Porting Linux to a new processor architecture

Embedded Linux Conference 2016

Joël Porquet

April 4th, 2016
Context

SoCLib
- FR-funded project (2007-2010)
- 10 academic labs and 6 industrial companies
- Library of SystemC simulation models

TSAR/SHARP
- Two consecutive EU-funded projects (2008-2010/2012-2015)
- Massively parallel architecture
- Shared and hardware-maintained coherent memory

Post-doc position at Sorbonne University, May 2013 - Feb 2015
Outline of the presentation

1. Mono-processor system

   - MIPS32
   - L1 cache + MMU
   - INT controller
   - Block device
   - Local Crossbar
   - Timer controller
   - UART
   - RAM

2. Multi-processor system

   - MIPS32
   - L1 cache + MMU
   - Local Crossbar
   - XICU
   - UART
   - RAM
   - Block device
   - DMA
   - Memory Cache (L2)
   - Frame Buffer
   - Boot ROM
   - UART
   - Frame Buffer
   - I/O network
   - I/O PIC
   - DMA
   - Block Device
   - 2D-mesh Network-on-Chip

3. Multi-node (NUMA) system

   - MIPS32
   - L1 cache + MMU
   - Local Crossbar
   - XICU (IRQ+timer+mailbox)
   - DMA controller
   - Memory Cache (L2)
   - L3 + Ext RAM

Joël Porquet
Embedded Linux Conference 2016
April 4th, 2016
Is a new port necessary?

Types of porting

- New *board* with already supported processor
- New *processor* within an *existing, already supported* processor family
- New processor architecture

Hints

- TSAR has processor cores compatible with the MIPS32 ISA
- But the virtual memory model is *radically* different

Answer

$ mkdir arch/tsar

*Porting Linux to a New Architecture, Marta Rybczyńska, ELC'2014 - https://lwn.net/Articles/597351/
How to start?

Two-step process

1. Minimal set of files that define a minimal set of symbols
2. Gradual implementation of the boot functions

Typical layout

$ ls -l arch/tsar/
   configs/
   drivers/
   include/
   kernel/
   lib/
   mm/

$ make ARCH=tsar
arch/tsar/Makefile: No such file or directory
How to start?

Two-step process

1. Minimal set of files that define a minimal set of symbols
2. Gradual implementation of the boot functions

Adding some build system

$ ls -l arch/tsar/
  configs/
       tsar_defconfig*
  include/
  kernel/
  lib/
  mm/
  Kconfig*
  Makefile*
How to start?

Two-step process

1. Minimal set of files that define a minimal set of symbols
2. Gradual implementation of the boot functions

Arch-specific headers

```bash
$ ls -l arch/tsar/
  configs/
    tsar_defconfig
  include/
    asm/*
    uapi/asm/*
  kernel/
  lib/
  mm/
  Kconfig
  Makefile
```
The boot sequence

(arch/tsar/kernel/head.S)              kernel_entry*
(init/main.c)                        start_kernel
(arch/tsar/kernel/setup.c)            setup_arch*
(arch/tsar/kernel/trap.c)             trap_init*
(init/main.c)                        mm_init
                                             mem_init*
(arch/tsar/mm/init.c)                 init_IRQ*
(arch/tsar/kernel/irq.c)              time_init*
(arch/tsar/kernel/time.c)             rest_init
                                             kernel_thread(kernel_init)
(init/main.c)                        kernel_thread(kthread)
(kernel/kthread.c)                    cpu_startup_entry
(kernel/cpu/idle.c)
Early assembly boot code

