Chapter 8

Process scheduling

8.1 The CPU scheduler

The CPU scheduler in the Linux kernel has been rewritten multiple times since its conception. This is because Linux has evolved during all these years and has gained support for many different kinds of systems and applications.

The original goals of Linux were to provide a Unix-like operating system to run on small personal computers. At that time, the applications were very simple and their requirements were not the same as the ones we have nowadays. One of these is quick response for interactive applications, something that has become very important due to Linux being used on desktop machines.

Since then, the kernel has evolved in a lot of areas and now runs on many kinds of different machines from high-end computer clusters to embedded devices, including, as already mentioned, desktop machines. These changes have forced the rewriting of the CPU scheduler multiple times during its existence to adapt to the new requirements.

The latest refactoring went in Linux 2.6.8.1, which introduced a new CPU scheduler with $O(1)$ algorithms (instead of the $O(n)$ one used before). This renewed scheduler can make decisions in a constant finite time, independently of the number of processes running on the machine. In other words, its response is generally deterministic and imposes a small penalty over system performance.
not. However, processes that have a parent/child relationship share the open file descriptions, and threads do too.

It is easy to imagine that this representation is quite heavyweight for threads, because each thread within a process replicates many information that conceptually describes the process as a container of threads. Even more, this description of threads "imposes" a 1 : 1 threading model, the one currently used in Linux.

### 8.1.2 Priority arrays

Each task within Linux has a priority attached to it; the lower the number, the more priority it has. Priorities are used to decide which is the next process to run when a context switch should happen.

A priority array has \( N \) lists of tasks where \( N \) is the number of possible priorities for any task. Tasks are added to the priority array in the list that corresponds to their priority. Furthermore, the priority array has a bitmap that indicates which of these lists have processes in them.

*Priority arrays* are an important part of the scheduler's implementation because they allow \( O(1) \) access to the next task to be scheduled. Checking to see if there is any task available in the priority array and which is its priority is a computation that has an upper-bound time limit because bitmaps are of fixed size. Getting that task from the priority array is also a constant-time operation because the process only consists on removing an item from the desired list.

### 8.1.3 Run queues

The *run queue* is the data structure that keeps track of all runnable tasks assigned to a CPU, hence each CPU has its own run queue. A run queue contains two priority arrays: one for tasks that can be executed in the current epoch and tasks whose quantum has already expired.

The idea behind having two priority arrays is also due to the \( O(1) \) scheduling goal. Tasks are always added to one of the arrays and moved into the other one when they have exhausted their quantum. Whenever the original list gets empty, the two lists are swapped so the expired list becomes the active one and vice-versa.
8.2. THE SPU SCHEDULER

8.2.1 The SPU context

The SPU context is the abstraction that describes a process to be run on a SPU. It is represented by the struct spu.context type and can be thought as the representation of a task running on a SPU, much like the struct task.struct type abstracts a process running on the PPU.

SPU contexts can be created regardless of the availability of SPUs to immediately run them. In fact, a SPU context can be seen as a virtual SPU: one can have as many instances as he wants, and the scheduler will later choose which one needs to be run.

When not running, SPU contexts contain the exact run-time status of a SPU, which includes the contents of all registers, a copy of the local store, etc. This allows resuming (or starting from scratch) their execution when a SPU becomes available.

8.2.2 Priorities and scheduling policies

The struct spu.context type has some fields that define its priority and its scheduling policy, which are later used by the SPU scheduler to take decisions. These are as follows:

unsigned long rt_priority An integer defining the real-time priority for this process.

int policy A value defining the scheduling policy for this process. This is currently unused.

int prio An integer defining the priority for this process.

The three fields described above are initialized when the SPU context is created, and inherit the values from the creator process’ struct task.struct (which is pointed to by current). They cannot be changed later on.

8.2.3 The scheduling algorithm

As mentioned earlier, the scheduling algorithm for SPU contexts follows a cooperative model in general cases, but does some preemption for real-time processes. The general behavior is depicted in figure 8.2.

