Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

A Symphony of Flavours: Using the device tree to describe embedded hardware

Grant Likely
Secret Lab
grant.likely@secretlab.ca

Josh Boyer
IBM
jwboyer@linux.vnet.ibm.com

Abstract

As part of the merger of 32-bit and 64-bit PowerPC support in the kernel, the decision was to also standardize the firmware interface by using an OpenFirmware-style device tree for all PowerPC platforms; server, desktop, and embedded. Up to this point, most PowerPC embedded systems were using an inflexible and fragile, board-specific data structure to pass data between the boot loader and the kernel. The move to using a device tree is expected to simplify and generalize the PowerPC boot sequence.

This paper discusses the implications of using a device tree in the context of embedded systems. We'll cover the current state of device tree support in *arch/powerpc*, and both the advantages and disadvantages for embedded system support.

1 Background

We could go on for days talking about how embedded systems differ from desktops or servers. However, this paper is interested in one particular aspect: the method used by the operating system to determine the hardware configuration.

In general, desktop and server computers are engineered to be compatible with existing software. The expectation is that the operating system should not need to be recompiled every time new hardware is added. Standardized firmware interfaces ensure that the boot loader can boot the operating system and pass it important details such as memory size and console device. PCs have the BIOS. PowerPC and Sparc systems typically use OpenFirmware. Commodity hardware is also designed to be probeable by the OS so that the full configuration of the system can be detected by the kernel.

The embedded world is different. Systems vary wildly, and since the software is customized for the system, there isn't the same market pressure to standardize firmware interfaces. You can see this reflected in the boot schemes used by embedded Linux. Often the operating system is compiled for a specific board (platform) with the boot loader providing minimal information about the hardware layout, and the platform initialization code is hard coded with the system configuration.

Similarly, data that is provided by the boot firmware is often laid out in an ad-hoc manner specific to the board port. The old embedded PowerPC support in the kernel (found in the `arch/ppc` subdirectory) uses a particularly bad method for transferring data between the boot loader and the kernel. A structure called `bd_info`, which is defined in `include/asm-ppc/ppcboot.h`, defines the layout of the data provided by the boot loader. `#defines` are used within the structure to add platform-specific fields, but there is no mechanism to describe which `bd_info` layout is passed to the kernel or what board is present. Changes to the layout of `bd_info` must be made in both the firmware and the kernel source trees at the same time. Therefore, the kernel can only ever be configured and compiled for a single platform at a time.

When the decision was made to merge 32-bit (`arch/ppc`) and 64-bit (`arch/ppc64`) PowerPC support in the kernel, it was also decided to use the opportunity to clean up the firmware interface. For `arch/powerpc` (the merged architecture tree), all PowerPC platforms must now provide an OpenFirmware-style device tree to the kernel at boot time. The kernel reads the device tree data to determine the exact hardware configuration of the platform.

```
/ {        // the root node
  an-empty-property;
  a-child-node {
    array-prop = <0x100 32>;
    string-prop = "hello, world";
  };
  another-child-node {
    binary-prop = [0102CAFE];
    string-list = "yes","no","maybe";
  };
};
```

Figure 1: Simple example of the `.dts` file format

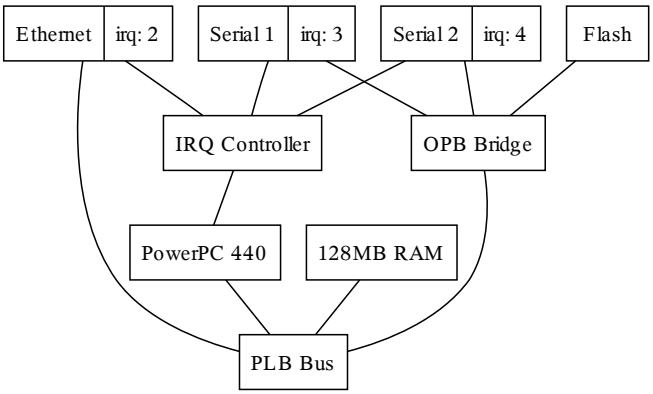

Figure 2: Example PowerPC 440 System

2 Description of Device Trees

In the simplest terms, a device tree is a data structure that describes the hardware configuration. It includes information about the CPUs, memory banks, buses, and peripherals. The operating system is able to parse the data structure at boot time and use it to make decisions about how to configure the kernel and which device drivers to load.

