Migrating Software to Multicore SMP Systems

Satyaki Mukherjee
Agenda

- Brief introduction to multicore on ARM
- Considerations on device drivers
- Considerations on user software
ARM Cortex-A MPCore

- ARMv7 architecture implementations
- Configurable number of cores
- Distributed Interrupt Controller
- MP aware D&T infrastructure
- MP optimized memory system
- Scalable & Flexible design
  - Supports SMP, AMP, virtualization
- Additional hardware support for:
  - Multimedia (VFP/NEON™)
  - Security (TrustZone®)
  - ACP (for MP implementations)
  - Virtualization (in Cortex-A15)
  - Large Addresses (in Cortex-A15)
## Announced MP SoC implementations

<table>
<thead>
<tr>
<th>Silicon Vendor</th>
<th>Platform name</th>
<th>Marketed performance</th>
<th>App Processor</th>
<th>Number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nvidia</td>
<td>Tegra 250</td>
<td>up-to 1 Ghz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Samsung</td>
<td>Orion</td>
<td>1 Ghz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>OMAP 4430</td>
<td>up-to 1 Ghz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>OMAP 4440</td>
<td>over 1 Ghz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>ST-Ericsson</td>
<td>U8500</td>
<td>up-to 1.2 Ghz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>ST-Ericsson</td>
<td>U5500</td>
<td>Optimized for power</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Renesas (NEC)</td>
<td>EMMA Mobile/EV2</td>
<td>533 MHz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Renesas (NEC)</td>
<td>EMMA Car EC-4270 (NaviEngine)</td>
<td>400 MHz</td>
<td>ARM11 MPCore</td>
<td>4</td>
</tr>
<tr>
<td>Renesas (NEC)</td>
<td>EMMA Car EC-4250 (NaviEngine-mini)</td>
<td>400 MHz</td>
<td>ARM11 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>Renesas (NEC)</td>
<td>EMMA Car EC-4260 (NaviEngine-MID)</td>
<td>400 MHz</td>
<td>ARM11 MPCore</td>
<td>3</td>
</tr>
<tr>
<td>Renesas (NEC)</td>
<td>EMMA Car EC-43xx Series</td>
<td>TBC</td>
<td>A9 MPCore</td>
<td>4</td>
</tr>
<tr>
<td>STMicroelectronics</td>
<td>SPEAr1310 (Comms MPU)</td>
<td>600 MHz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>NuFront</td>
<td>NuSmart 2816</td>
<td>up-to 2 GHz</td>
<td>A9 MPCore</td>
<td>2</td>
</tr>
<tr>
<td>MindSpeed</td>
<td>Transcede 4000 (4G wireless baseband stations)</td>
<td>600 MHz</td>
<td>A9 MPCore</td>
<td>4+2</td>
</tr>
<tr>
<td>MindSpeed</td>
<td>Transcede 4020</td>
<td>750 MHz</td>
<td>A9 MPCore</td>
<td>4+2</td>
</tr>
<tr>
<td>MindSpeed</td>
<td>M8500 (Comcerto® 5000 Media Processing SoC)</td>
<td>600 MHz</td>
<td>A9 MPCore</td>
<td>4+2</td>
</tr>
<tr>
<td>Marvell</td>
<td>88F6323/2/1 (Sheeva)</td>
<td>0.8 to 1 Ghz</td>
<td>ARMv5TE</td>
<td>2</td>
</tr>
<tr>
<td>Marvell</td>
<td>ARMADA 1000 (88DE3010) (Sheeva)</td>
<td>1.2 GHz</td>
<td>ARMv5TE</td>
<td>2</td>
</tr>
<tr>
<td>Marvell</td>
<td>ARMADA Quadcore (Announced at CES 2010)</td>
<td>-</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>Marvell</td>
<td>ARMADA 628</td>
<td>2x 1.5GHz + 1x 624 MHz</td>
<td>ARMv7 MP</td>
<td>3</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>MSM8260 Snapdragon</td>
<td>up-to 1.2 Ghz</td>
<td>ARMv7</td>
<td>2</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>MSM8660 Snapdragon</td>
<td>up-to 1.2 Ghz</td>
<td>ARMv7</td>
<td>2</td>
</tr>
<tr>
<td>Qualcomm</td>
<td>QSD8672 Snapdragon</td>
<td>1.5 GHz</td>
<td>ARMv7</td>
<td>2</td>
</tr>
</tbody>
</table>

Publically announced SoCs based on multicore ARM processors. Does not include future roadmap. Correct at time of publishing.
<table>
<thead>
<tr>
<th>Toshiba AC/100</th>
<th>MSI WindPad 110</th>
<th>Notion Ink Adam Tablet</th>
<th>NVIDIA/Foxconn</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-Noa Interpad</td>
<td>Toshiba Folio 100</td>
<td>Aigo N700</td>
<td>RIM PlayBook</td>
</tr>
<tr>
<td>Audi MMI System</td>
<td>eLocity A7</td>
<td>Malata SMB-A1011</td>
<td>Mobinova Beam</td>
</tr>
</tbody>
</table>
But... Why SMP?