**kernel_entry()**

- resets the processor to a default state
- clears the `.bss` segment
- saves the bootloader argument(s) (e.g. device tree)
- initializes the first page table
  - maps the kernel image
- enables the virtual memory and jumps into the virtual address space
- sets up the stack register (and optionally the current thread info register)
- jumps to `start_kernel()`

```
/* enable MMU */
li t0, mmu_mode_init
mtc2 t0, $1
nop
nop

/* jump into VA space */
la t0, va_jump
jr t0
va_jump:
...```

![Virtual Memory Diagram](image-url)
**setup_arch()**

- Scans the flattened device tree, discovers the physical memory banks and registers them into the memblock layer
- Parses the early arguments (e.g. `early_printk`)
- Configures memblock and maps the physical memory
- Memory zones (ZONE_DMA, ZONE_NORMAL, ...)

---

**Diagram:**

- **Virtual Memory**
  - *user space*
  - *kernel space*

- **Physical Memory**

- Physical addressing
  - `0GiB` to `3GiB`
  - `3GiB` to `4GiB`

- Direct Mapping

- `vmalloc`
The exception vector acts as a dispatcher:

\begin{verbatim}
mfc0 k1, C0_CAUSE
andi k1, k1, 0x7c
lw k0, exception_handlers(k1)
jr k0
\end{verbatim}

- Configures the processor to use this exception vector
- Initializes exception_handlers[] with the sub-handlers (handle_int(), handle_reserved(), etc.)
Sub-handlers:

ENTRY(handle_int)
    SAVE_ALL
    CLI
    move a0, sp
    la ra, ret_from_intr
    j    do_IRQ
ENDPROC(handle_int)

ENTRY(handle_bp)
    SAVE_ALL
    STI
    move a0, sp
    la ra, ret_from_exception
    j    do_bp
ENDPROC(handle_bp)

/* CLI: switch to pure kernel mode and disable interruptions */
/* STI: switch to pure kernel mode and enable interruptions */

do_* are C functions:

void do_bp(struct pt_regs *regs)
{
    die_if_kernel("do_bp in kernel",
                  regs);
    force_sig(SIGTRAP, current);
}
mem_init()

- Releases the free memory from memblock to the *buddy allocator* (aka page allocator)

Memory: 257916k/262144k available (1412k kernel code, 4228k reserved, 267k data, 84k bss, 169k init, 0k highmem)

Virtual kernel memory layout:

- *vmalloc*: 0xd0800000 - 0xffffffff (759 MB)
- *lowmem*: 0xc0000000 - 0xd0000000 (256 MB)
  - *.init*: 0xc01a5000 - 0xc01ba000 (84 kB)
  - *.data*: 0xc01621f8 - 0xc01a4fe0 (267 kB)
  - *.text*: 0xc00010c0 - 0xc01621f8 (1412 kB)

---

**Overview: memory management sequence**

1. Map kernel image ∴ *memory cannot be allocated*
2. Register memory banks in memblock ∴ *memory can be only reserved*
3. Map physical memory ∴ *memory can be allocated*
4. Release free memory to page allocator ∴ *pages can be allocated*
5. Start slab allocator and vmalloc infrastructure ∴ *kmalloc() and vmalloc()*
Scans device tree and finds all the nodes identified as interrupt controllers.

```c
icu: icu {
    compatible = "soclib,vci_icu";
    interrupt-controller;
    #interrupt-cells = <1>;
    reg = <0x0 0xf0000000 0x1000>;
};
```

→ First device driver!
time_init()

- Parses clock provider nodes
  
  ```
  clocks {
    freq: frequency@25MHz {
      #clock-cells = <0>;
      compatible = "fixed-clock";
      clock-frequency = <25000000>;
    }
  }
  ```

- Parses clocksource nodes
  - Clock-source device (monotonic counter)
  - Clock-event device (counts periods of time and raises interrupts)

→ Second device driver!
To init (1)

- Process management
  - Setting up the stack for new threads
  - Switching between threads (switch_to())
To init (2)

Page fault handler
- Catching memory faults

Freeing unused kernel memory: 116K (c022d000 - c024a000)
switch_mm: vaddr=0xcf8a8000, paddr=0x0f8a8000
IBE: ptr=0x0f8a8000, ietr=0x00001001, ibvar=0x00400000
DBE: ptr=0x0f8a8000, detr=0x00001002, dbvar=0x00401064

System calls
- List of system calls
- Enhancement of the interrupt and exception handler

Freeing unused kernel memory: 116K (c022d000 - c024a000)
Hello world!

Signal management
- Execution of signal handlers

User-space memory access
- Setting up the exception table

arch/tsar/include/asm/uaccess.h
get_user()
put_user()
Initial port to mono-processor system

Embedded distribution
- uClibc
- crosstool-ng
- Buildroot