The data structure itself is organized as a tree with a single root node named `/`. Each node has a name and may have any number of child nodes. Nodes can also have an optional set of named property values containing arbitrary data.

The format of data contained within the device tree closely follows the conventions already established by IEEE standard 1275. While this paper covers a basic layout of the device tree data, it is strongly recommended that Linux BSP developers reference the original IEEE standard 1275 documentation and other OpenFirmware resources. [1][2]

The device tree source (`.dts`) format is used to express device trees in human-editable format. The device tree compiler tool (`dtc`) can be used to translate device trees between the `.dts` format and the binary device tree blob (`.dtb`) format needed by an operating system. Figure 1 is an example of a tree in `.dts` format. Details of the device tree blob data format can be found in the kernel's Documentation directory. [3]

For illustrative purposes, let's take a simple example of a machine and create a device tree representation of the various components within it. Our example system is shown in Figure 2.

```
/dts-v1/
/ {
  model = "acme,simple-board";
  compatible = "acme,simple-board";
  #address-cells = <1>;
  #size-cells = <1>;

  // Child nodes go here
};
```

Figure 3: Example system root node

It should be noted that the device tree does not need to be an exhaustive list of all devices in the system. It is optional to itemize devices attached to probable buses such as PCI and USB because the operating system already has a reliable method for discovering them.

2.1 The root Node

The start of the tree is called the root node. The root node for our simple machine is shown in Figure 3. The `model` and `compatible` properties contain the exact name of the platform in the form `<mfg>,<board>`, where `<mfg>` is the system vendor, and `<board>` is the board model. This string is a globally unique identifier for the board model. The compatible property is not explicitly required; however, it can be useful when two boards are similar in hardware setup. We will discuss compatible values more in Section 2.4.1.

2.2 The cpus Node

The `cpus` node is a child of the root node and it has a child node for each CPU in the system. There are no ex-

```
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    device_type = "cpu";
    model = "PowerPC,440GP";
    reg = <0>;
    // 400MHz clocks
    clock-frequency = <400000000>;
    timebase-frequency = <400000000>;
    i-cache-line-size = <32>;
    d-cache-line-size = <32>;
    i-cache-size = <32768>;
    d-cache-size = <32768>;
  };
};
```

Figure 4: `cpus` node

plicitly required properties for this node; however, it is often good practice to specify `#address-cells=<1>`, and `#size-cells=<0>`. This specifies the format for the *reg* property of the individual CPU nodes, which is used to encode the physical CPU number.

CPU nodes contain properties for each CPU on the board. The unit name for CPU nodes is in the form `cpu@0` and it should have a `model` property to describe the CPU type. CPU nodes have properties to specify the core frequency, L1 cache information, and timer clock frequency. Figure 4 is the `cpus` node for our sample system.

2.3 System Memory

The node that describes the memory for a board is, unsurprisingly, called a *memory node*. It is most common to have a single memory node that describes all of the memory ranges and is a child of the root node. The `reg` property is used to define one or more physical address ranges of usable memory. Our example system has 128 MiB of memory, so the memory node would look like Figure 5.

2.4 Devices

A hierarchy of nodes is used to describe both the buses and devices in the system. Each bus and device in the system gets its own node in the device tree. The node for

```
memory {
  device_type = "memory";
  // 128MB of RAM based at address 0
  reg = <0x0 0x08000000>;
};
```

Figure 5: Memory node

the processor local bus is typically a direct child of the root node. Devices and bridges attached to the local bus are children of the local bus node. Figure 6 shows the hierarchy of device nodes for the sample system. This hierarchy shows an interrupt controller, an Ethernet device, and an OPB bridge attached to the PLB bus. Two serial devices and a Flash device are attached to the OPB bus.

