Linux®: Asymmetric Multiprocessing (ASMP) Versus Symmetric Multiprocessing (SMP)

V1.0

Ted Peters
Sr. Applications Engineer
Agenda

► Intro to the QorIQ™ P4080 processor

► Multicore operating system architecture: SMP versus ASMP

► SMP and ASMP use cases

► Configuration of resources: SMP versus ASMP

► P4080 SMP u-boot/Linux® boot process

► P4080 ASMP u-boot/Linux® boot process

► Questions and answers
Intro to the QorIQ™ P4080 Processor
QorIQ™ P4 Series Block Diagram

128KB Backside L2 Cache

Power Architecture™ e500-mc Core

32KB D-Cache
32KB I-Cache

1024KB Frontside L3 Cache

CoreNet™ Coherency Fabric

Power Architecture™ e500-mc Core

Frame Manager

Parse, Classify, Distribute

Buffer

10GE
1GE
1GE
1GE

18-Lane 5 GHz SERDES

64-bit DDR-2 / 3 Memory Controller

64-bit DDR-2 / 3 Memory Controller

CoreNet™ Trace

Real Time Debug

Watchpoint Cross Trigger

Perf Monitor

PCIe

SRIO

Aurora

PCIe

PCIe

PCIe

PCIe

2x DMA

1024KB Frontside L3 Cache

1024KB Frontside L3 Cache

RapidIO Message Unit (RMU)

1GE
1GE
1GE
1GE
Multicore Operating System Architecture: SMP Versus ASMP
Symmetric Multiprocessing (SMP)
Symmetrical Multiprocessing

- An architecture that provides fast performance by making multiple CPUs available to complete individual processes simultaneously (multiprocessing)

- Any idle CPU can be assigned any task, and additional CPUs can be added to improve performance and handle increased loads

- SMP uses a single operating system and shares common memory, all the IO and interrupt resources

- Processes and threads are distributed among CPUs
Asymmetrical Multiprocessing

- Each CPU group runs its own OS which may be same or different from each other
  - Each CPU group can be assigned with a specific application to run
  - If Linux®, multiple copies of uImage are needed, but located at different physical address spaces

- All CPU groups must cooperate to share the resources
  - No single OS can own the whole system
  - I/O and interrupts are divided up amongst the CPU groups
  - Static configuration for resources
  - If Linux, resources allocation can be done by device trees (dts)

Multiple cores → multiple OS’s
A common multicore usage model is to run multiple operating systems.

Requires partitioning hardware resources:
- Private resources: CPUs, memory, I/O devices
- Shared resources: memory, devices

Doing this cooperatively (all operating systems well-behaved) presents challenges.
Partitioning with a Hypervisor

Hypervisor Software

- Analogous to role of an operating system kernel in managing user processes
- More privileged than operating systems
- Enforces system security
- Manages globally shared resources
- Virtualizes some resources—e.g. interrupt controller, UART

Diagram:
- Partitioning with Hypervisor Software
- Hypervisor manages partitions
- Multiple operating systems (Linux®, RTOS, Legacy OS)
- Multicore System Hardware
- Shared Cache, Interrupt Controller, Memory, I/O
SMP and ASMP Use Cases
Use Cases

► Consolidation
  • Multiple operating systems/partitions on a single multicore chip
  • Multiple homogeneous operating systems in an AMP configuration on multiple cores

► Divided workload (e.g. control plane, data plane)
  • Multiple operating systems, possibly heterogeneous, need to work securely and seamlessly together
  • Isolation mechanisms are needed for safety, robustness
  • Efficient inter-partition communication mechanisms are needed for cooperation

► In-service upgrade

► Isolate untrusted software/sandboxes
  • Isolate untrusted operating systems: Proprietary OS + open OS (e.g. Linux®)
  • Isolate end-user installed software
  • Software under test
  • Isolated partition for GPL-based software

► Migration
  • Migrate functionality from legacy RTOS to another OS (e.g. Linux).

► Security
  • Secure partition for sensitive security tasks (e.g. access rights control, rule definitions, key storage/management)
Configuration of Resources: SMP Versus ASMP
Configuration of Resources: SMP

► In an SMP environment:
  • All cores execute from one Linux® image
  • Care must be taken to not reinitialize shared resources that have already been configured by u-boot
    ▪ LAWs, DDR, L2, etc.

► QorIQ™ P4080 SMP u-boot and Linux
  • U-boot on core0 is responsible for:
    ▪ Initializing the memory map, DDR and all other cores
  • Linux on core0 is responsible for initializing all shared resources not already configured by u-boot and releasing other cores from their spin loop
    ▪ MPIC, eTSEC, PCI/PEX drivers
  • When a secondary core boots the same Linux image, it will enter Linux at a specific location initializing resources specific to that secondary core and then enters the idle loop
Configuration of Resources: AMP

In an AMP environment:

- All partitions execute their own OS image
- Care must be taken to not reinitialize shared resources that have already been configured by u-boot
  - LAWs, DDR, L2, etc.

