Chapter 5

Abstract design of the PE system

In this chapter, we analyze how we can generalize the representation of PEs at a very abstract level. We define a set of classes and relationships that define the characteristics of PEs and how they interact with each other. This design loosely coupled with the final implementation as no specific operating system nor programming language is considered at this point. Chapters 6 and 7 will build upon this design to make it concrete to a specific operating system.

5.1 Class model

In order to generalize the representation of PEs, we must start by defining an abstract class model. Figure 5.1 shows our extremely simple diagram that represents all the necessary components.

The purpose of each class is detailed below:

**PE** The class that abstracts the internal management details of a PEs. This consists of a set of attributes, described in detail in section 6.4, and a set of operations, explained in section 6.5. This is the core of all the design.

**PPE, SPE, etc.** The specialization of the PE class, representing a specific type of PE. Each unit can define its own data fields and its own operations. This specialization is marked as disjoint because the classes cannot overlap, and incomplete because we are not listing all possible processor types.
5.2 Machine description

With the ability to represent PEs, we have to express the relationships between them. These relations form what we call the *machine description*: an abstraction that expresses how all PEs interact among themselves.

There are two kinds of relations: one links a set of PEs in a list, in which the links have no purpose other than keeping all the elements related; i.e. the links have no meaning on their own. These are *plain relations*. The other kind of relations are those that define some interaction between PEs; these are defined as graphs and are named *directed relations*.

5.2.1 Plain relations

**boot** This plain relation links together all PEs that can boot the system. Maybe not all of them have been used to boot it, but they have all the necessary capabilities to do so. These are the ones that we have called CPUs so far.

**everything** This is the most basic relation and links all available PEs without expressing any differences among them. Such a list is needed whenever a generic kernel subsystem has to treat all PEs as a whole. For example, when we need to print user-friendly information of all units in the system or when we have to look for a specific unit given its textual name. It is also the entry point for algorithms that search for a PE that matches some characteristics. This relation is depicted in figure 5.2.

![Machine description: list of all possible PEs.](image)

**online** This is a subset of the *everything* relation and links those PEs that are currently online: i.e. those that are active and ready to execute threads.
Figure 5.3: Machine description: scheduled-by relation.

Figure 5.4: Machine description: siblings relation.
Chapter 6

Design realization of the PE system

In chapter 5, we saw the abstract design of our PE system. In this one, we adapt that design to the Linux kernel and the C programming language, which forces us to change the design due to the current kernel’s programming constraints.

6.1 Overall structure

We have already learned in section 4.2 that CPUs are currently identified by an integral number and that there are several bitmaps that indicate their availability. Foreign PEs are completely unknown to this scheme, so we shall see whether it is good to integrate their representation in the current scheme or to invent another level of abstraction.

The first approach we considered was to morph foreign PEs into the current representation of CPUs, exposing the former as if they were regular CPUs but with some restrictions. This involves assigning an unused identifier to each foreign PEs and registering it in the existing CPU maps. Here are its pros (+) and cons (-):

- (+) We can reuse the existing CPU bitmaps and the percpu framework. We only need to assign unused integral numbers – those set to 0 in the possible CPUs map – to the foreign PEs.

- (-) Forces us to use the CPU name for everything. Foreign PEs will be referred to as CPUs, which may be very confusing (specially in user space). CPU is a very commonly
For example: we could introduce a DEFINE_PER_PE() macro that simply wrapped the existing DEFINE_PER_CPU() to do its job. Even they could end up doing the same, the intentions of the programmer could be clear at first sight.

Given all the above, it seems interesting to follow the second approach: define a new abstraction level to represent all PEs in the system. This way we do not have to worry about messing the current representation of CPUs because our work will be done separately from it. We will be migrating existing machine-specific code to the new design as we go.

6.2 Basic concepts

The PE system is the name of our novel scheme to abstractly represent any PE in a machine. This section describes the fundamentals of this system as well as how we mapped the abstract design to the programming constraints imposed by Linux.