Better energy efficiency

Increased performance
The SMP OS Takes Care of it All!

Optimal multi-tasking
- User Interface
- Multimedia
- Communication
- Web
- Other tasks

Automatic load balancing to enable dynamic power management

Cortex-A? MPCore
- CPU 1
- CPU 2
- CPU 3
- CPU 4

No code re-write necessary

All complexities hidden by the OS

Automatic distribution of existing applications across available cores
SMP Power Management Options

Full Operation

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DVFS

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Individual-Core Power-Down

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ICPD + DVFS

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>66%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OFF

<table>
<thead>
<tr>
<th></th>
<th>CPU0</th>
<th>CPU1</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OFF</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Architecture for the Digital World®
Example: Android SMP

- Every application runs in its own Linux process
- Java threading abstracts the multicore SMP architecture
- Low-level PThreads API still accessible for advanced developers
- Each instance of the Dalvik virtual machine runs in its own process
- By default all components of an application run in the same process, but it is possible to change this and spawn threads/processes
- Middleware can be independently optimized for SMP
- Linux is a mature and optimized SMP kernel

![Android SMP Diagram]

Applications

Application Framework

Libraries

Runtime

SMP Linux Kernel

SMP Processor

Simplified Android stack
Migrating Software to SMP

**VALIDATION**
Ensure that software works on a SMP system

**OPTIMIZATION**
- Remove serialization bottlenecks
- Modify software to expose concurrency
- Parallelise if the algorithm is suitable and the effort is worthwhile
Migrating software to multi-core

DEVICE DRIVERS
Synchronization is required when multiple agents interact with shared resources (or critical regions)

Resource can be a location in memory, a shared data structure, a physical device, a non-thread-safe piece of code...
Critical Sections and Mutual Exclusion

Thread A

...  
mutex_lock(&lock);  
/* critical section */  
mutex_unlock(&lock);  
...  

Thread B

...  
mutex_lock(&lock);  
/* critical section */  
mutex_unlock(&lock);  
...
Implications of Enabling Multitasking

- To enable multitasking various scheduling models are available:
  - Run To Completion
  - Cooperative: the running task yields to relinquish CPU
  - Pre-emptive: the OS is in charge of scheduling

- On a UP system
  - Only one task is **executing** at any time
    - Cooperation may assume this
  - The kernel is protected by:
    - Making kernel processes non-pre-emptable
    - Disabling interrupts inside critical sections

- On an SMP system
  - More than one task can be **executing** at the same time
    - UP kernel protection mechanism not adequate
    - Can no longer assume that lower priority tasks are in the wait queue
Spinlocks (kernel)

- Busy-wait mechanism to enforce mutual exclusion on multicore systems
  - It is a binary mechanism (lock/unlock). It is very quick.
  - Not suitable when holding lock for long time – spinlocks do not sleep
  - Energy efficient implementation for ARM architecture
  - If the protected resource can be accessed from an ISR, use the \texttt{*_irq*} variants
  - Not recursive: acquiring the same lock twice causes dead-lock

```c
#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED; /* initialize */
...
spin_lock(&lock); /* acquire the lock */
/* critical section */
spin_unlock(&lock); /* release the lock */
...```
Semaphores (kernel)

- Use semaphores for synchronization between processes
  - There’s also APIs for killable/interruptable/timeout/try
  - Semaphores can sleep: Cannot be used in interrupt handlers
  - They are relatively slow
  - Not recursive: acquiring the same lock twice causes dead-lock
  - In reality should use `down_interruptible()`, since `down()` will not return if a signal is received

```c
#include <linux/semaphore.h>
struct semaphore sem;
sema_init(&sem, 1); /* initialize the semaphore */
...
down(&sem); /* acquire the semaphore */
/* critical section */
up(&sem); /* release the semaphore */
...```
Spinlocks in Interrupt Handlers

- Potential for deadlock if a locked spinlock is acquired inside an interrupt handler

```c
... 
spin_lock(&lock);  
/* critical section */ 
... 
... 
... 
spin_unlock(&lock);  
...

/* Interrupt handler */ 
spin_lock(&lock);  
/* DEADLOCK */ 
... 
INTERRUPT
```
Spinlocks in Interrupt Handlers

- Use the appropriate API variant
  - They disable interrupts on the calling CPU whilst the lock is held
  - Use to synchronize interrupt handler and non-interrupt code

- Entering critical region:
  - `spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`
  - `spin_lock_irq(spinlock_t *lock)`