Traditional AMP u-boot and Linux®

- U-boot on core0 is responsible for:
  - Initializing the memory map, DDR and all other cores
  - Load Linux image and device tree for each of the other cores and release them
- Linux on each core is responsible for initializing all assigned resources specified in that core’s dts
  - MPIC interrupts, eTSEC, PCI/PEX drivers
Configuration of Resources: Partitioned With Hypervisor

U-boot on core0 is responsible for:
- Initializing the memory map, DDR and boot all other cores to spin table
- Load hypervisor image and hardware device tree

Hypervisor on core0 is responsible for:
- Performs its own internal initialization of hypervisor controlled resources
  - MPIC, UART, MMU, etc.
- Releases all secondary CPUs from the spin table
- Initializes each partition, and boots guest operating system with dynamically created guest device tree on each partition

Linux® on each core is responsible for initializing all assigned resources specified in that core’s dts
- eTSEC, PCI/PEX drivers, DMA, etc.
QorIQ™ P4080 SMP u-boot/Linux®
Boot Process
Multicore Boot Architecture

- ePAPR describes specifics on how secondary CPUs are booted for a system with multiple CPUs

- Default boot architecture
  - The boot program releases all CPUs from hardware reset
  - One CPU is designated to be the client program’s boot CPU
  - All other CPUs are secondary and are placed into loop where the CPUs spin, waiting for a spin table field to change that directs them where to go
  - Control is transferred to the client program on the boot CPU
  - When the client program is ready for secondary cores to start, it releases them by writing the spin table field with the desired address

- The architecture allows for other custom-defined secondary CPU release mechanisms as well

Reference: ePAPR (Power.org *Standard for Embedded Power Architecture™ Platform Requirements (ePAPR)*. power.org, 2008.)
SMP Boot Process

- Power-on Reset
  - Core0 comes out of reset at the reset vector 0xFFFF_FFFC and all other cores are in boot hold off mode
    - Core0 runs u-boot
      - After setting BPTR to secondary core’s start page, core0 kicks off each additional secondary core
      - Each secondary core comes out of reset at __secondary_start_page(boot page), initialize resources specific that that core (caches, MMU, etc.) and enters a spin loop
    - Core0 loads Linux® image and device tree and boots Linux
SMP Boot Process: U-boot Boot Process

► In the u-boot source, the entry point is the reset vector 0x0_FFFF_FFFC containing a simple branch to `start_e500`, which is at the base of default boot page at 0x0_FFFF_F000; both are found in `cpu/mpc85xx/start.S`

```
Reset vector → call _start_e500()
```

► `_start_e500()` will:

```
_start_e500() → Boot other cores → Start command interpreter → Prepare Linux® device tree
```

1. Enable L1 caches
2. Configure interrupt vectors
3. Configure MMU and LAWs
4. Configure L1D RAM
5. Configure local bus
6. Configure DDR
7. Relocate to DDR

1. Setup BSTRL and BSTAR
2. Release all other cores from boot holdoff using BRR and wait for them to boot.

1. Copy device tree to DDR
2. Copy uImage to DDR
3. Copy Ramdisk to DDR (optionally)
4. Launch kernel
QorIQ™ P4080 ASMP u-boot/Linux®
Boot Process
ASMP Boot Process

Power-on Reset

- Core0 comes out of reset at the reset vector 0xFFFF_FFFC and all other cores are in boot hold off mode
  - Core0 runs u-boot
    - After setting BPTR to secondary core’s start page, core0 kicks off each additional secondary core
    - Each secondary core comes out of reset at __secondary_start_page(boot page), initialize resources specific that that core (caches, MMU, etc.) and enters a spin loop
  - Core0 loads Linux® image and device tree for each of the other cores individually and releases the core to boot Linux
  - Core0 then loads its own Linux image and device tree and boots Linux
In the u-boot source, the entry point is the reset vector 0x0_FFFF_FFFC containing a simple branch to _start_e500, which is at the base of default boot page at 0x0_FFFF_F000; both are found in cpu/mpc85xx/start.S

=start_e500() will:

1. Enable L1 caches
2. Config interrupt vectors
3. Config MMU and LAWs
4. Config L1D RAM
5. Config local bus
6. Config DDR
7. Relocate to DDR

Reset vector → call _start_e500()

