Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

Porting Drivers to HP ZX1

Grant Grundler
Linux Development Lab
Hewlett Packard
Cupertino, CA, USA, 95014
grundler@cup.hp.com

Abstract

"Porting" doesn't accurately describe how one gets a Linux driver to run on different architectures. If a driver doesn't "just work," generally it's a matter of figuring out which wrong assumptions about the HW (or OS) are embedded in the driver. The goal of this talk is to describe the *HP ZX1* IO subsystem and some of the wrong assumptions I've found in 2.4.17 kernel drivers.

A Block Diagram of the ZX1 IO subsystem is quite similar to current PA-RISC systems. In contrast to Intel Itanium boxes, neither supports legacy x86 IO space. For booting, *EFI drivers* (ugh, DOS is back) are required and IA32 Expansion ROMs are ignored. *Platform Services* must be used for DMA mapping, interrupts, PCI device discovery. I'll discuss how those services are different between HP's ZX1 platform and my (weak) understanding of IA32. Fortunately, use of these services is the same between both architectures.

I was surprised by which drivers did not *Just Work* (e.g. tulip, acenic) and will talk about why they didn't. Primarily, the timing of CPU interaction with IO devices is different. ZX1 IO subsystem is also less tolerant of driver "quirks"—things that are wrong but other platforms don't puke on. Lastly, I'll explain what an MCA is and how it's useful for debugging IO driver problems.

1 HP ZX1 IO Subsystem

The HP ZX1 chip set doesn't have many surprises to folks who've worked on RISC systems. Other architectures including PA-RISC, Alpha, and SPARC have similar block diagrams. The main parts of HP's implementation are the *System Bus Adapter* (SBA) and *Lower Bus Adapter* (LBA).

From a very high level, most IO subsystems aren't that different since PCI bus behaviors are defined by the various PCI specifications. IO Interrupts (IRQ Line), IO Port, and MMIO functionality provided have the same semantics as on IA32. This is good since it makes it possible to write (mostly) portable drivers.

The SBA provides an IO MMU, memory controller, and interconnect between the *ropes* bus and *McKinley* bus. The IO MMU design is based on the implementation used in PA-RISC workstations and low end servers. Two significant differences is how 64-bit DMA addressing is supported and cache coherency model. Other less obvious differences are greater bandwidth of both the McKinley bus and ropes bus.

The LBA is the PCI Host bus adapter and also contains the IO SAPIC. Unlike its PA-RISC predecessor, this LBA supports PCI-X (133MHz, 64-bit). The IO SAPIC was also used in PA-RISC platforms. I'm still amazed that 80% of the code is the same between the IA64 and PA-RISC implementations. Because of NDAs with Intel, both implementations were developed completely independently inside HP and published on the same day (Feb 3, 2000) when the IA64 source tree was published. (See `http://lists.parisc-linux.org/hypermail/parisc-linux-cvs/2860.html`).

This type of architecture has some clear performance advantages over legacy North/South bridge topology in IA32 systems and also introduces some new issues. The performance advantage is more raw IO bandwidth between IO, memory, and CPU which exceeds the single PCI bus model by orders of magnitude. Some obvious problems are ordering of transactions (e.g. IRQ vs. DMA), DMA latency, and PIO latency.

2 DMA Mapping

Use of PCI DMA mapping services is required for several reasons:

- **A**ddress Translation: The primary purpose is to provide a device view of memory for DMA.
- **3**2-bit DMA: IO MMU provides 32-bit devices that ability to DMA into memory which lives above 4GB address boundary. This provides at least 3x better performance than SW for block IO.
- **P**ortability: The old interface, `virt_to_bus()`, could only support systems w/o IOMMU or the IOMMU could map all of host memory statically. HP ZX1 IOMMU can only map 1GB at a time. That's not as bad as it sounds since only 32-bit PCI devices are required to use the IO MMU. 64-bit PCI devices (capable of DAC) can bypass the IO MMU.

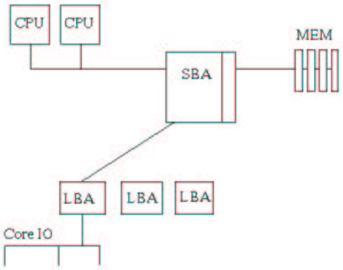

Figure 1: HP ZX1 architecture

3 Interrupts

`request_irq()` works the same as before. What's really different from IA32 is the number and type of IRQs available. IA64 defines 256 vectors vs the woefully inadequate 15 in legacy IA32. The following sections describe some of the high level behaviors of IO SAPIC and implementations.

3.1 Message Signalled Interrupts

As far as I can tell, no one is using this. At least not directly. The IO SAPIC translates the line based IRQ into a transaction on the "upstream" bus. The Local SAPIC in the CPU is the target of this transaction and is identified by its *EID*. The data portion of the transaction identifies which interrupt vector is being delivered.