Let’s start by explaining how non-real-time scheduling works. Whenever a program requests the execution of a SPU context, the scheduler attempts to find a free SPU to run it. If it is
Chapter 9

Scheduling ideas and changes

This chapter presents several novel ideas that could be implemented in the system scheduler to improve the management of resources in heterogeneous multiprocessor systems. We have also implemented a couple of these to be able to evaluate them on real hardware and draw some conclusions.

9.1 Application concept

Current operating systems lack the concept of an application. At the moment, all they do is define processes and the execution threads within them: while a single process is often all that makes up an application in an homogeneous multiprocessor system, this is not so common in heterogeneous multiprocessor systems.

What this means is that it is currently not possible to conceptually specify that several processes and/or threads belong to the same application. The programmer should be able to specify that all execution threads – no matter on which PE they are running on and which process they belong to – are part of the same application, and he should be able to specify run-time properties and relations among them to describe how they should be managed by the operating system’s scheduler.

In our case of study, this could be useful, for example, to avoid scheduling a PPE thread when all its SPU tasks are unable to run. Consider an application that launches several SPU tasks and that those communicate with the main PPE thread by writing a value in a specific
9.3. **UNIFY ALL THE SCHEDULERS**

A way to improve this situation could be to shift all these decisions to the system scheduler. This algorithm, when launching a task that can be potentially speeded-up by using an accelerator, should analyze the system load and decide whether it is better to use the specialized PE or to leave the task running on the general-purpose processor, aiming at finishing its job as soon as possible.

In order to get this to work, the application programmer should only provide different versions for his performance-critical code paths: one for each kind of accelerator in the machine and one for the general-purpose processor. He might also need to provide execution hints to the system so that it knew how much of a speed-up could the optimized version deliver versus the normal one. The scheduler could then choose among the available versions of the code and the status of the machine load.

As an specific example to our case study: suppose a graphical program that has some complex algorithms that deal with matrices. These algorithms could be provided in the form of Altivec routines and SPE programs, both of them having the exact same behavior when seen as a black box. The application could tell the scheduler — possibly by using some library or by letting the compiler insert the appropriate instructions, to keep everything transparent — that it has these two versions available. At the time of launching these algorithms, it could be the operating system who decided which version was the best one to run.

### 9.3 Unify all the schedulers

Another possible improvement, which has already been outlined multiple times along this report, is the unification of the different schedulers that are part of the system.

At the moment, the operating system includes separate schedulers for every kind of PE it is able to manage (see chapter 8). This means that each scheduler is not *aware* of the decisions taken by the other schedulers, so none of them can make intelligent decisions based on the requirements of each application.

If we materialized the application concept described earlier on, we could homogenize the different schedulers in the system and have just one that can see how the application is behaving and take better decisions to increase its throughput and the overall system performance.
9.5. STATIC PARTITIONING OF SPEs

9.5.1 Micro-benchmarks

To measure the changes made to the scheduler, we used a set of micro-benchmarks that characterize the different types of applications that can run on top of a Cell. We opted to use these instead of real programs because these make exclusive use of the SPEs and do not impose any load on the PPEs, which is interesting because we have not modified the scheduling of tasks on top of the latter. (Remember that there are two different schedulers in Linux: one for the SPEs and one for the PPEs.)

All these micro-benchmarks launch 12 concurrent SPU threads so, in the case of the PlayStation 3, they always overflow the available physical SPEs (which are 6). This way they force the scheduler to always intervene in the executions, even if there is just one instance of the program running at a given time.

Each line in the tables 9.1, 9.2 and 9.3 describes a test run on the machine in an isolated way: there were no other users no programs using the SPEs when the tests were run. The columns' meanings are:

**SPEs/task** The maximum number of SPEs each running task can use concurrently. This is, effectively, what implements the static partitioning idea we described.

**Tasks** The number of instances of the test launched in parallel to keep the SPEs' run queues full of tasks. The total number of running SPU threads at the beginning of the test (before the first instance finishes) is this number multiplied by 12.

