Linux on the PowerPC 4xx

David Gibson <david@gibson.dropbear.id.au>, IBM LTC OzLabs

September 4, 2002

Abstract

The 4xx series is a family of PowerPC processors designed for embedded applications. These processors are in the high end for the embedded space: they are reasonably powerful, although slow compared to a workstation processor, and they have an MMU which although basic compared to a general purpose CPU is sufficient to fully support a virtual memory operating system like Linux.

The 4xx processors are designed for building single board computers (SBC), so each has various peripheral devices (e.g. Ethernet controller, serial port, I^2C) built into the chip itself. This talk will look at how the Linux kernel takes advantage of these chips' features and works around their peculiarities. We will focus particularly on the 405GP, but also look at some of the features of other processors in the series.

1 Introduction: PowerPC 4xx processors

The PowerPC 4xx series is a family of processors made by IBM for various embedded applications. As embedded chips, they are considerably less powerful and featureful than the chips found in desktop or workstation machines. However the 4xx chips are in the high-end for the embedded space and can run a fully fledged modern operating system such as Linux.

In particular, while the 4xx's MMU (a software loaded TLB) is simpler that that found in many desktop CPUs, it is sufficient to implement full virtual memory. Furthermore as the name implies the 4xx chips are an implementation of the PowerPC architecture, so they implement all the standard PowerPC instructions and most code written for a "normal" PowerPC will also run on the 4xx (albeit slowly). The 4xx chips, unlike desktop PowerPC chips, do lack a floating-point unit, but this is essentially the only difference between a 4xx chip and a "normal" PowerPC which can affect non-kernel (unprivileged) code.

The 4xx chips are designed for building single board computers. As such, in addition to the processor core, each 4xx chip includes many of the components needed for a fully working computer, including a interrupt controller, SDRAM controller and various peripheral devices built into the chip itself. Exactly what peripherals are present varies from chip to chip, but can include:

- Ethernet controllers (excluding the PHY)
- serial ports
- PCI host adaptors
- USB controllers
- IDE controllers
- real time clock
- ...

These on-chip peripherals are built as separate logic cores on the same piece of silicon. They connect to the processor core through several specialised on-chip busses. Thus, it is relatively easy to design a new chip with a particular set of peripheral devices by combining a CPU core with the relevant extra logic cores.

The 4xx chips also contain extra support for debugging embedded devices. This consists of a JTAG connection to which an external hardware debugger can be attached. Software on another machine can then use this hardware debugger to single step the processor, examine or change the contents of registers and memory, set breakpoints and perform other debugging tasks. It is even possible to step through low-level interrupt handlers and kernel boot code. This is particularly useful for an embedded device, which may well lack the normal IO devices (screen, keyboard etc.) which might be used for debugging on a more conventional computer.

2 An example: the 405GP

As a more concrete example, let's examine the 405GP, a reasonably typical chip from the 4xx family. This has a 200MHz PowerPC CPU core, with:

- a software-loaded TLB (see §2.1)
- 16kB of instruction cache and 8kB of data cache
- an interrupt controller (UIC)
- 4kB of on-chip memory (as fast as cache)
- an SDRAM controller
- an expansion bus controller
- a DMA controller
- a "decompression controller" (which allows code to be stored in a compressed form in memory and transparently decompressed when executed)
- a PCI host bridge
- an 10/100 Ethernet controller
- two serial UARTs
- an I^2C controller
- a number of general purpose IO lines (GPIO)

and (of course) the buses to hook all these together.

2.1 The 405GP's MMU

The MMU consists of a fully associative TLB with 64 entries, which are loaded by the operating system using the special `tlbwe` (TLB Write Entry) instruction. Each entry contains a virtual page number and a context number (known as the translation ID or TID), a real page number (a.k.a. page frame number), a page size, a "valid" (V) bit and a number of access control and attribute bits for the page. The TLB supports a number of different page sizes, varying from 1kB to 16MB, which can be used simultaneously (i.e. each TLB entry can have a different page size).