- Exiting critical region:
  - `spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`;
  - `spin_unlock_irq(spinlock_t *lock)`
Multiple-reader Locking

- **Read/write semaphores**
  - Same semantics as conventional semaphores, allow multiple readers

- **Read/write spinlocks**
  - Same semantics as conventional spinlocks, allow multiple readers

- **Read-Copy-Update (RCU)**
  - Specialist synchronization mechanism for when readers >> writers
  - Example: Shared access of networking routing tables: every outgoing packet (process) needs to read the routing table. Once the read is completed the entry can be removed. The removal can be deferred. The reading can be done without locking

- **Seqlocks**
  - Fast, lockless, suitable when the critical region is very small
  - Example: A global variable is regularly updated with system time or event counters, and many threads regularly sample this information
Thread Safety and Re-entrancy

- Functions that can be used concurrently by several threads need to be **thread-safe** and **re-entrant**

- Important for device driver calls and application/library functions

**Re-entrant** function
- All data provided by the caller:
  - Function does not hold static (global) data over successive calls
  - Function does not return a pointer to static data
  - Does not call non-re-entrant functions

**Thread safe** function
- Protects shared data with locks

- `strtok()` and `ctime()` are not re-entrant
  - `strtok()` holds the string to be broken into tokens
  - `ctime()` returns a pointer to static data that is overwritten by each call
Explicit Task Ordering Using Priorities

- Code written for uni-processor system, using real-time scheduling policies may assume ordered execution of tasks based on their priority
  - Higher priority task runs to completion, or until it explicitly yields
  - Tasks’ priorities are used to guarantee execution order
- The issue: On a multicore SMP system a task of lower priority may in fact run concurrently, on another CPU
  - The expected execution order is no longer guaranteed
**Solution 1: Use Task Affinity**

- Setting task affinity to a specific CPU and specifying SCHED_FIFO for that process will ensure that implicit execution order of legacy code on SMP systems is maintained.

```c
#include <sched.h>

int sched_setaffinity(pid_t pid,
                       unsigned int cpusetsize,
                       cpu_set_t* mask);
```

- **Advantages:**
  - Small code change/redesign overhead
  - Can set up the scheduler to run all other tasks on other CPUs
  - Better and more predictable performance than on UP: dedicated CPU for real-time activity, and more bandwidth for everything else

- **Disadvantages:**
  - Reliance on a scheduler’s mechanism
  - Breaks the SMP paradigm and comes in the way of OS load balancing
Solution 2: Use Explicit Synchronization

- Enforce serial execution using synchronization mechanisms
  - Semaphores, spinlocks, signals, completions, etc
  - The task that has to wait blocks on a lock
  - The task that has to signal releases the lock

- Advantages
  - The programmer explicitly controls the execution flow
  - Performance is likely to be similar or better on multi-core vs single-core
  - ... actually, other tasks can run on other cores, therefore multi-core wins!

- Disadvantages
  - Need to manually serialize the code, and fine tuning may be required
  - Latency associated with specific locking mechanisms

- The kernel provides different synchronization primitives for different needs, and the best fit is case-specific.
**Linux Kernel Memory Barriers**

- Barriers are needed to enforce ordering of memory operations
  - Compiler aggressive optimization
  - Processor micro-architecture optimizations
  - Multicore CPUs
  - Relaxed memory ordering systems

- Linux provides a selection of many barriers to choose from

- Barriers are platform and architecture dependant
  - Not recommended to reverse engineer when to use certain barriers by looking at what their implementation is for a specific architecture version – this can be misleading…
  - Build Linux with the right parameters and you are ok

- Locking and scheduling functions imply barriers
## Barriers in Linux

| Compiler barrier | `barrier()` | Generic explicit compiler barrier  
|                 |             | Compilers not to reschedule memory accesses to/from either side  
| I/O barrier     | `mmiowb()` | To be used with memory mapped I/O writes  
|                 |             | I/O memory is un-cached, but ordering behavior cannot be guaranteed  
|                 |             | Ensure ordering between CPU and I/O device  
| CPU memory barrier | `mb()` | Used for ordering on Normal Cacheable memory  
|                 |             | Can be used to control MMIO effects on accesses through relaxed memory I/O windows  
|                 |             | They also imply a compiler barrier  
|                 | `wmb()` | Examples of typical memory barriers uses:  
|                 |             | - after writing DMA buffers and before starting DMA transfers  
|                 | `rmb()` | - when a device driver writes some state to memory and expects it to be visible in an interrupt routine which may arrive on a different CPU  
|                 | `read_barrier_depends()` | Relevant to both UP and SMP systems  
| SMP barriers    | `smp_mb()` | Used to control the ordering of references to shared memory between CPUs within a cache-coherent SMP cluster  
|                 |             | A compiler barrier is also implied  
|                 | `smp_wmb()` | On UP systems (no CONFIGURE_SMP build option), all `smp_*()` barriers are reduced to compiler barriers  
|                 | `smp_rmb()` | `smp_*` barriers are weaker than `?mb()`  
|                 | `smp_read_barrier_depends()` |  