2.4.1 The compatible property

You'll notice that every node in the device hierarchy has a `compatible` property. `compatible` is the key that an OS uses to decide what device a node is describing. In general, compatible strings should be in the form `<manufacturer>,<part-num>`. For each unique set of `compatible` values, there should be a *device tree binding* defined for the device. The binding documents what hardware the node describes and what additional properties can be defined to fully describe the configuration of the device. Typically, bindings for new devices are documented in the Linux Documentation directory in `booting-without-of.txt` [3].

You'll also notice that sometimes `compatible` is a list of strings. If a device is register-level compatible with an older device, then it can specify both its compatible string and the string for the older device, so that an operating system knows that the device is compatible with an older device driver. These strings should be ordered with the specific device first, followed by a list of compatible devices. For example, the flash device in the simple system claims compatibility with `cfi-flash`, which is the string for CFI-compliant NOR flash chips.

2.4.2 Addressing

The device address is specified with the `reg` property. `reg` is an array of *cell* values. In device tree terminol-

```
plb {
  compatible = "simple-bus";
  #address-cells = <1>;
  #size-cells = <1>;
  ranges;
  UIC0: interrupt-controller {
    compatible = "ibm,uic-440gp",
                 "ibm,uic";
    interrupt-controller;
    #interrupt-cells = <2>;
  };
  ethernet@20000 {
    compatible = "ibm,emac-440gp";
    reg = <0x20000 0x70>;
    interrupt-parent = <&UIC0>;
    interrupts = <0 4>;
  };
  opb {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0xe0000000
              0x20000000>;
    serial@0 {
      compatible = "ns16550";
      reg = <0x0 0x10>;
      interrupt-parent = <&UIC0>;
      interrupts = <1 4>;
    };
    serial@10000 {
      compatible = "ns16550";
      reg = <0x10000 0x10>;
      interrupt-parent = <&UIC0>;
      interrupts = <2 4>;
    };
    flash@1ff00000 {
      compatible = "amd,s29gl256n",
                   "cfi-flash";
      reg = <0x1ff00000 0x100000>;
    };
  };
};
```

Figure 6: Simple System Device Hierarchy

ogy, cells are simply 32-bit values. Array properties like `reg` are arrays of cell values. Each `reg` property is a list of one or more address tuples on which the device can be accessed. The tuple consists of the base address of the region and the region size.

```
reg = <base1 size1 [base2 size2 [...]]>;
```

The actual size of each `reg` tuple is defined by the parent nodes' `#address-cells` and `#size-cells` properties. `#address-cells` is the number of cells used to specify a base address. Similarly, `#size-cells` is the number of cells used to specify a region size. The number of cells used by `reg` must be a multiple of `#address-cells` plus `#size-cells`.

It is important to note that `reg` defines *bus addresses*, not *system addresses*. The *bus address* is local to the bus that the device resides on, or in device tree terms, the address is local to the parent of the node. Buses in turn can map bus addresses up to their parent using the `ranges` property. The format of `ranges` is:

```
ranges = <addr1 parent1 size1 [...]>;
```

Where *addr* is a bus address and is `#address-cells` wide, *parent* is an address on the parent bus and is the parent node's `#address-cells` wide. *size* is the parent node's `#size-cells` wide.

Buses that provide a 1:1 mapping between bus address and parent address can forgo the explicit mapping described above and simply specify an empty `ranges` property:

```
ranges;
```

In this example system, the Flash device is at address 0x1ff00000 on the OPB bus, and the OPB bus specifies that PLB bus address 0xe0000000 is mapped to address 0x0000000 on the OPB bus. Therefore, the Flash device can be found at base address 0xfff00000.

2.5 Interrupts and Interrupt Controllers

The natural layout of a tree structure is perfect for describing simple hierarchies between devices, but is not particularly suitable for capturing complex interconnections.

Interrupt signals are a good example of additional linkages. For example, it is correct to describe the serial device in our sample system as a child of the OPB bus. However, it is also correct to say that it is a child of the interrupt controller device, so how should this be described in the device tree? Established convention says that the natural tree structure should be used to describe the primary interface for addressing and controlling the devices. Secondary connections can then be described

with an explicit link between nodes called a *phandle*. A *phandle* is simply a property in one node that contains a pointer to another node.