6.2.1 Identifiers

In the name of efficiency, the best way to identify elements in a set is to use a data type that is native to the machine. This rules out strings and the only option left is to use integral numbers. Furthermore, using integers allows us to quickly find the PE-specific data by looking it up in an array.

We assign an integral identifier to each PE. To maintain compatibility with the current system to manage CPUs, we respect the CPUs' identifiers up to the maximum number of possible CPUs in the PE system; this way we can keep a 1:1 mapping between PE identifiers and CPU identifiers. Foreign PEs are assigned numbers after the last possible CPU.

This idea is depicted in figure 6.1. We have a set of PPEs and SPEs on a Cell-based machine. There can be four PPEs, one of which is missing; this means that the system has reserved four entries in the CPU maps to represent this condition, so we reserve those same four entries in the PE maps. Above the fourth identifier, we are free to assign new numbers to other PEs as we wish, as we shall not maintain any kind of compatibility with any other subsystem. In the
these bitmaps are suited to represent relations between PEs—such as their interconnections or dependencies—so this is one more reason to add support for them.

We define the `pemask_t` type to abstract the representation of a PE bitmap. These bitmaps are sized according to the `NR_PES` constant defined at build time, which specifies the maximum number of PEs supported in the system.

Aside from the requirement to have support for bitmaps, we need to abstract the current CPU bitmaps described in section 4.2.2. Therefore, we will have these standard PE bitmaps:

**Possible PEs** This bitmap is built on top of the possible CPUs bitmap. It specifies all the PEs that may be present at some time in the system, even if they are not currently installed at a given time.

This bitmap is split in two parts: the first one matches exactly the contents of the possible CPUs bitmap because we can know beforehand which CPUs are possible and which are not by inspecting the machine's hardware configuration. However, the second part has all of its bits set to one because foreign PEs can be initialized at any time during system operation and we cannot predict which these will be. Hence, all the PEs are candidates for being present during the run time of the system.

We considered dropping this bitmap completely because of what has been explained above: the set of possible PEs is not known at boot time because PEs may appear in any hot-plugging device (such as a Peripheral Component Interconnect (PCI) card). However, it is useful to keep it for compatibility with the CPU management code and for the per-PE data fields.

To mitigate the problem of having too much bits set to one in this bitmap's second part, we provide the system user the ability to tune the maximum number of PEs at boot time. This way we can ship a kernel with a large `NR_PES` value built in but avoid wasting a lot of memory when defining per-PE data.

This group is represented by the `pe_possible_map` bitmap.
6.2. BASIC CONCEPTS

Figure 6.2: Example of the PE maps on a motherboard with 2 sockets and a PCI card with two foreign PEs.

Consider the example in figure 6.3. This shows the links between a PE and all those PEs that can handle the exceptions the former generates.

Figure 6.3: Representation of relationships between PEs.

Similarly, figure 6.4 illustrates how the machine description is expressed in terms of bitmaps. It shows a single relation, the managed-by relationship (the counterpart of in-charge-of), among four different PEs. These PEs are reachable from the online map and, once there, one can easily navigate to all the other units connected by the relation.

6.2.4 ISA identifiers

As already outlined in section 4.3.5, we need to keep track of the ISAs each PE supports. In order to implement this, we could define a new type and a custom list of all known ISA types. The
6.3. BASIC DATA TYPES

ISA from ELF (pe.isa_from_elf) Returns a pe.isa_t value deduced from an ELF file based on its e_machine header field. Its signature:

\[ \text{pe.isa_from_elf : unsigned int \to pe.isa_t} \]

This conversion function looks up values according to table 6.1.

<table>
<thead>
<tr>
<th>a.out machine type</th>
<th>PE ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_386</td>
<td>EM_386</td>
</tr>
<tr>
<td>M_68010</td>
<td>EM_68K</td>
</tr>
<tr>
<td>M_68020</td>
<td>EM_68K</td>
</tr>
<tr>
<td>M_MIPS1</td>
<td>EM_MIPS_RS3_LE</td>
</tr>
<tr>
<td>M_MIPS2</td>
<td>EM_MIPS_RS4_BE</td>
</tr>
<tr>
<td>M_SPARC</td>
<td>EM_SPARC</td>
</tr>
</tbody>
</table>

Table 6.1: Conversion table between a.out machine specifications and PE ones.