```
io scheduler cfq registered (default)
SoClib VCI IOC driver, major=254
  ioc0: p1
console [ttyVTTYO] enabled
console [ttyVTTYO] enabled
bootconsole [early_tty_cons0] disabled
bootconsole [early_tty_cons0] disabled
Freeing unused kernel memory: 844K (c0269000 - c033c000)
ls
drwxr-xr-x 7 0 0 0 .
drwxr-xr-x 7 0 0 0 ..
lrwxrwxrwx 1 0 0 6 init -> bin/sh
drwxr-xr-x 2 0 0 0 mnt
drwxr-xr-x 2 0 0 0 lib
drwxr-xr-x 2 0 0 0 etc
drwxr-xr-x 2 0 0 0 dev
drwxr-xr-x 2 0 0 0 bin
uname -a
Linux (none) 3.13.0-00111-g8583fe8-dirty #503 SMP Wed Feb 25 11:06:47 CET 2015
tsar tsar
echo "I really like this presentation so far!"           I
I really like this presentation so far!
reboot
reboot: Restarting system
requested machine_restart[
```

```
$ sloccount arch/tsar

Total Physical Source Lines of Code (SLOC) = 4,840
...
Total Estimated Cost to Develop = $ 143,426
...
```
Atomic operations for multi-processor

Before SMP, IRQ disabling was enough to guarantee atomicity

```c
#include asm-generic/atomic.h:

static inline void atomic_clear_mask
  (unsigned long mask, atomic_t *v)
{
  unsigned long flags;
  mask = ~mask;
  raw_local_irq_save(flags);
  v->counter &= mask;
  raw_local_irq_restore(flags);
}
```

With SMP, need for hardware-enforced atomic operations

```c
#include arch/tsar/include/asm/atomic.h:

static inline void
__tsar_atomic_mask_clear
  (unsigned long mask, atomic_t *v)
{
  int tmp;
  smp_mb__before_llsc();
  __asm__ __volatile__ ("1: ll %[tmp], %[mem] \n"
" and %[tmp], %[mask] \n"
" sc %[tmp], %[mem] \n"
"   beqz %[tmp], 1b \n"
"   : [tmp] "=&r" (tmp),
   [mem] "+m" (v->counter)
   : [mask] "Ir" (mask));
  smp_mb__after_llsc();
}
```

Headers

bitops.h, barrier.h, atomic.h, cmpxchg.h, futex.h, spinlock.h, etc.
Inter-Processor Interrupt (IPI) support

IPI functions
- Reschedule
- Execute function
- Stop

XICU
Generic hardware interrupt, timer, mailbox (IPI) controller
SMP Boot sequence
(from the boot CPU’s point of view)

start_kernel
setup_arch
    smp_init_cpus*
smp_prepare_boot_cpu*
kernell_init
kernel_init_freeable
    smp_prepare_cpus*
do_pre_smp_initcalls
smp_init
    for_each_present_cpu(cpu) {
        cpu_up
            _cpu_up
                __cpu_up*
    }
smp_cpus_done*

- CPU discovery (from DT)
- idmap page table
- Spinlock vs. IPI boot
Multi-processor system