For the case of interrupt connections, device nodes use the `interrupt-parent` and `interrupts` properties to describe a connection to an interrupt controller. `interrupt-parent` is a phandle to the node that describes the interrupt controller and `interrupts` is a list of interrupt signals on the interrupt controller that the device can raise.

Interrupt controller nodes must define an empty property called `interrupt-controller`. They also must define `#interrupt-cells` as the number of cells required to specify a single interrupt signal on the interrupt controller, similar to how `#address-cells` specifies the number of cells required for an address value.

Many systems, particularly SoC systems, only have one interrupt controller, but more than one can be cascaded together. The links between interrupt controllers and devices form the *interrupt tree*.

Referring back to the serial device node, the property `interrupt-parent` defines the link between the node and its interrupt parent in the interrupt tree.

The `interrupts` property defines the specific interrupt identifier. Its format depends on its interrupt parent's `#interrupt-cells` property and the values contained within are specific to that interrupt domain. A common binding for this property when the parent's `#interrupt-cells` property is 2 is to have the first cell represent the hardware interrupt number for the device in the interrupt controller, followed by its level/sense information.

2.6 Special Nodes

2.6.1 The chosen Node

Firmware often needs to pass non-hardware types of information to the client OS, such as console port, and boot arguments. The node that describes this kind of information is called the `/chosen` node. There are no required properties for this node; however, it is quite useful for defining the board setup. If our example system booted from a USB key and used the serial port as the console, the `/chosen` node might look like Figure 7.

```
chosen {
  bootargs = "root=/dev/sda1 rw ip=off";
  linux,stdout-path =
      "/plb/opb/serial@10000";
};
```

Figure 7: Chosen node

```
  aliases {
    console = "/plb/opb/serial@10000";
    ethernet0 = "/plb/ethernet@20000";
    serial0 = "/plb/opb/serial@0";
    serial1 = "/plb/opb/serial@10000";
  };
```

Figure 8: Aliases node

2.6.2 aliases

In order to ease device lookup in client operating systems, it is often desirable to define an `aliases` node. This allows one to provide a shorthand method for identifying a device without having to specify the full path on lookup. This is typically only done for the more common devices on a board, such as Ethernet or serial ports. Figure 8 provides an example.

The types of nodes and properties that can be contained in a device tree are as varied as the hardware that they describe. As hardware designers invent new and creative ways of designing components, these unique properties will continue to grow. However, the direction is to be as general as possible to allow commonality between the various hardware components across these boards. Given the flexible nature of the device tree concept, the hope is that the client operating systems will be able to adapt to new hardware designs with a minimal amount of code churn and allow the most possible reuse of code.

3 Usage of Device Tree in Linux Kernel

3.1 Early Boot

To understand how the kernel makes use of the device tree, we will start with a brief overview of the `arch/powerpc` boot sequence. `arch/powerpc` provides a single entry point used by all PowerPC platforms. The kernel expects a pointer to the device tree

blob in memory to be in register r3 before jumping to the kernel entry point.[1]

The kernel first does some basic CPU and memory initialization, and then it tries to determine what kind of platform it is running on. Each supported platform defines a `machdep_calls` structure. The kernel calls `probe_machine()`, which walks through the `machine_desc` table, calling the `.probe()` hook for each one. Each `.probe()` hook examines the device tree and returns true if it decides that the tree describes a board supported by that platform code. Typically, a probe will look at the `compatible` property on the root node of the tree to make the decision, but it is free to look at any other property in the tree. When a probe hook returns true, `probe_machine()` stops iterating over the table and the boot process continues.

3.2 Device initialization

In most regards, using the device tree has little impact on the rest of the boot sequence. Platform code registers devices into the device model and device drivers bind to them. Probable buses like PCI and USB probe for devices and have no need for the device tree. Sequence-wise, the boot process doesn't look much different, so the real impact is not on the sequence, but rather on where the kernel obtains information from about peripherals attached to the system.

The interesting questions, then, revolve around how the platform code determines what devices are present and how it registers them with the kernel.

