td>_IORW</td>
<td>consumer_thread_t*</td>
<td>3</td>
<td>no</td>
</tr>
<tr>
<td>UKERNEL_LAUNCH_CONSUMERS</td>
<td>_IO</td>
<td>N/A</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>UKERNEL_COMMS_FINISHED</td>
<td>_IO</td>
<td>N/A</td>
<td>5</td>
<td>yes</td>
</tr>
</tbody>
</table>
Opening the communications device

```c
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/unistd.h>
#include <string.h>
#include <stdio.h>

#include "ukernel_mod.h"

#define DEVICE_NAME "/dev/ukernel"

int main(int argc, char **argv)
{
    int dev_fd;
    int ret;
    int num_threads, i;
    consumer_thread_t consumer_threads[MAX_CONSUMER_THREADS];

    dev_fd = open(DEVICE_NAME, O_RDWR);

    if (dev_fd == -1)
    {
        printf("\nNot able to open device file \"%s\".\n", DEVICE_NAME);
        return -1;
    }
```
List available consumers

    printf("\nThe list of available consumers is:\n");

    ret = ioctl(dev_fd, UKERNEL_GET_CONS_DESCR,
                 (consumer_thread_t *) &consumer_threads[0]);

    if (ret != 0)
        perror("ioctl");

    for (i = 0; i < MAX_CONSUMER_THREADS; i++)
        if (consumer_threads[i].function != NULL)
            printf("\n%d -> %s", i, consumer_threads[i].description);
Launch consumers

printf("\nLaunching %d threads of ID %d. \n", num_threads, consumer_id);

/* Setting the consumer ID */
ret = ioctl(dev_fd, UKERNEL_SET_CONSUMER_ID, &consumer_id);

if (ret != 0)
    perror("ioctl");

/* Setting the number of threads */
ret = ioctl(dev_fd, UKERNEL_SET_NUM_THREADS, &num_threads);

if (ret != 0)
    perror("ioctl");

/* Launch them */
    ret = ioctl(dev_fd, UKERNEL_LAUNCH_CONSUMERS);

if (ret != 0)
    perror("ioctl");
Interacting with the data plane

- Once consumer(s) have been launched
  - Data Plane will be polling for data
    - Power saving mode (WFI while no data available)
    - ioctl() signals no more data to be sent from control plane
      - Quits after buffer has been thereafter emptied

- Implementation defined what to do with this data
  - Depending on application requirements

- Passing data to data plane as simple as:

  $ data_file.dat > /dev/ukernel
Usage cases – Data engines

- Inside the work plane (the uKernel)
  - Register consumer threads

```c
void register_consumer_function(int id,
                                 void *(*function)(void *),
                                 char *description)
```

- Provide implementation of consumer threads
  - POSIX Thread’s thread-function format
  - Data consumption and returning implementation defined:
    - Blowfish Encrypt/Decrypt
    - Work out CRC of data packet
    - etc
    - Input stream packetized or not
    - Output stream order dependant
Usage cases – Video decoder

- Additional Linux patch
  - Secondary frame buffer
    - Linux writes to Primary buffer – original RGB out
    - Work plane writes to secondary buffer
    - Swapping between the two done with IOCTL command
- Installed XviD decoder consumer engine
  - Displays color-converted frames to secondary frame buffer
- $ cat video_stream.mp4 > /dev/ukernel
  - Data copied to internal temporary storage inside work plane by concurrent data-retriever thread

- CPU utilisation on CPU0 is none while video is processed!
Conclusions

ARM11 MPCore processor designed for:

- **Flexibility**
  - All CPUs configurable as AMP or partaking SMP subsystem
  - SCU designed to optimised migration of data within L1 boundaries
  - IPIs designed to allow for CPUs communications without memory

- **Scalability**
  - Adding processing power when required
  - Offload computation to separate cores/data-engine

- **Power saving**
  - Limiting power utilization when resource not necessary
    - Switch off work plane
    - WFI when data not available