**1st, 2nd, 3rd, 4th, 5th and 6th** The instants, in seconds, in which the parallel executions ended. These are measured as a relative time since the start of the test.

**Massive amount of events**

This micro-benchmark runs 12 tasks on the SPEs and each of them issues 10,000 `printf(3)` calls. The standard output of these tests is sent to the null device to avoid having the delays of a physical device such as the hard disk. As can be expected, the SPEs themselves do almost no execution on their own, given that they will be continuously stopping and requesting the PPEs to do the library calls. Even more, on each stop the task voluntarily yields its SPEs which
9.5. *STATIC PARTITIONING OF SPEs*

<table>
<thead>
<tr>
<th>SPEs/task</th>
<th>Tasks</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2.90</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5.78</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>17.31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4.36</td>
<td>5.80</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5.79</td>
<td>5.80</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>17.33</td>
<td>17.33</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5.79</td>
<td>5.79</td>
<td>8.67</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8.69</td>
<td>8.69</td>
<td>8.70</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>17.32</td>
<td>17.33</td>
<td>17.34</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>8.68</td>
<td>10.12</td>
<td>11.57</td>
<td>13.01</td>
<td>17.32</td>
<td>17.34</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5.81</td>
<td>7.23</td>
<td>14.43</td>
<td>17.33</td>
<td>18.76</td>
<td>20.18</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>17.32</td>
<td>17.34</td>
<td>17.35</td>
<td>17.34</td>
<td>17.36</td>
<td>17.37</td>
</tr>
</tbody>
</table>

Table 9.2: Results of the calculus-only benchmark.

Each SPE fetches this buffer into its LS by using a fixed data transfer block size. This transfer can be repeated multiple times to make the test take a long time without having to use lots of main memory, which is scarce in the PlayStation 3.

The parameters for this test were: 4 MB of total data for each SPE, a transfer size of 512 bytes and 100 repetitions for the whole transfer. The results of this micro-benchmark are shown in table 9.3.

<table>
<thead>
<tr>
<th>SPEs/task</th>
<th>Tasks</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>0.90</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1.36</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3.71</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1.22</td>
<td>1.56</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1.60</td>
<td>1.61</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3.93</td>
<td>3.94</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1.23</td>
<td>1.59</td>
<td>2.27</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1.98</td>
<td>1.98</td>
<td>2.81</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3.90</td>
<td>3.97</td>
<td>4.01</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>27.17</td>
<td>34.03</td>
<td>42.01</td>
<td>42.16</td>
<td>42.58</td>
<td>42.78</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>38.06</td>
<td>44.24</td>
<td>46.77</td>
<td>47.64</td>
<td>48.86</td>
<td>48.86</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>36.71</td>
<td>37.37</td>
<td>42.28</td>
<td>45.15</td>
<td>45.67</td>
<td>46.64</td>
</tr>
</tbody>
</table>

Table 9.3: Results of the DMA transfers benchmark.
9.6 Dynamic partitioning of SPEs

An obvious extension to the static partitioning policy described in section 9.5 is to make the splitting of resources dynamic. The idea would be to let an application use as much SPEs as it wanted, as long as there were no other applications requesting SPEs. Once another application was executed needing to use this resource, the other running programs could be forced to yield some of the SPEs they were using, leaving them to the new applications.

This splitting could be done halving the amount of allocated SPE each time a new application was launched. This way, the program could start using 6 SPEs (in the case of the PlayStation 3), then be reduced to 3 SPEs when introducing a new application, then to 2 with another program and at last to 1 with yet another task.

We have not been able to test this policy due to lack of time, given that implementing it is fairly complex due to locking issues among PPE threads and SPE threads.