3.2.1 `of_platform` bus

Currently, most embedded platforms using the device tree take advantage of the `of_platform` bus infrastructure. Like the `platform` bus, the `of_platform` bus doesn't represent a hardware bus. It is a software construct for manually registering devices into the device model; this is useful for hardware which cannot be probed. Platform code[2] can use the `of_platform_bus_probe()` convenience function to iterate over a part of the device tree and register a `struct of_device` for each device. Device drivers in turn register a `struct of_platform_driver`, and the `of_platform` infrastructure matches drivers to devices.

The core of both the `platform` and `of_platform` buses is almost identical, which begs the question of why do two separate software buses exist in the first place? The primary reason is that they use different methods to match devices to drivers. The platform bus simply matches a device and a driver if they share the same `.name` property. The `of_platform` bus instead matches drivers to devices on the basis of property values in the tree; in particular, the driver provides a match table of `name`, `device_type`, and `compatible` properties' values. When the values in one of the table entries match the values in an `of_device`, then the bus calls the driver's probe hook.

3.2.2 platform bus adapters

While the `of_platform` bus is often convenient, it is by no means mandated or the only way to retrieve device information out of the device tree. Some drivers already exist with platform bus bindings and the developers have decided not to rework the binding to use `of_platform`. Rather, a helper function is used to search the device tree for nodes with the appropriate property values. When interesting nodes are found, the function creates a new `struct platform_device`, populates it with data from the tree node, and registers it with the platform bus. Several examples of this can be seen in `arch/powerpc/syslib/fsl_soc.c`. As of this writing, the Freescale Gianfar, USB host controller, and I2C device drivers work this way.

There is some debate amongst the Linux PowerPC hackers over whether to merge the `of_platform` bus functionality back into the platform bus. Doing so would eliminate a lot of duplicated code between them, but it leaves the question of how to match drivers and devices. The following are some of the options:

[1] Actually, this is not entirely true. If the boot firmware provides an Open Firmware-compatible client interface API, then the kernel first executes the `prom_init()` trampoline function to extract the device tree from the firmware before jumping into the common entry point.

[2] Not to be confused with the *platform bus*. *Platform code* in this context refers to the support code for a particular hardware platform and can be found in the `arch/powerpc/platforms` subdirectory.

Teach platform bus about device tree matching. If the platform drivers could optionally supply an OF match table, it could be used if the platform device also had a pointer to a device node in the tree. The downside is that it increases the complexity of platform devices, but these are intended to be simple constructs. It is uncertain whether this approach would be acceptable to the platform bus maintainers.

Translate between nodes properties and platform bus names. This approach has a minimal amount of impact on existing platform bus drivers. However, it requires the match table to also supply functions for populating `pdata` structures from the data in the device tree. Also, device-tree-to-platform-bus translation must occur at boot time and not at module load time, which means that the binding data must be contained within the driver module. Besides, device registration is supposed to be under the control of platform code. It is poor form for drivers to register their own platform devices.

Make drivers search the device tree directly. This solves the problem of data about matching devices being separate from the device driver, but it doesn't work so well because there is no easy way to prevent multiple drivers from binding against the same node. Once again, it is bad form for drivers to register their own devices.

3.2.3 Other Methods

Of course, not all initialization fits simply within the `platform/of_platform` bus model. Initialization of interrupt controllers is a good example, since such controllers are initialized directly from one of the platform code hooks and do not touch the driver model at all. Another example is devices that are logically described by more than one node within the device tree. For instance, consider an audio device consisting of a node for the I2S bus and another node for the CODEC device. In this case, each node cannot be probed independently by separate drivers. The platform code most likely needs to interpret the tree data to create device registrations useful to the device drivers.

The key here is that the device tree is simply a data structure. Its sole purpose is to describe the hardware layout and it does not dictate kernel architecture. Platform code and device drivers are free to query any part of the tree they desire to make appropriate decisions.

At this point it is worth mentioning that it can be a strong temptation to design new device tree bindings around what is convenient for the device drivers. The problem is that what might seem like a good approach when you start writing a device driver often turns out to be just the first of several bad ideas before you finish it. By keeping the device tree design focused on hardware description alone, it decouples it from the driver design and makes it easier to change the driver approach at some point in the future. There are literally decades of Open Firmware conventions to help you design appropriate bindings.