2.2 Power management

Minimising the amount of power consumed is clearly a desirable goal for many embedded applications, so 4xx chips have a number of power saving features. To start with the 405GP has only a moderate basic power consumption - it can run at full speed (200MHz) without a heatsink or fan. In addition many of the on-board peripherals and components can be switched off or placed in a low-power state when not in use.

3 Linux Support

Linux supports a number of boards based on 4xx processors, including the "Walnut", "Oak" and "Ebony" boards (IBM reference development boards for the 405GP, 403GCX and 440GP respectively), the EP405 (a PC/104+ form-factor, 405GP based board made by Embedded Planet), the Tivo PVR and others. This section looks at how we deal with the various peculiarities of the 4xx within Linux. Again, we'll use the 405GP as a concrete example.

Of course many "real" (as opposed to prototype or demonstration) embedded applications will be operating on a custom designed board. Hence for a particular application it usually won't be sufficient to just compile the kernel - some tweaking will probably be necessary for the particular hardware in use. Often custom boards are quite similar to reference design boards though, so the amount of core kernel work involved in such a port should be quite small.

As we've seen, the 4xx chips do not have floating point instructions. This causes no problem for the kernel code itself, since floating-point is never used there. For user programs the kernel can emulate a floating point unit, in which case Linux on the 4xx runs exactly the same user binaries as any PowerPC Linux machine. If floating point emulation is disabled then user executables and libraries must be compiled specially to use software floating point.

Although there is some code for 4xx support in the standard kernels from Linus Torvalds and Marcelo Tosatti, it is incomplete and largely broken. The most developed 4xx support current exists in the `linuxppc_2_4_devel` BitKeeper tree which, as the name suggests, is a tree based on Marcelo's 2.4 kernel maintained for PowerPC Linux development. A lot of the code for embedded machines in this tree has been contributed by MontaVista Software Inc.

Currently support for the 4xx in 2.5 is lagging behind `linuxppc_2_4_devel`. The code is gradually being merged into 2.5 though, and standard 2.5 is certainly much closer to having working 4xx support than standard 2.4.

3.1 Handling a software MMU

Supporting a software loaded TLB under Linux turns out to be conceptually quite simple. The kernel maintains a set of page tables in much the same way as on any Linux system. When a TLB miss exception occurs, a small piece of assembly code walks through the page tables, finds the appropriate entry and if valid loads it into the TLB (replacing existing entries in a round-robin fashion). If there isn't a valid entry we call the page fault handler just as we would on page fault generated by a hardware MMU. In order to keep the TLB fault handler as small and fast as possible the page table entries are laid out so that they can be loaded straight into the TLB with as little additional processing as possible.

As on all other PowerPC machines, the kernel uses a 4kB page size for normal page mappings. However the 4xx's support for larger page sizes is used for some optimisations. The `linuxppc_2_4_devel` kernel has a configuration option to enable "pinned" TLB entries. This option loads two 16MB TLB entries at boot time, which are never invalidated or overwritten. These entries cover the first 32MB of system RAM, including the kernel text and static data structures. Without this, entering the kernel (e.g. for a system call) tends to flush out most user entries from the TLB with entries for the kernel, thus reducing performance. This also significantly reduces the number of TLB misses taken while executing code in the kernel. It does however slightly reduce the number of TLB entries available to user code, which can slow down memory accesses when using a reasonably large working set.

There also exists code, although it hasn't been committed to the `linuxppc_2_4_devel` tree yet, which uses an improved technique to take advantage of large page TLB entries. This uses special "large page" entries in the Linux page tables. Linux on PowerPC normally uses a two-level page table, but a "large page" entry is a specially marked entry in the top level page directory which directly describes a mapping of a 4MB region (the area normally covered by all the PTEs in a second level table). Mappings of 16MB regions can also be described by setting 4 consecutive page directory entries.

At boot time we set up large page entries to describe a mapping of physical RAM (again including the kernel text and most of its data structures). These entries can then be loaded directly into the TLB (using large page TLB entries) by special code in the TLB miss handler. This again reduces the TLB usage by the kernel, but in a more flexible way which doesn't preclude user code from using all 64 TLB entries when appropriate.