9.7 Accounting of SPU execution time

If we discard the idea that foreign PEs are I/O devices and we consider them as real processors, we should take into account the time they spend executing code and the load they impose on the system. Unfortunately, current systems do not do so because this makes no sense for I/O activity.
results. We used only one SPE for simplicity, and we used a 4096 matrix size. These executions were done before and after our changes. Table 9.4 shows the timing results without SPU time accounting; table 9.5 shows the same executions after SPU time accounting was implemented.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Total time</th>
<th>User time</th>
<th>System time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>8.952s</td>
<td>8.522s</td>
<td>0.432s</td>
</tr>
<tr>
<td>2nd</td>
<td>9.238s</td>
<td>8.803s</td>
<td>0.717s</td>
</tr>
<tr>
<td>3rd</td>
<td>8.956s</td>
<td>8.522s</td>
<td>0.432s</td>
</tr>
<tr>
<td>4th</td>
<td>8.960s</td>
<td>8.523s</td>
<td>0.435s</td>
</tr>
<tr>
<td>5th</td>
<td>8.949s</td>
<td>8.520s</td>
<td>0.428s</td>
</tr>
<tr>
<td>6th</td>
<td>8.953s</td>
<td>8.521s</td>
<td>0.430s</td>
</tr>
<tr>
<td>7th</td>
<td>8.976s</td>
<td>8.531s</td>
<td>0.444s</td>
</tr>
<tr>
<td>8th</td>
<td>8.965s</td>
<td>8.532s</td>
<td>0.428s</td>
</tr>
<tr>
<td>9th</td>
<td>8.957s</td>
<td>8.523s</td>
<td>0.432s</td>
</tr>
<tr>
<td>10th</td>
<td>8.953s</td>
<td>8.524s</td>
<td>0.426s</td>
</tr>
</tbody>
</table>

| Average   | 8.986s     | 8.552s    | 0.460s      |

Table 9.5: Example of time accounting considering SPUs.
Part IV

SPE management
Chapter 10

Interaction with the SPUs

As we have already seen in multiple places, the SPEs in a Cell-based machine are currently managed by the Linux kernel and are exposed to the user as I/O devices. The way this is done is through a low-level interface presented in the form of a virtual file system that allows direct control of the SPEs. On top of this, the Cell SDK provides a library that acts as a higher-level interface to these units.

This chapter details how this virtual file system behaves and how the user-space library interacts with it, as well as the services it provides to the programmer.

10.1 The SPU file system

The SPU File System (spufs) [Ber05] is the interface exposed by the Linux kernel to manage SPU contexts from user space. It provides a virtual file system that abstracts access to the resources that describe the execution of a task, which are presented as regular files. This interface is very low-level and should not be used by end-user programs.

10.1.1 Context representation

SPU contexts—described in section 8.2.1—are exposed to the user through the spufs. Each one has the form of a directory, and each directory contains a known set of files that describe the context’s properties and status.
10.2. LIBSPE 2.X INTERNALS

10.1.2 System calls

Aside from the file system-level interface, the kernel provides two system calls to create and execute the contexts stored in the spufs. These are:

**Context creation (spu_create)** Given a context name, its creation flags, its mode and a SPE affinity list, creates a new context inside the spufs. Its signature is:

\[ \text{spu.create} : (\text{const char *}, \text{unsigned int}, \text{mode_t}, \text{int}) \rightarrow \text{long} \]

This *does not imply* that the context is ready to be executed on a SPE, nor that it is assigned to one.

**Context execution (spu_run)** Given a file descriptor for a context, the program counter and a place to store the execution’s return code, puts the context in a physical SPE and executes it. This function *blocks* until the SPE has stopped. Its signature is:

\[ \text{spu.run} : (\text{int}, \text{u32}, \text{u32 *}) \rightarrow \text{long} \]

One could argue if such a design is optimal. While the spufs provides a kernel-agnostic way to access SPEs, the need to use these two system calls breaks this abstraction. Hence, everything could have been done with a kernel-based API. However, it is worth to note that preliminary versions of the file system did not require the use of the system calls and instead relied on the traditional *mkdir(2)* and *ioctl(2)* operations; this can be seen in the descriptions given in [Ber05], which talks about older versions of the spufs.