System Firmware assigns EIDs and initializes the Local SAPICs. The IO SAPIC driver reads the Interrupt Routing Table from ACPI. This table describes how each of the 4 IRQ *Pins* from each PCI device or slot is routed to a particular IO SAPIC IRQ Line. When a device driver registers its interrupt handler via `request_irq()`, the IO SAPIC driver programs the IRTE (an internal IO SAPIC register) for the IRQ input line.

Note that Foster CPU is the first IA32 CPU to use IO xAPIC (˜= SAPIC) and Local xAPIC. Support for IO xAPIC was only recently added to i386 arch in order to properly distribute IRQs across CPUs. All architectures with IO xAPICs (PA-RISC, IA64, IA32) direct interrupts at specific CPUs. None use XTPR transactions to enable the HW to redirect interrupts to a "lower priority" CPU. For IA64 and IA32, this is by design to preserve driver state in the CPU cache associated with a given PCI interface card. PA-RISC has no Local SAPIC or XTPR support and consumes the interrupt transaction directly.

So why talk about MSI? There's several good reasons for devices to use MSI:

- **I**nterrupt Code Path: It allows the driver interrupt to be directly called from the trap handler—no traversing lists or lookup tables. Typically though, a layer of indirection is only needed if the HW can't generate an EOI to the IO SAPIC or the IRQ Line is shared.

- **E**xclusive Vector: The driver can avoid shared PCI IRQ line and the the resulting shared vector. IO SAPIC implementations to date typically only have 7 IRQ lines—not really enough if the PCI bus hosts multiple devices/slots.

- **D**MA ordering: Normally, when the IRQ is a line, it bypasses the normal DMA data path. Thus race conditions exist where a DMA might not reach memory before the IRQ is delivered and acted upon. For PCs and the like this isn't a problem since all the IO paths are short.

  For larger systems, this can be a problem. When the interrupt is a transaction on the bus, PCI ordering rules prevent it from bypassing any inbound DMA transaction. Thus, when the interrupt finally reaches the CPU, one can be certain all DMA has reached memory as well and not stuck in any coalescing buffers between the IO device and the memory it was writing to. Thus one doesn't need any additional magic to guarantee the in-flight DMA is coherent with CPU caches.

- **T**arget multiple CPUs: This is wish list. Given the right services, a smart device can target transaction completions at different CPUs by generating interrupt transactions for specific Local SAPICs. The goal is to service the interrupt on the same CPU that initiated the transaction. Tradeoffs between driver D-cache footprint and interrupt latency would help determine applications for this. Clustering folks I've talked were looking at this but didn't prototype anything to test it out.

3.2 More than 256 Interrupts?

Large systems (32 CPU and up) can end up using more than 256 vectors and exhaust the Interrupt Vector Table. One example is cc-Numa machines where one really doesn't want to (or can't) deliver interrupts across the fabric. Linux can gracefully work around this issue by defining *IRQ Regions*. For IA64, each region could represent a different Interrupt Vector Table. PA-RISC uses IRQ regions for every level of the interrupt handling that has to decode a bit mask or handle an array of IRQs.

The ACPI Interrupt Routing Tables may not need to collude if arch specific code can correctly direct interrupt transactions generated by IO SAPIC to the targeted CPU. I'm not familiar with details of ccNUMA support but know it has been done.

4 Posted vs Non-Posted Writes

Nearly all linux drivers started out using *IO Port* address space since ISA/EISA was the standard when linux was born. On IA32, special instructions (yes, most of you know this already) exist to access this alternative address space.

What's key here is IO Port space also has different semantics. One similarity is reads and writes to either IO Port or MMIO space do not interact with CPU Cache. A subtle difference is writes to IO Port space are *Non-Postable*. This means **t**he CPU stalls waiting for the transaction to complete.

4.1 IO Port space sucks

IO Port space has several serious issues:

- **I**SA Aliasing: Most of IO port address space isn't available because of ISA compatibility where to many ISA devices only support 10 (or more) address lines and alias everything above that.
- **L**egacy IO: serial, timers, fd, parallel and a host of other devices occupy de-facto standard addresses in IO port space.
- **M**ore Registers: Many new devices require more register space. Either more mail boxes or on-board RAM. Just isn't room for a 4k or bigger shared RAM in IO port space. Maybe for single devices, but I've been told that's not useful when 4 or more cards need to be installed in the system.
- **D**evice Discovery: For devices which don't have legacy addresses assigned, they had to poke around in IO port space to discovery where their devices where. Fortunately with PCI, that's no longer necessary though some drivers still do that for ISA compatibility.