4 Case Studies

4.1 PowerPC 440 SoCs

The PowerPC 440 chip is a widely used SoC that comes in many different variations. In addition to the PPC 440 core, the chip contains devices such as 16550-compatible serial ports, on-board Ethernet, PCI host bridge, i2c, GPIO, and NAND flash controllers. While the actual devices on the various flavors of the 440 are typically identical, the quantity and location of them in the memory map is very diverse. This lends itself quite well to the device tree concept.

In `arch/ppc`, each PPC 440 board had its own unique board file, and described the MMIO resources for its devices as a set of `#define` directives in unique header files. There were some attempts to provide common code to share among board ports; however, the amount of code duplication across the architecture was quite large. A typical board file was around 200 lines of C code. The code base was manageable and fairly well maintained, but finding a fairly complete view of the interaction among the files was at times challenging.

When contrasted with the `arch/powerpc` port for PPC 440, some of the benefits of the device tree method are quickly revealed. The average board file is around 65 lines of C code. There are some boards that have no explicit board file at all, as they simply reuse one from a similar board. Board ports have become relatively easy to do, often taking someone familiar with device trees a relatively short time to complete base support for a

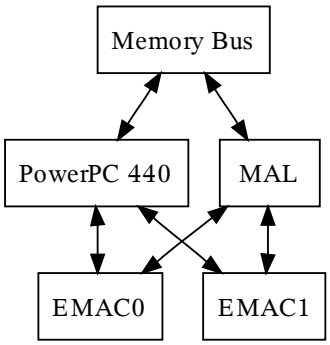

Figure 9: Logical EMAC/MAL connections

board.[3] Additionally, multiplatform support makes it possible to have a single `vmlinux` binary that will run on any of the PPC 440 boards currently supported today. This was virtually impossible before device trees were used to provide the device resources to the kernel.

However, using a device tree does present unique challenges at times. Situations arise that require the introduction of new properties or using different methods of defining the interaction between nodes. For PowerPC 440, one of those cases is the MAL and EMAC nodes. The MAL and EMAC combined comprise the on-board Ethernet. A simplified view of the interconnects is shown in Figure 9.

The MAL device has a number of channels for transmit and receive. These channels are what the various EMAC instances use for their transmit and receive operations. The MAL also has 5 interrupts, but not all these interrupts go to a single interrupt parent. These issues required some more complex concepts and new properties to be applied in the device tree.

To solve the multiple interrupt parent problem for the MAL, an *interrupt map* was used. In this situation, the MAL node's `interrupt-parent` property is set to itself, and the `interrupts` property simply lists interrupts 0–5. Recall that this is possible because the representation of that property is dependent upon the node's `interrupt-parent` property, which in this case is the MAL itself. To properly map the real interrupts to the appropriate controllers, the `interrupt-map` property is used. In this property, each MAL-specific interrupt is mapped to its proper interrupt parent using the interrupt domain for that parent. Figure 10 shows the device tree node for the MAL. Here you can see that MAL interrupt 0 maps to UIC0 interrupt 10, and so on.

```
MAL0: mcmal {
  compatible = "ibm,mcmal-440gp",
    "ibm,mcmal";
  dcr-reg = <0x180 0x62>;
  num-tx-chans = <4>;
  num-rx-chans = <4>;
  interrupt-parent = <&MAL0>;
  interrupts = <0 1 2 3 4>;
  #interrupt-cells = <1>;
  interrupt-map = <
   /*TXEOB*/ 0 &UIC0 0xa 4
   /*RXEOB*/ 1 &UIC0 0xb 4
   /*SERR*/  2 &UIC1 0 4
   /*TXDE*/  3 &UIC1 1 4
   /*RXDE*/  4 &UIC1 2 4>;
  interrupt-map-mask = <0xffffffff>;
};

EMAC0: ethernet@40000800 {
  device_type = "network";
  compatible = "ibm,emac-440gp",
    "ibm,emac";
  interrupt-parent = <&UIC1>;
  interrupts = <1c 4 1d 4>;
  reg = <40000800 0x70>;
  local-mac-address = [000000000000];
  mal-device = <&MAL0>;
  mal-tx-channel = <0 1>;
  mal-rx-channel = <0>;
};
```