10.1.3 Example

Figure 10.1 shows a very simple example of how to create a context using the spufs interface, how to load a program on the context’s local store and how to execute it.

10.2 libspe 2.x internals

The spufs is a too low-level interface for SPE interaction in generic applications. The libspe is a library that provides a higher-level interface on top of the spufs.
4. `sys_spucreate()` delegates on `spucreate()`, which:

- Only registers entries in the virtual file system, hence we can create as much SPU contexts as we want.

- Finalizes the creation by delegating on `spucreate_gang()` if the context is located inside a gang, or `spucreate_context()` if it is created independently.

5. Maps the context’s local store into memory and stores the address in the `mem.mmap.base` variable.

6. If the user gave the `SPE_MAP_PS` flag, maps the context’s problem store into memory.

10.2.2 Image loading

SPU programs can be stored on disk as independent ELF images. These can be opened at runtime by using the `spe_image_open()` function, which has the following signature:

```
spe_image_open : (const char *) → spe_program_handle_t *
```

Once the image has been loaded, this function returns a pointer to a `program handler` to be used later to load the program onto a SPE.

This operation:

1. Opens the specified file.

2. Ensures that the file is executable.

3. Ensures that the file is a valid ELF image and that it has the correct architecture type.

4. Maps the ELF image into memory for later usage.

10.2.3 Program loading

SPU programs are loaded into a SPU context through the `spe_program_load()` function, whose prototype is:

```
spe_program_load : (spe_context_ptr_t, spe_program_handle_t *) → int
```
(a) Asks the scheduler to put the given context into executable state and waits until it has stopped execution due to any reason.

(b) Checks why the SPE stopped executing the context. If it was due to any condition that can be resolved into the kernel – such as a PPE-assisted system call, a memory fault, etc. – the event is resolved and execution is immediately resumed without going to user space.

(c) If the execution stopped due to a stop and signal condition – a library call, for example –, the system call returns and tells the user about this condition.

(d) When returning to the user, the current SPE’s program counter is returned as well as the stop code.

5. Checks the return code for the spu_run() system call:

- If it refers to a standard library call and the SPU_NO_CALLBACKS flag was not given, issues the library call and later resumes execution by calling spu_run() again with an updated program counter.
- If the code is unknown, it is returned to the caller for further processing, who must explicitly continue execution once he has finished dealing with the event.

### 10.2.5 Signal handling

The stop and signal operation is a very common one in SPU programs. This is used to voluntarily stop the execution of the process and send a user-defined code to the PPE. This code is used to signal specific conditions.

When that happens, the PPE captures the signal code, interprets it, handles the event as appropriate and resumes the execution. This implements a coroutines programming model between a SPE and a PPE.

The standard signal codes are illustrated in table 10.1.
Chapter 11

Addition of asynchronous SPE execution

Following the current Linux programming model, programs for a SPE cannot be executed stand-alone. There must be a program in the PPE which takes care of the task being executed on the SPE; the former is officially called SPU thread and the latter has no specific name, but we will refer to it as SPU task for simplicity. This structure is depicted in figure 11.1, where we have the main program’s thread and four SPU threads managing four SPU tasks. The main program’s thread can continue executing after it has spawned the SPU threads, but these are blocked waiting for the SPU tasks to complete.

The SPU thread is responsible for starting the execution of the SPU task and is in charge of handling any events reported by it, which can be of these types:

**Execution errors** These events refer to problems that appeared when executing code in a SPE. The most common example of one of these events is the attempt to execute an invalid instruction.

**Memory management errors** These events are raised whenever the SPE executes an invalid DMA transaction to access main memory. Some of these problems cannot be resolved and appear because of incorrect programming of the SPE, such as data misalignment or an
call is served in a synchronous way, and after its processing, the SPE resumes execution until it requests the task's termination.

Figure 11.2: Synchronous cooperation of a PPE and a SPE; a coroutine model.