3.2 Cache coherency issues

4xx chips, unlike most PowerPC CPUs, don't do cache-coherent DMA. That is to say the data cache in 4xx chips doesn't snoop the bus for DMA transfers and update or invalidate itself appropriately. This means that if DMA transfers are occurring then the CPU's view of memory (i.e. what's in the data cache) and a peripheral device's view of memory can get out of sync. This means some special techniques are required when using DMA.

There are two approaches to handling this situation. The first is to specially allocate some memory for DMA and map it marked non-cacheable. With the cache disabled the CPU's and the device's view of the memory will be consistent and there are no further problems. This approach is simple, but disabling caching can introduce performance problems, and this can be inconvenient if a driver is handed blocks of data which have been allocated by other parts of the kernel (e.g. network socket buffers) - using this approach would require copying the data into a new, non-cacheable buffer.

The second approach is to use normal cacheable memory, but to perform explicit cache write-backs and invalidates where necessary to ensure consistency. Using this approach at any given time a block of memory will "belong" either to the CPU or to the device. When the buffer belongs to the CPU there must be no DMA active in to or out of it, and when it belongs to the device the CPU must not read or write the memory (i.e. it must not pull any of the buffer into cache). A driver can transfer ownership of the buffer by flushing the cache[1]. This approach does require that no DMA buffer share a cacheline with any other kernel data structure.

Which approach is the most suitable depends on the nature of the hardware, and sometimes it's desirable to use both in the same driver. For example, consider the driver for the 405GP's built-in Ethernet controller. The hardware uses tables of descriptors to describe buffers to transmit from or receive into. At initialisation time, the driver allocates some uncached memory for these tables - because the hardware can write or alter descriptors in the tables at essentially any time, it's impossible to use explicit cache flushes to manage them.

At initialisation the driver also prepares a pool of buffers to be used for receiving packets. It does this by allocating memory (with `dev_alloc_skb()`), then invalidating the cachelines in each buffer, so that the buffers now belong to the device. It then informs the device of the locations of the buffers by writing descriptors into the Rx descriptor table. When a packet is received by the hardware the following will happen:

1. The device receives the packet and DMAs the data into the next available Rx buffer.
2. The device alters the Rx descriptor table to indicate which buffer it has used and signals an interrupt.
3. The driver's interrupt handler parses the Rx descriptor table to find which buffers have been filled.

[1] Whether a writeback and invalidate or both is needed depends on whether the DMA is from memory, to memory or in both directions.

4. The driver reads the buffer, processes the packet and passes it to the network layer. Reading the buffer pulls it into cache, thus transferring "ownership" back to the CPU - this is safe because the device is now finished with the buffer.

5. If the pool of buffers is low, the driver prepares more buffers as at initialisation time.

A somewhat similar procedure is needed for transmitting packets (outgoing DMA). Once the driver has been passed a packet buffer by the network layer the following will happen:

1. The driver does a cache writeback on the cachelines the buffer occupies (the device now "owns" the buffer).

2. The driver writes a descriptor for the buffer to the Tx descriptor table.

3. The device reads the new descriptor and transmits the packet, DMAing from the buffer.

4. The device alters the descriptor to indicate that it is done and signals an interrupt.

5. The driver determines which buffers have been handled by the hardware and frees them.

The 4xx chips are not the only non-cache-coherent CPUs which Linux supports, so there already exists some infrastructure for handling this situation. For the first approach, the function `pci_alloc_consistent()` is used by PCI drivers to allocate memory which is DMA consistent - this will be ordinary memory on cache-coherent processors and non-cacheable memory on non-cache-coherent processors such as the 4xx. For the second approach, there are a variety of functions such as `pci_map_single()` and `pci_map_page()` which are used to perform the necessary cache operations[2]. On 4xx, these functions are implemented in terms of the architecture specific functions `consistent_alloc()` (for `pci_alloc_consistent()`) and `consistent_sync()` (for the `pci_map_*()` functions), which are also used by non-PCI drivers such as those for the 4xx's on-chip peripherals.