Figure 10: PowerPC 440 MAL and EMAC nodes

You'll also notice in Figure 10 that there are some new MAL-specific properties introduced for the total number of transmit and receive channels. When looking at the EMAC node in Figure 10, you will see that there are new properties specific to the MAL as well. Namely, the `mal-device` property is used to specify which MAL this particular EMAC connects to, pointing back to the phandle of the MAL node. The `mal-tx-channel` and `mal-rx-channel` properties are used to specify which channels within that MAL are used. The device driver for the on-board Ethernet parses these properties to correctly configure the MAL and EMAC devices.

While this is certainly not the most complex interaction between devices that can be found, it does illus-

[3] The AMCC PowerPC 440EPx Yosemite board support was completed in 5 hours, with the majority of the work coming from rearranging the existing 440EPx support and adapting a new DTS file.

trate how a more interesting setup can be accomplished using the device tree. For those interested in further examples of complex setups, including multiple bridges and unique address ranges, the `arch/powerpc/boot/dts/ebony.dts` file found in the kernel would be a good starting point.

4.2 Linux on Xilinx Spartan and Virtex FPGA platforms

Xilinx has two interesting FPGA platforms which can support Linux. The *Spartan* and *Virtex* devices both support the *Microblaze* soft CPU that can be synthesized within the FPGA fabric. In addition, some of the *Virtex* devices include one or more dedicated PowerPC 405 CPU cores. In both cases, the CPU is attached to peripherals which are synthesized inside the FPGA fabric. Changing peripheral layout is a simple matter of replacing the bitstream file used to program the FPGA.

The FPGA bitstream file is compiled from VHDL, Verilog, and system layout files by Xilinx's *Embedded Development Kit* tool chain. Historically, EDK also generated an include file called `xparameters.h` which contains a set of `#define` statements that describes what peripherals are present and how they are configured. Unfortunately, using `#defines` to describe the hardware causes the kernel to be hard coded for a particular version of the bitstream. If the FPGA design changes, then the kernel needs to be recompiled.

Just like other platforms, migrating to `arch/powerpc` means that Virtex PowerPC support must adopt the device tree. Fortunately, the device tree model is particularly suited to the dynamic nature of an FPGA platform. By formalizing the hardware description into the device tree, the kernel code (and therefore the compiled image) is decoupled from the hardware design—particularly useful now that hardware engineers have learned software's trick of changing everything with a single line of source code.

On the other hand, the burden of tracking changes in the hardware design is simply shifted from making changes in the source code to making changes in the device tree source file (`.dts`). For most embedded platforms, the `.dts` file is written and maintained by hand, which is not a large burden when the hardware is stable and few changes are needed once it is written. The burden becomes much greater in the FPGA environment if every change to the bitstream requires a manual audit of the design to identify device tree impacts.

Fortunately, the FPGA tool chain itself provides a solution. The FPGA design files already describe the system CPUs, buses, and peripherals in a tree structure. Since the FPGA tool chain makes significant use of the TCL language, it is possible to write a script that inspects EDK's internal representation of the system design and emits a well formed `dts` file for the current design. Such a tool has been written; it is called *gen-mhs-devtree* [**?**] and it is in the process of being officially integrated into the EDK tool chain.

Figure 11 shows an example of a device tree node generated by an instance of version 1.00.b of the `opb_uartlite` ipcore. As you can see, the node includes the typical `compatible`, `reg`, and `interrupt` properties, but it also includes a set of properties with the *xlnx,* prefix. These properties are the ipcore configuration parameters extracted from the FPGA design. The device tree has made it possible to provide this data to the operating system in a simple and extensible way.

```
RS232_Uart_1: serial@40400000 {
  compatible =
      "xlnx,opb-uartlite-1.00.b";
  device_type = "serial";
  interrupt-parent = <&opb_intc_0>;
  interrupts = < 4 0 >;
  port-number = <0>;
  reg = < 40400000 10000 >;
  xlnx,baudrate = <2580>;
  xlnx,clk-freq = <5f5e100>;
  xlnx,data-bits = <8>;
  xlnx,odd-parity = <0>;
  xlnx,use-parity = <0>;
} ;
```

Figure 11: node generated by `gen-mhs-devtree`

With the success of the device tree for Xilinx PowerPC designs, Xilinx has decided to also adopt the device tree mechanism for Microblaze designs. Since many of the Xilinx peripherals are available for both PowerPC and Microblaze designs anyway, it was the natural choice to use the same mechanism for describing the hardware so that driver support can be shared by both architectures.