Given that \texttt{spu\_run(2)} always blocks, there must be a SPU thread for each SPU task (or otherwise a program could only run a single SPU task at a time, because it could be blocked on the first execution attempt). This is very similar to a 1:1 threading model, in which each user-level thread has a kernel-level thread. Even though the relationship between SPU threads and SPU tasks has nothing to do with user-level and kernel-level threads, we will use the same terminology here due to its similarity. That is, a \(i:j\) specification, for any value of \(i\) and \(j\), will refer to \(i\) SPU threads managing \(j\) SPU tasks.

It is important to note at this point that SPU threads are generally sleeping (blocked in the \texttt{spu\_run(2)} call). The key idea when developing a program for a SPE is to minimize the interactions with the PPE, allowing it to run in a completely standalone way. For example: the ability to issue system calls from SPU tasks is provided to ease the porting of applications to it, but this feature should not be used on programs supposed to deliver high performance. Even though, some events are hard to avoid such as those caused by page faults.
• If the polling mode is selected, the kernel exports a new event file in the spufs' description of each SPU context. This file is later used by the SPU thread's application to block and wait for events. As it is a file, the program can use the traditional select(2) or poll(2) system calls to wait for events from multiple SPU contexts (tasks).

Once the user program has received an event notification, it has to receive more information about it, which includes the reason for the event as well as any information attached to it. This is possible by using the new spu.process.event(2) system call. This call also provides a way to resume the execution of the SPU task once the event has been successfully processed in user space.
Signals This mode completely decouples the execution of SPU tasks from the manipulation of their events. The SPU thread only requests the execution of the SPU tasks, and it will later receive a signal (handled in another thread) when it is time to manage events.

Figure 11.5 shows how this mode works. It illustrates the program's main thread, which is the only thread running in the PPE for the application. This thread launches the execution of four SPU tasks using the signals mode, and when any of these raise an event, the main thread is notified by means of a signal. The way the signal is handled is up to the program. E.g. it could have an auxiliary thread specifically meant to handle them, but it could do so from the main's program loop if it wanted to.

To enable this behavior, define `CBE_BACKUP_THREADS` to the number of SPU threads that ought to control the SPEs and set `CBE_ASYNC_EXEC_MODE` to `signals`. 
Asynchronous execution (spe_run_async) Starts the execution of a new SPU task and configures it to deliver SPU events in the new N : M model. This call returns immediately. Its prototype is:

\[
\text{spe.run.async : spe.gid.t, spe.program.handle.t, void *, void *, unsigned long, int, enum spe.evnotif.mode \rightarrow \perp}
\]

This new call behaves exactly as the original spe.create.thread() function but takes an extra parameter: the enum spe.evnotif.mode mode. This parameter indicates the kind of event notification we want for the new thread, and can be one of SPE_EVNOTIF_DEFAULT, SPE_EVNOTIF_POLLING or SPE_EVNOTIF_SIGNALS.

11.4 Usage

Using the new libspe 1.x library is trivial, given that it is source and binary compatible with the original versions of it. Our work is based on libspe 1.2.2, so further versions of the 1.x line may not be compatible.

Of course, programs cease to be compatible with the original library if they use the new APIs, but compatibility is a strong point for evaluation given that it allows us to run existing programs in the new model without changes. There is no need to even rebuild the programs, since we can use the LD_PRELOAD feature to force the usage of our own library; just do the following:

\[
\text{
$ \text{LD_PRELOAD=/path/to/our/libspe.so.1.2.2}$
}\]

\[
\text{
$ \text{export LD_PRELOAD}$
}\]

From now on, all programs that use the libspe 1.x will have the new N : M threading model available to them. The rest of this section provides some examples of its usage.

Let's assume we have a utility called test that spawns multiple SPU tasks and waits for their finalization. Each SPU task then does some I/O and calculations, raising a bunch of events. This application is written using the original APIs, so if we run it as shown below, it will use the 1 : 1 threading model:
11.5. EVALUATION

11.5.1 Micro-benchmarks

To see how our solution behaves in real hardware, we prepared several benchmarking utilities that represent different situations that can arise when running programs designed for the Cell processor.