Combined, all of these issues have encouraged nearly all PCI devices to move to MMIO space regardless of the Non-Post-able semantics.

4.2 Memory Mapped IO is better

Since PCI has become a standard, many PCI devices support both IO Port space and MMIO address space to provide compatibility and a transition for drivers to use MMIO space. And that transition has been taking place. In implementing this transition, many driver writers assume MMIO is the same as IO port space and there's just more of it. **T**hat's wrong.

MMIO space is *Post-able*. The CPU writes the data and just continues doing other work. The CPU may not even wait for the transaction to hit the *Central Bus* (aka Front Side Bus) before continuing. This is good. It means a burst of writes are exactly that.

4.3 MMIO is harder to get Right

Even **g**ood driver writers get MMIO space usage wrong.

This is from acenic, a "mature" driver. But here is an example of this wrong assumption:

```
  writel(local, &regs->LocalCtrl);
  udelay(ACE_LONG_DELAY);
  mb();
  local |= EEPROM_CLK_OUT;
  writel(local, &regs->LocalCtrl);
```

The problem is the CPU starts executing the udelay() before the data reaches the device. The `writel()`s are timing sensitive. And the `mb()` is orthogonal to the udelay(). Switching the order around shouldn't change things. The fix is **a**dd a `readl()` after the first `writel()`. PCI transaction ordering rules require the write get pushed to the device before the read. Since the CPU has to wait for the read return, the write is effectively flushed. We don't care what the read returns in this case.

In all fairness, Jes Sorensen caught what was going on right away and accepted my patch. I added 35 readl() calls. He did gripe about my formatting. That's OK. That's Jes and it's his driver.

Dave Miller was a slightly harder sell for a patch to tg3. Jeff Garzik caught on right away and provided Dave with the explanation that I somehow didn't.

```
http://linux.bkbits.net:8080
/linux-2.4/cset@1.383.17.6
?nav=index.html|ChangeSet@-4w|
```

tg3 driver got 3 more reads for similar issues. One was a slightly different case and worth noting:

```
  tw32(RX_CPU_BASE + CPU_STATE,
       0xffffffff);
  tw32(RX_CPU_BASE + CPU_MODE,
       0x00000000);
+
+ /* Flush posted writes. */
+ tr32(RX_CPU_BASE + CPU_MODE);

  return 0;
```

Code after the return was expecting the `CPU_MODE` to have been cleared already. I got lazy and stopped trying to figure out what.

4.4 CPU v.s. IO Timing Trend

The speed of the CPU is getting much faster than the IO path is. HP likes high bandwidth bridges that favor bandwidth over MMIO access. Thus while a problem may not be visible on a 2GHz Pentium, it will show up on on 800 or 1GHz HP ZX1 system. And probably on other systems from SGI, SUN or IBM.

In the case of current HP ZX1 platforms, the *System Bus Adapter* (aka SBA) and *Lower Bus Adapter* (aka LBA, PCI-X Controller) both have FIFOs to queue data in both directions. The fact the a MMIO transaction has to cross 3 busses to get to a device (Central, internal, PCI) is a good hint that timing is going to be longer than on systems with only one or two busses.

One example of different timing was exposed in the tulip driver where it resets the *Phy* (DP83840A or LXT971D). No issue exists with this code using the same 100BT cards on 400 MHz PA-RISC. The issue showed up occasionally on 550MHz PA-RISC and consistently on faster HP ZX1 platforms. HP 100BT products needed the patch that appears in Figure 4.4.

Though this works, I want to be clear the patch is wrong. I discovered this worked and submitted the patch before I found and read the respective product data sheets. The right fix is to poll the phy after the `reset_sequence` until an "in-reset" bit clears. Then one should wait about 500 microseconds before sending the `init_sequence`.

4.5 MMIO Reads are expensive

The last time I measured the cost of an MMIO read on a 400MHz PA-8500 system, I got something around 500-600 CPU cycles. The same measurement on an 800 MHz HP ZX1 system was around 900-1000 CPU cycles. PCI bus traces from a 450MHz PII system suggested the MMIO read time was in the same ball park.
Conclusion: **MMIO reads are expensive.**

For an example of MMIO read avoidance, see

```
http://cvs.parisc-linux.org
/linux/arch/parisc/kernel
/sba_iommu.c?rev=1.66
```

and search for `DELAYED_RESOURCE_CNT`. This code only works because MMIO writes are *Post-able*.

4.6 Soft Fail v.s. Hard Fail

The first time we tried the bcm5700 driver it came up and started talking on the LAN. I was impressed until I tried to *ifconfig eth0 down* the NIC. The system MCA'd. Using MCA state dump, I was able to determine the address which failed to respond was a register on the BCM5701 chip.