6.3 Basic data types

The following list shows several data types used to represent the data fields of each PE:

pe_id_t An identifier for a PE. Described in detail in section 6.2.1.

pe.isa_t An identifier for an ISA. At the moment this is just a wrapper over the ELF machine definition values such as EM_PPC or EM_SPU, but this type is used to abstract the type from the real binary format.

pe.name_t Represents a PE’s textual, user-friendly name. For example: ppu0 or spu5. However, PEs always have a generic name derived from their identifier; e.g. pe2 or pe7.

pe.type_t An identifier for the PE’s type. These numbers are assigned at run time and are not known beforehand.

pemask_t A bitmap representing a set of PEs. Described in detail in section 6.2.2.

struct pe.context An abstract representation for the PE’s execution context. Each PE must extend this structure to represent its context, so this will always be used in the form of a pointer.
operation : (pe_id_t, type1, ..., typeN) → typeZ

We define the following operations for each PE:

Creation of the idle thread (create_idle) Initializes the idle thread of the unit. That is, the program that it will run when it has no real work to do. Its signature:

create_idle : (pe_id_t) → ⊥

Textual description (describe) Prints a textual description of the unit. This is useful to fill the /proc/cpuinfo file from a platform-independent part of the code. Its signature:

describe : (pe_id_t, struct seq_file *) → ⊥

Start execution (start) Starts the processor in case it was not processing instructions. Its signature:

start : (pe_id_t) → ⊥

Stop execution (stop) Stops the processor so that it does not process more instruction. Calling this operation does not necessarily stop the processor immediately; the unit may need some time to finalize the execution of the current instruction before it is completely stopped. Its signature:

stop : (pe_id_t) → ⊥

Reinitialization (reset) Reinitializes the unit so that it acquires a know state. Its signature:

reset : (pe_id_t) → ⊥

Single stepping (single_step) Executes a single operation on the processor and then stops. Its signature:

single_step : (pe_id_t) → ⊥

Save the execution context (context_save) Takes a snapshot of the unit’s current execution context. For the data to be valid, the unit should have been stopped before executing this operation. In some situations this function may not be needed, but this is a required operation in case a PE is scheduling processes for another PE in a preemptive way. Its signature:
6.7 Dynamic registration of PEs

The PE system is a system-wide subsystem. Every module in the kernel that manages PEs is in charge of registering and unregistering the PEs it has discovered into the PE system's tables. This way, all other subsystems in the kernel can transparently access the new PEs and can get the appropriate notifications to be aware of the hardware changes. Figure 6.5 illustrates this idea: several hardware-specific subsystems register units into the PE system, and several other generic subsystems access their information through the PE system.

![Diagram showing the relations between the PE system, the modules that register PEs, and the modules that access them.]

Figure 6.5: Relations between the PE system, the modules that register PEs, and the modules that access them.

In order to register a PE, we need to know, at the very least, its system-wide name and its list of operations. However, the identifier can be assigned automatically in many cases and thus is generally not needed. With this in mind, four public operations are provided:

**Registration of a PE (pe_register)** Registers a PE into the system. This function takes the name of the unit and the its list of operations. It later assigns an identifier to the new processor and returns it to the caller for further usage. Its signature looks like:
6.7. DYNAMIC REGISTRATION OF PES

6.7.1 Notifications

Kernel subsystems can maintain persistent per-PE data on their own. These data are often initialized at the same time that the subsystem is initialized. However, given the dynamic nature of a heterogeneous multiprocessor system, PEs may be registered at a much later time and hence they may be unknown to the subsystem.