The micro-benchmarks were executed on the PlayStation 3 when there were no other users working with it. This way the results are not biased due to the interactions with other applications nor users – i.e. all the 6 SPEs were available for use by our tests.

Each micro-benchmark was executed with four different setups:

1. Synchronous (6 threads): This test was done with the original kernel and library. Due to the 1 : 1 relation imposed by these components, this test used 6 SPU threads – one for each possible SPU.

2. Asynchronous (polling, 1 thread): This test used the modified kernel and library, with the polling mode, and one backup thread. The idea was to see how well a single thread scales to manage the events generated by multiple SPEs.

3. Asynchronous (polling, 2 threads): This test used the modified kernel and library, with the polling mode, and one backup thread. The idea was to see if two threads – one for each PPE in the machine – were able to better split the load of the machine.

4. Asynchronous (polling, 6 threads): This test used the modified kernel and library, with the polling mode, and one backup thread. The idea was to simulate the behavior of the original library by having one backup thread for each SPE.

We only used the polling mode because it proved to be more stable than the signal one. The implementation should be fixed to work perfectly in both cases, but we did not have time to do so.

To get meaningful results, we ran each of the above configurations 10 times and later averaged the results. All of them were done with hot caches.
Summarizing: our implementation behaves worse than the original one if a Cell program executes lots of library calls from the SPE. Fortunately, these applications do not exist in the real world because the key idea behind the SPEs is to act as accelerators: doing library calls goes against this idea due to the way they are handled.

Calculus-only tasks

This micro-benchmark simulated a real Cell application. It spawns a calculus-only task on all available SPEs and later waits until they complete the execution. To make things simple, this task simply runs a very long loop so that no DMA nor events are involved.

Table 11.2 shows the results of this experiment.

<table>
<thead>
<tr>
<th>Total time</th>
<th>User time</th>
<th>System time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous (6 threads)</td>
<td>28.82s</td>
<td>0,000s</td>
</tr>
<tr>
<td>Asynchronous (polling, 1 thread)</td>
<td>28.83s</td>
<td>0,000s</td>
</tr>
<tr>
<td>Asynchronous (polling, 2 threads)</td>
<td>28.83s</td>
<td>0,000s</td>
</tr>
<tr>
<td>Asynchronous (polling, 6 threads)</td>
<td>28.83s</td>
<td>0,000s</td>
</tr>
</tbody>
</table>

Table 11.2: Calculus-only task before and after our changes.

As can be seen in the table, the timing difference between the original implementation and the new one (which amounts to 0.01 seconds) is constant but not significant at all. This extra time is probably due to the overhead in setting up the backup threads and to our specific kernel and library implementation. It should be possible to make the times exactly equal by tuning the implementation.

11.5.2 Real application: Matrix multiplier

Aside from the micro-benchmarks presented in the previous section, we used a real-world application to see how it behaved after we implemented our changes. We had some stability problems with applications that issued massive amounts of DMA transfers, so we finally had to settle with a simple sample program included in the Cell SDK: the matrix multiplier\(^3\).

\(^3\)This utility can be found in /opt/ibm/cell-sdk/prototype/src/workloads/matrix.mul in the 2.1 version of the Cell SDK.
Part V

Miscellaneous work
Chapter 12

Bugs fixed

During the development of the project, I found several bugs in the existing code. As one of my personal goals for the project was to get involved into Linux and Cell development, I opted to fix and submit patches to the mainstream developers so that future versions of their software did not contain those errors.

Aside that, some of the bugs prevented me from doing further work, so I was forced to fix them. This was because they made the code fail or not work as expected, yet I needed that specific functionality to behave correctly.

This chapter describes the problems I found and how they were fixed.

12.1 Linux’s RPM package creation

The Linux’s build system provides a `rpm-pkg` target that builds the kernel and later generates a RPM package. This package can be used to painlessly install the customized kernel on a RPM-based system, such as is the case for the distribution we were using (Fedora Core 6). I wanted to use this target to create my own RPM packages and later install them on the PlayStation 3 on a daily basis.