After tracing through lots of code, we finally figured out what was happening. The bcm5700 driver was resetting the card twice during the `close(2)`. And the bcm5700 chip wasn't being re-enabled on the PCI bus after the second reset. The MCA occurs after the `close(2)` when a request for statistics tries to read data from the now defunct BCM5701 chip. Bad driver. Don't do that. HP implements *Hard Fail* in its chipsets. HP engineers decided it's better to crash a server if it's known the drivers do not properly handle failed reads (return -1 typically).

The Intel Itanium systems don't crash running the bcm5700. I gather tradional PCs implement *Soft Fail* since it seems to be OK to get garbage back from failed MMIO reads. I suspect it's because the problem will look like a SW problem (which it is) and not a HW problem. I.e. the HW vendor doesn't have to take the support call and doesn't look worse than its competitors.

AFAIK, LBA supports this mode of operation as well but can only be enabled by modifying kernel source. I like using HW to expose SW problems. I don't expect this to change.

Figure 2: Incorrect patch for HP 100BT products

```
diff -u -p -r1.2 media.c
--- drivers/net/tulip/media.c   25 Jan 2002 20:14:57 -0000      1.2
+++ drivers/net/tulip/media.c   25 Mar 2002 19:57:19 -0000
@@ -284,6 +284,10 @@ void tulip_select_media(struct net_devic
                for (i = 0; i < init_length; i++)
                        outl(init_sequence[i], ioaddr + CSR12);
        }
+
+       (void) inl(ioaddr + CSR6); /* flush CSR12 writes */
+       udelay(500);            /* Give MII time to recover */
+
        tmp_info = get_u16(&misc_info[1]);
        if (tmp_info)
                tp->advertising[phy_num] = tmp_info | 1;
```

5 BIOS vs EFI drivers

Some drivers (e.g. VGA, megaraid) depend on expansion BIOS to initialize and fire up the card before the linux driver sees it. The previous Itanium platform EFI emulates x86 and supports the x86 BIOSs. For better or worse, HP decided to drive the migration to EFI at the risk of backwards compatibility. In order to work on HP ZX1 systems, an EFI "driver" must be provided to do the same thing. To date, all the IO card vendors that supply HP have committed to providing such a driver and I know they are delivering or have delivered.

6 iDebugging IO driver crash

You wrote a driver and tried it on an HP ZX1 box. It crashed. Welcome to hell …just kidding. Like PA-RISC systems, IA64 platforms provide a crash state under several circumstances. MCA and INIT are two of those circumstances that are interesting for developers. For the PA-RISC literate, MCA and INIT roughly equate to *HPMC* and *TOC* respectively.

6.1 Intro to `errdump MCA`

MCAs will occur anytime an error signal is broadcast on the McKinley bus. For driver problems, this is typically a CPU read time out. CPU read timeouts occur when a dereferenced MMIO address don't return before a timer in the CPU expires. Since MMIO writes are *Posted*, normally the *victim* is a MMIO read even if a MMIO write caused the error.

Two cases can cause this: either the PCI device stopped responding (e.g. firmware died, chip locked up, MMIO BAR disabled) or a DMA was attempted to an invalid address. The former cases can typically be debugged with printk and knowing which address caused the dump.

One can view the MCA dump with `errdump MCA` command at the EFI shell. Once the MCA data is captured and saved, it's usually a good idea to `errdump clear`. IA-64 Linux will print this dump on the next boot. That's ˜1000 lines of output in a less friendly format. And make sure to save the matching System.map in order to look up symbols. When loading kernel modules, squirrel away the dynamically linked symbols too.

Here is what some of the fields mean:

- `IIP` is the current Instruction Pointer when the system noticed the error.
- `XIP` is the IIP of the most recent trap or interrupt occurred.
- `Requestor ID` is the ID of the originator of a transaction.
- `Responder ID` is the ID of the device that responded with data
- `Target ID` is IO address we are trying to reach.

6.2 Intro to `errdump INIT`

An INIT is used like an NMI. It resets the machine and saves the current state. HP ZX1 platforms have a small blue button in the back of the box label which can be used to generate an INIT. Like an MCA, similar data gets stored.

To be honest, I've never used an INIT and only know of it. Problems I tend to chase are MCAs and not lockups.

7 Acknowledgements

I've learned a lot from folks in the HPUX community when I worked on it and continue to learn from them. I dare not name names for fear of retribution.

And for the past two years, I've been learning new things from (in no particular order): Lamont "NMU" Jones, Ryan Bradetich, Matthew Wilcox, Martin Petersen, Paul Bame, Bdale Garbee, Jes Sorensen, Dave Miller, and a host of other Open Source kernel and application hackers.

More information about IA64-linux can be found at:

```
http://www.linuxia64.org/
http://www.hp.com/
```