3.3 On chip peripherals

Linux drivers have been written for many of the 4xx on-chip peripherals. For example, on the 405GP, the PCI bridge, Ethernet controller, UARTs, I^2C controller and GPIO lines all have drivers. There is also a driver for the on-chip IDE controller which appears on some other 4xx chips.

Since the on-chip peripherals are built as independent logic cores, a number of 4xx chips may share an identical peripheral. So, it's obviously desirable to share a common driver for the device between these chips. However the on-chip buses to which the peripherals are attached do not support any type of scanning or device detection - the kernel has to "just know" what devices are there.

Because of this, a kernel must be built to run on a particular 4xx chip. Specifying the chip at compile time will (amongst other things) include an object into the build which contains a table listing the various on-chip peripherals, their addresses, interrupt assignments and so forth. This table is used to provide information to the device drivers for the on-chip peripherals, so that they can locate their devices. In general the drivers won't need to directly know the chip they're running on - this makes supporting new 4xx chips with similar on-chip peripherals much easier.

The on-chip devices and their drivers are connected together by some code known as the OCP (for On-Chip Peripheral) subsystem. This keeps track of what devices are present and which drivers are active.

[2]On machines with an IO MMU or similar these functions also establish the necessary mappings there.

3.4 Board and initialisation issues

In addition to the devices on the 4xx chip itself, each board a 4xx chip is used on usually has some special devices itself (if nothing else boards frequently include an FPGA or CPLD with some control registers). The firmware, if any, on these boards is usually limited to some basic initialisation and loading, and also provides no support for hardware detection.

Hence, a kernel must generally be built for a specific board, as well as for a specific chip. Specifying a particular board in the kernel configuration will include an object file related to the board into the kernel. This will provide several board-specific initialisation functions, in particular `board_init()`, which is called quite early during boot by code generic to all 4xx chips.

3.5 Remaining problems and development to do

As usual, of course, there is still work to be done in fully supporting 4xx chips.

One of the biggest areas for work on 4xx support is in power management: at the moment Linux has only quite limited support for the power saving features of the 4xx chips. In particular the drivers for a number of the on-chip devices need updating to make use of those device's features for saving power.

A number of things in the 4xx support, particularly the handling of on-chip devices and initialisation for specific boards, are not implemented as cleanly as they could be. The situation is gradually improving though, which should make life easier when adding support for new 4xx chips and boards.

As discussed previously, the support for 4xx chips in the 2.5 kernel is quite incomplete at the moment. A large part of the reason being that people building particular embedded devices tend to work from 2.4 because they want to use a stable kernel. However, integrating 4xx support into the mainline 2.5 kernel will be important for making 4xx devices "full citizens" of the Linux world. One of the major tasks for 2.5 is to integrate the handling of 4xx on-chip peripherals into the new unified device model being implemented in 2.5.

There are some other aspects of the 4xx chips which are not supported and which it's unclear how to support, or even whether supporting them would be useful:

Currently Linux makes no use of the 405GP's on-chip memory (OCM)[3]. Since this memory is as fast as cache, it might be possible to improve performance by doing so. However, the OCM is only 4kB so not a lot can fit in there. What to put in there to see any benefit will obviously depend on what the machine is actually being used for - and it's quite likely that the most interesting code or data from this point of view will be in user space. Hence using the OCM really has to be implemented by the integrator of a particular embedded device. That said, it might be useful to provide a device driver which allows the OCM to be mapped into userspace.

Acknowledgements

Thanks must go to IBM, my employer, for their support of this paper. Thanks to Martin Schwenke and Paul Mackerras for proofreading.

References

[1] `linuxppc_2_4_devel` kernel tree, `bk://ppc@ppc.bkbits.net/linuxppc_2_4_devel`.

[2] IBM Corporation, "PowerPC 405GP Embedded Processor User's Manual", Seventh Preliminary Edition, 2000.

[3] Patrick Mochel, Linux Kernel Driver Model Documentation", `http://www.kernel.org/pub/linux/kernel/people/mochel/doc/`, 2001.

[3] Well, actually, it is used in a few instances for obtaining parameters from the firmware/loader at boot time, but there is no substantial use.