```
.data : 0xc02158a0 - 0xc0268b68  ( 332 kB)
.text : 0xc0041000 - 0xc02158a0  (1874 kB)
SLUB: HAlign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=64
Hierarchical RCU implementation.
   RCU restricting CPUs from NR_CPUS=256 to nr_cpu_ids=4.
NR_IRQS:64 nr_irqs:64 0
Console: colour dummy device 80x25
console [tty0] enabled
Calibrating delay using timer specific routine.. 102.13 BogoMIPS (lpj=204274)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 512
CPU_0_0_1: booting secondary processor (logical CPU1)
CPU_0_0_2: booting secondary processor (logical CPU2)
CPU_0_0_3: booting secondary processor (logical CPU3)
Brought up 4 CPUs
SMP: Total of 4 processors activated.
bio: create slab <bio-0> at 0
Switched to clocksource mips32_clksrc
Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
SoCLib VCI IOC driver, major=254
ioc0: p1
console [ttyVTTY0] enabled
```

$ sloccount arch/tsar

Total Physical Source Lines of Code (SLOC) = 6,543

(+35% compared to mono-processor support)
The full multi-node (NUMA) architecture

40-bit address format: \( X \quad Y \quad \text{node-local address} \)

39-36 35-32 31-0

40-bit address space:

<table>
<thead>
<tr>
<th>Node</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>0x0000000000</td>
</tr>
<tr>
<td>(0,1)</td>
<td>0x0100000000</td>
</tr>
<tr>
<td>(0,2)</td>
<td>0x0200000000</td>
</tr>
<tr>
<td>(1,0)</td>
<td>0x1000000000</td>
</tr>
<tr>
<td>(1,1)</td>
<td>0x1100000000</td>
</tr>
<tr>
<td>etc.</td>
<td>0x...</td>
</tr>
</tbody>
</table>
Memory mapping: Highmem support

- Map the first node
- Consider the other node as high memory
Multi-node interrupt network

- 1 XICU/node
- I/O PIC: transfers hardware interrupts to XICU mailboxes

Interrupt-controller evolution (SLOC)

1. ICU + timer controllers: 200
2. XICU (multi-processor): 500
3. XICU (multi-node): 800
Memory mapping: stacking

- Stack discontiguous memory in the direct mapping segment
- Tweaking of __va() and __pa()
Memory mapping: cap

If there is too much physical memory

1. Reduce the amount of mapped memory per node
2. Reduce the number of mapped node

Virtual Memory

user space

kernel space

Physical Memory

Direct Mapping

vmalloc  kmap

0GiB  3GiB  4GiB

vmalloc  kmap

kernel space

user space

0GiB  3GiB  4GiB

node (0,0) node (0,1) node (0,2) etc.

0x0000000000 0x0100000000 0x0200000000 etc.
Kernel code and rodata replication

- Replicate in nodes
- Round-robin strategy for patching new page tables with the different replicats

![Diagram of memory management](image)

- Direct Mapping
- `vmalloc` `kmap`

Virtual Memory

- `user space`
- `kernel space`

Physical Memory

- `node (0,0)`
- `node (0,1)`
- `node (0,2)`
- etc.

Physical Memory Addresses:

- `0x0000000000`
- `0x0100000000`
- `0x0200000000`
- `0x...`
Multi-node (NUMA) system

... Memory: 1553400K/1572864K available (2357K kernel code, 90K rwdta, 336K rodata, 856K init, 525K bss, 19464K reserved, 786432K highmem)

Virtual kernel memory layout:
- fixmap : 0xfebff000 - 0xffffffff (20480 kB)
- pkmap : 0xfee800000 - 0xfee00000 (2048 kB)
- vmalloc : 0xfe000000 - 0xfe7fe000 (223 MB)
- lowmem : 0xc0000000 - 0xf0000000 (768 MB) (cached)
  .init : 0xc02c2000 - 0xc0398000 (856 kB)
  .data : 0xc0256730 - 0xc02c1be8 (429 kB)
  .text : 0xc0009000 - 0xc0256730 (2357 kB)

SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=96, Nodes=64

... Brought up 96 CPUs
SMP: Total of 96 processors activated.

$ sloccount arch/tsar

Total Physical Source Lines of Code (SLOC) = 7,588

(+16% compared to multi-processor support)
Conclusion

Bad news 😞, good news 😊

- Boots in 4s with kernel replication, 3s without
- No runtime results
- Internship is starting today to resume this work

Bedtime reading

- Series of articles on LWN.net (mono-processor support)
  - The basics: https://lwn.net/Articles/654783/
  - The early code: https://lwn.net/Articles/656286/
  - To the finish line: https://lwn.net/Articles/657939/
Questions?