---
Software Configurable Interrupts

- On ARM multicore systems, interrupts are assigned to CPU0 by default.
- They can be re-assigned dynamically using Linux kernel APIs:

```c
int irq_set_affinity(unsigned int irq, const struct cpumask *m);
```
Interrupt Load Balancing

- Linux used to support in-kernel interrupt load balancing for some architectures
  - This has been removed from 2.6.29 ([http://bit.ly/bYj4Op](http://bit.ly/bYj4Op))

- `irqbalance` is a user-space daemon to dynamically load-balance interrupts
  - Designed to take into account both power savings and optimal performance

- Load-balancing of interrupts across cores is not always the best solution
  - Main considerations relate to cache migrations and contention, and their effect on performance (note that the ARM MPCore™ architecture provides hardware optimizations and instructions to mitigate these)
  - Example1: A device has multiple interrupts associated to it, and a unique control data structure
  - Example2: Migrations overheads amplified by DFS
  - Example3: It may be desirable to keep a CPU in idle state for longer so that the kernel will decide to put it in a deeper sleep state
Migrating software to multi-core

APPLICATION & MIDDLEWARE
Application software migration to multi-core

- All modern Operating Systems support multicore, various models:
  - Symmetric multi-processing
  - Asymmetric multi-processing
  - Bound computational domains
  - Message passing architectures and distributed systems

- The vast majority of heritage code base needs NO modification
  - OS automatically distributes tasks across abstracted processing layer
  - Distributed communication protocols (e.g. MPI) offset overhead
  - Thread safety and re-enterancy safety required for shared code
  - Performance fine tuning and partial re-design may be required

- ARM is actively working with HLOS/RTOS vendors
  - Power efficient SMP implementations, incl. Advanced Power Management
  - Various levels of progress – contact vendors for details
Fundamentals of Explicit Parallelism

- Code modified to split workloads across available CPUs
- Frequency and power requirements lowered by utilization of multiple cores
- *Dominant thread* split across available resources
- Single application enabled to use more than one CPU

### ANALYZE
Analyze the UP application/problem to find parallelisable areas

### DECOMPOSE
Decompose problems into independent tasks for parallel execution:
- Task Decomposition
- Functional Block Partitioning
- Data Decomposition

### PARALLELIZE
Parallelize using threading industry standard APIs, libraries and models such as POSIX Threads and OpenMP
Data Decomposition

Each data item is independent

Split large quantity of DATA into smaller chunks that can be operated in parallel.
Data Decomposition

- Split large quantity of **data** into smaller chunks to consume in parallel
  - **No change in algorithm’s source code**
  - Concurrent execution of tasks on independent data allows execution to terminate earlier and therefore save time and energy
  - Appropriate level of granularity needs to be determined
  - Shared memory system is not to be abused
Data Decomposition Granularity

- Subdividing a data processing operation into several threads executing in parallel on smaller chunks of data
  - Pixel by pixel – 1 pixel per thread
  - Line by line – 1 line per thread
  - Section by section – 1/n of picture per thread (n = number of PUs)
  - Individual images or group of images in a stream
Functionally independent tasks can be executed concurrently.
Functional Block Partitioning

Functional blocks are serially dependent but temporally independent

Distribute different functional blocks across available processors
Split into defined functional threads
Uses passing of data blocks between threads to allocate work

Requires code changes and fine tuning

Example: Real Time Video Codec

(Simplified MPEG encoding functional block diagram)
Task Allocation Models

FORK-EXEC MODEL

Master thread

Parallel region

Create a thread on demand

Master thread

WORKER-POOL MODEL

Thread pool

Task queue

Completed tasks

Hand off work to pool of worker threads

Create a thread on demand

Hand off work to pool of worker threads
Typical Pitfalls of SMP Software

- Synchronization overheads
  - Serialized regions caused my critical sections or rendezvous points

- Dead-locks
  - 2+ threads are each other waiting for another to release a resource
  - Such threads are blocked on a lock that will never be released

- Live-lock
  - Multiple threads continue to run (without blocking indefinitely like in the case of a dead-lock) but the system as a whole is unable to make progress due to repeating patterns of non-productive contention

- Priority Inversion
  - When a lower priority task acquires a lock on a resource and inhibits a higher priority task from progressing
  - Addressed by priority inheritance and priority ceiling techniques

- Cache trashing and false sharing
Summary

- Multicore and SMP brings many outstanding benefits
- Several common knowledge pitfalls when migrating software
- Understanding the challenges and implications is key
Thank You

Please visit www.arm.com for ARM related technical details

For any queries contact < Salesinfo-IN@arm.com >