For example: the process scheduler is in charge of maintaining the per-PE run queues. These queues are initialized at very early stages of the boot process, much earlier than the SPEs are recognized and registered.

In order to resolve this situation, we need to introduce support for notifications of PE registration and unregistration. Whenever a PE is registered into the system, all interested parties must be notified of this hardware change; similarly, when it is unregistered, the contrary notification must happen to free any related resources.

We use Linux's notifier framework to handle these events reporting. We export two operations to add and remove notifiers from the system:

**Register a notifier** (pe.register.notifier) This call adds a notifier to the PE system. When a new notifier is added, it is called for each unit already registered in the system. It will also be executed for any further registration. Its prototype is:

```
pe.register.notifier : (struct notifier_block *) \rightarrow \perp
```

**Unregister a notifier** (pe.unregister.notifier) This function removes a notifier from the PE system. Its prototype is:

```
pe.unregister.notifier : (struct notifier_block *) \rightarrow \perp
```
Chapter 7

Implementation of the PE system

Chapter 6 detailed how we map the PE system design made in chapter 5 to the internals and coding constraints of the Linux kernel. In this chapter we analyze the changes we finally made to the code and show how they behave in a real machine.

It is important to say that, due to time constraints, we were unable to implement the complete PE system. However, we wrote the code that composes the core functionality of the system and later put it into use in some visible areas of the kernel so that we could check their behavior.

7.1 PE bitmaps

The first thing we did was to add a `include/linux/pemask.h` file that defines the `pemask_t` type. This file is mostly a duplicate of `include/linux/cpumask.h` and mimics its behavior as much as possible.

7.2 perpe framework

To implement the perpe framework, we created some files that provide a very similar functionality to the percpu one. In fact, the two share most of the implementation as perpe is built on top of percpu. This is suboptimal because perpe is a more generic concept that percpu, so future versions should reverse this.
7.4. GENERALIZATION OF PPES

- new_pe_type().
- delete_pe_type().
- register_pe().
- unregister_pe().
- pe_isa_from_aout().
- pe_isa_from_elf().
- register_pe_notifier().
- unregister_pe_notifier().

7.4 Generalization of PPEs

To generalize the representation of PPEs in a Cell machine we added a new file in the PowerPC directory: arch/powerpc/kernel/ppe.c. This implements the generic PE operations for PPEs processors and a function that is called after basic system initialization to properly register all boot CPUs into the PE systems. This may be suboptimal but is the easiest way we found to make this work.

7.5 Generalization of SPEs

To generalize the representation of SPEs in a Cell machine we modified the core SPE management file, arch/powerpc/platforms/cell/spu_base.c, and added the definition of the PE MI-operations in there.

7.6 Results

After our preliminary prototype, we were able to generalize two system-wide areas that only dealt with PPEs before. Of course, we could have gone on and generalized more subsystems, but that was too complex to be achieved on time. However, our PE system has proven useful to do this kind of changes, so we could keep modifying subsystems until all the systems were converted.
processor : ppe1 (pe1)
cpu : Cell Broadband Engine, altivec supported
clock : 3200.000000MHz
revision : 5.0 (pvr 0070 0500)

processor : spe0 (pe2)
spu : 0
local store : vaddr 0xd000080080050000, paddr 0x200000000000

processor : spe1 (pe3)
spu : 1
local store : vaddr 0xd0000800800d0000, paddr 0x200000080000

processor : spe2 (pe4)
spu : 2
local store : vaddr 0xd000080080150000, paddr 0x200001000000

processor : spe3 (pe5)
spu : 3
local store : vaddr 0xd0000800801d0000, paddr 0x200001800000

processor : spe4 (pe6)
spu : 4
local store : vaddr 0xd000080080250000, paddr 0x200002000000

processor : spe5 (pe7)
spu : 5
local store : vaddr 0xd0000800802d0000, paddr 0x200002800000
As can be seen, we also added a new file, called name, to these directories that shows the user-visible name of each PE. This allows the user to easily distinguish the type of each unit.
Part III

Process scheduling