5 Tradeoffs and Critique

5.1 kernel size

One of the often-heard concerns of using the device tree method is how much larger the kernel binary will be. The common theory is that by adding all the device tree probing code, and any other "glue" code to make the board-specific drivers function with the generic kernel infrastructure, the overall `vmlinux` binary size will drastically increase. Combine this with having to store the dtb for the board and pass it in memory to the kernel, and one could see why this might be a concern for embedded targets.

While it is certainly true that the size of the `vmlinux` binary does grow, the actual differences are not as large as one may think. Let's examine the sizes of an `arch/ppc` and an `arch/powerpc` `vmlinux` binary using feature-equivalent kernel configs for minimal support with a 2.6.25-rc9 source tree.[4] Table 1 shows the resulting binary size for each arch tree.

Arch	Text	Data	BSS	Total
ppc	2218957	111300	82124	2412381
powerpc	2226529	139564	94204	2460297

Table 1: Section sizes of `vmlinux` binaries

As you can see, the overhead for the device tree method in this case is approximately 47KiB. Add in the additional 5KiB for the dtb file, and the total overhead for a bootable kernel is approximately 52KiB.

While to some this may seem like quite a bit of growth for such a simple configuration, it is important to keep in mind that this brings in the base OpenFirmware parsing code that would be required for any `arch/powerpc` port. Each device driver would have some overhead when compared to its `arch/ppc` equivalent; however, this would be a fairly small percentage overall. This can be seen when examining the `vmlinux` size for a multiplatform `arch/powerpc` config file. This config builds a kernel that runs on 6 additional boards, with support for 5 additional CPU types, and adds the MTD subsystem and OF driver. The resulting `vmlinux` adds approximately 130 KiB of overhead when compared to the single-board `arch/ppc` config. A comparison with a similar multiplatform config in `arch/ppc` cannot be done, as there is no multiplatform support in that tree.

[4] Essentially, the kernel was configured for BOOTP autoconf using an NFS rootfilesystem for the PowerPC 440GP Ebony evaluation board.

5.2 Multiplatform Kernels

Another question that is often heard is "But why do I care if I can boot one `vmlinux` on multiple boards? I only care about one board!" The answer to that, in short, is that most people probably don't care at all. That is particularly true of people who are building a single embedded product that will only ever have one configuration. However, there are some benefits to having the ability to create such kernels.

One group that obviously benefits from this are the upstream kernel maintainers. When changing generic code or adding new features, it is much simpler to build a multiplatform kernel and test it across a variety of boards than it is to build individual kernels for each board. This is helpful for not only the architecture maintainers, but anyone doing wider cross-arch build testing.

It's important to note that for most of the embedded boards, there is nothing that precludes building a single board config. Indeed, there are approximately 35 such defconfigs for the Freescale and IBM/AMCC embedded CPUs alone. However, doing a "buildall" regression build of this takes quite a long time, and the majority of it is spent building the same drivers, filesystems, and generic kernel code. By using a multiplatform defconfig, you only need to build the common bits once and the board-specific bits are built all together.

So while not everyone agrees that multiplatform embedded kernels are useful, the current direction is to put emphasis on making sure new board ports don't break this model. The hope is that it will be easier for the maintainers to adapt existing code, perform cleanups, and do better build test regressions.

References

[1] `http://playground.sun.com/1275/home.html`, Accessed on June 24, 2008.

[2] `http://www.openfirmware.org`, Accessed on June 24, 2008.

[3] Benjamin Herrenschmidt, Becky Bruce, *et al.* `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/powerpc/booting-without-of.txt`, Retrieved on June 21, 2008.