Unfortunately, this target did not work with Linux sources fetched from Git repositories. The reason was because the RPM build file was created from the Git source tree, hence deducing a specific version number for the Linux kernel, but when the code was actually built, the Git control files were removed, thus producing a different version number.
Furthermore, there was a problem in the `DEBUG_PRINTF` macro that prevented it to be used in single-line control sequences that did not use braces; doing so could raise a build error. For example:

```c
if (condition)
    DEBUG_PRINTF("Condition X was met\n");
else {
    ... do something else ...
}
```

These problems were reported to the `cbe-oss-dev` mailing list and they got the 10111 tracking number\(^2\).

### 12.3 Invalid syscall names in the libspe 1.x

The libspe 1.x provides a header file, `include/sys/spe.h`, that checks whether the SPU-related syscall symbolic names have been defined in the standard system headers (`unistd.h`) and, if not, provides sane replacements. This situation can arise when building the libspe 1.x against a recent kernel but with obsolete header files.

The problem is that the check to see if the macros were defined and the their corresponding fallback names were incorrectly specified. The kernel uses the `_NR_spu_run` and `_NR_spu_create` names but the library was expecting `_NR_spe_run` and `_NR_spe_create` instead.

This problem was reported to the `cbe-oss-dev` mailing list and it got the 10152 tracking number\(^3\).

### 12.4 Lack of the loop device in the PlayStation 3

The Linux kernel sources provide a sample configuration file for the PlayStation 3 machine, appropriately named `ps3_defconfig`. However, this file failed to enable the loop device, pre-


\(^3\) [http://patchwork.ozlabs.org/cbe-oss-dev/patch?id=10152](http://patchwork.ozlabs.org/cbe-oss-dev/patch?id=10152)
variable name in the code. Hence, after doing a make clean from the project’s top directory, a machine-specific file remained in it.

This problem was reported to the cbe-oss-dev mailing list and it got the 10408 tracking number\footnote{http://patchwork.ozlabs.org/cbe-oss-dev/patch?id=10408}. 
Chapter 13

Repository for Cell applications

Testing changes made to the Linux kernel and/or the libspe on a Cell-based machine is not easy. This is because, at the time of this writing, there were few real-world applications that made use of the SPU. Strictly speaking, there are some programs available, but the problem is that they are still experiments, are in early stages of development or are simply private and not redistributable.

To make things worse, building these applications in machines that differ from the developers’ ones is a nightmare: they rely on old and incompatible versions of the Cell SDK and they have non-portable paths hard-coded in the build scripts. The only solution to the former problem is to migrate the programs to newer versions of the platform, but the latter could be better handled by the mainstream developers if they used tools such as GNU Autoconf [Fou06a] and GNU Automake[Fou06b].

One of my tasks, not originally planned, was to create a repository to hold a collection of applications for the Cell, ready to be used by other researchers. This involved porting the applications other researchers gave us to our development platform — the Cell SDK 2.1 running on a Fedora Core 6 on top of the PlayStation 3—, writing some easy to use build scripts and providing some minor documentation on how to build and execute each utility.

This chapter describes how I configured the repository and the applications we stored in it. Some of them are private (we cannot redistribute them) so access to the repository is restricted.
13.3. THE PROJECTS

Figure 13.1: Gitweb's main screen.

- **apps/djimenez/3ddock.git**: The FTDock application, ported to the Cell, ready to be used on the PlayStation 3. This was provided by Daniel Jiménez and is private.

- **apps/dsn/pbpi.git**: The PBPI application, ported to the Cell, ready to be used on the PlayStation 3. This was provided by Dimitris Nikolopoulos and is private.

- **apps/dsn/raxml.git**: The RaXML application, ported to the Cell, ready to be used on the PlayStation 3. This was provided by Dimitris Nikolopoulos and is private.

- **jmerino/benchmarks.git**: The micro-benchmarks used to evaluate our changes to the system.
Part VI

Denouement