Start command interpreter

Prepare Linux® device tree

- bootm start $loadaddr $ramdiskaddr $fdtaddr
- bootm loados
- bootm ramdisk
- bootm fdt
- Boardsetup
- fdt chosen $initrd_start $initrd_end
- bootm prep
- cpu 1 release $bootm_low - $fdtaddr -
Questions and Answers
Backup
This session will cover the concepts of asymmetric multiprocessing (ASMP) and symmetrical multiprocessing (SMP) in the Freescale QorIQ™ multicore Linux® implementations. Key differences between ASMP and SMP will be highlighted followed by a detailed view of resource sharing in the hardware. ASMP and SMP require different initialization sequences which are explained with code specifics. Learn what systems and scenarios lend themselves to ASMP or SMP.
Basic System Configuration

- The device tree allocates resources to individual partitions
- Partition nodes within the device tree define how individual partitions use resources
Terminology

- SMP – Symmetric Multiprocessing
- ASMP – Asymmetric Multiprocessing (aka. AMP)
- CAMP – Cooperative AMP
- Guest OS – OS running in guest supervisor state under a hypervisor
- Hypervisor - Manages globally shared resources, more privileged than operating systems, enforces system security, virtualizes some resources
- Light Weight Executive (LWE) - Provides a communications mechanism between an application running in a partition and the hypervisor software
- Device Tree - A data structure used for representing a partition’s physical and virtual devices
In the Linux® kernel, the entry point is `_start` found in `arch/powerpc/kernel/head_fsl_booke.S`

```
_start  → call _start_kernel()
```

`start _kernel()` is in `init/main.c`, it will:

```
... mpc85xx_smp_init() {
  smp_ops = &smp_85xx_ops;
} ......
```

```
_kernel()  → setup_arch()  → smp_prepare_cpu()  → cpu_up()  → smp_cpus_done()  → smp_ops->setup_cpu()
```

```
_kernel()  → setup_arch()  → smp_init()  → smp_ops->probe()
```

```
_kernel()  → setup_arch()  → smp_ops->kick_cpu()
```
Let’s look at `smp_mpc85xx_kick_cpu()` found in:

```
arch/powerpc/platforms/85xx/mpc85xx_smp.c
```

```c
smp_mpc85xx_kick_cpu() {
    ..... 
    /* Get the BPTR */
    bptr_vaddr = ioremap(get_immrbase() + MPC85xx_BPTR_OFFSET, 4);
    /* Set the BPTR to the secondary boot page */
    oldbptr = in_be32(bptr_vaddr);
    bptr = (BPTR_EN | (__pa((unsigned)__secondary_start_page) >> 12));
    out_be32(bptr_vaddr, bptr);
    /* Kick that CPU */
    smp_85xx_release_core(nr);
    /* Wait a bit for the CPU to take the exception. */
    while ((__secondary_hold_acknowledge != nr) && (++n < 1000))
        mdelay(1);
    /* Restore the BPTR */
    out_be32(bptr_vaddr, oldbptr);
    ..... 
}
```
SMP Boot Process: Core1 “Kick-Off” Process

- Core 0 kicks off core 1 by setting EEBPCR[CPU1_EN]
- Let's look at smp_85xx_release_core() in
  `arch/powerpc/mpc85xx/mpc85xx_smp.c`

```c
smp_85xx_release_core(int nr)
{
    ....
    /*
     * Startup Core #nr.
     */
    ecm_vaddr = ioremap(get_immrbase() + MPC85xx_ECM_OFFSET,
                        MPC85xx_ECM_SIZE);
    pcr = in_be32(ecm_vaddr + (ECM_PORT_CONFIG_OFFSET >> 2));
    pcr |= EEBPCR_CPU1_EN;
    out_be32(ecm_vaddr + (ECM_PORT_CONFIG_OFFSET >> 2), pcr);
}
```

<table>
<thead>
<tr>
<th>R</th>
<th>W</th>
<th>CPU1_EN</th>
<th>CPU0_EN</th>
<th>CPU1_PRI</th>
<th>CPU_RD_HI_DIS</th>
<th>CPU0_PRI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

EEBPCR
SMP Boot Process: Core1 Kernel Code Flow

Core1 starts up from __secondary_start_page defined in arch/powerpc/kernel/head_fsl_booke.S

- __secondary_start_page:
  - 4K
  - The code in this page must not exceed 1023 instruction. The 1024th is a branch instruction.
  - It will initialize I-cache and D-cache
  - Use TLB1[1] to map 16M memory
  - Jump to __early_start

- __early_start
  - If it is core1, it will not call start_kernel(), but jump to __secondary_start

- __secondary_start
  - Jumps to startSecondary()

start_secondary() defined in arch/powerpc/kernel/smp.c

- Call smp_ops->setup_cpu()
- ......
- Call cpu_idle()