Northumbria University

School of CEIS

Masters Programme in Computing

Individual Project

Student Dissertation

2007/2008

NAME: MOREY-CHAISEMARTIN Nicolas

I certify that the work contained in this report is the sole work of the author except where indicated. All material that has been taken from other sources has been clearly acknowledged. Quotations from other sources have been clearly marked, using quotation marks or a block quote.

Nicolas Morey-Chaisemartin

Signature numérique de Nicolas Morey-Chaisemartin
DN : cn=Nicolas Morey-Chaisemartin, email=nicolas@morey-chaisemartin.com, c=FR
Date : 2008.08.26 20:59:28 +01'00'

Signature ______________________________ **Date** ________________

Dissertation title:

A Hard Real-Time Kernel for a Heterogeneous Multicore Architecture: The Cell Broadband Engine

Acknowledgements

I would like to thank my supervisor, Adrian P. Robson, for helping me through my project and refocusing me when needed.

I also would like to thank, the people from IBM, and Ruben Niederhagen from Aachen University for their advices and numerous ideas.

Last but not least, I would like to thank my family for their support and their help brining the final touch to my dissertation.

Abstract

In this dissertation, a design of a hard real-time kernel for the Cell Broadband Engine is proposed. This design focuses on the problems due to the heterogeneous multicore architecture on the Cell. Task allocation, scheduling and synchronization algorithms are discussed to maximize the kernel performances. A particular attention has been ported to the semaphores as synchronization is a complex problem in a distributed environment.

A proof of concept, Virt-K, has been implemented over Linux to demonstrate the efficiency of the design. Timing analysis and performances measurements have been realized using Virt-K to localize the weak points of the design. Based on the performances, the design flaws are discussed, solutions are proposed and enhancements suggested.

It is concluded by proving that a soft real-time kernel can be run on the Cell Broadband Engine but the results are not conclusive for hard real-time kernels due to unbounded semaphore acquisition time. Suggestions on further research axes are proposed.

Contents

Table of Figures

1. Introduction

In the late years, microprocessor founders have had trouble following the Moore Law. Not only they have had to face physical problems due to the size of transistors but also what is called the “Memory Wall” [88] due to the memory latency and cache misses. Numerous new ideas have been tried to solve this problem but only few were successful. Classical multicore architectures, like Intel Core Duo, have managed to achieve greater performances but memory accesses are being more critical than ever.

However as described in [35], STI, the union of Sony, Toshiba and IBM, has worked since 2000 on this problems and has found a working solution: the Cell Broadband Engine, commonly called Cell. The results of the Cell will not be discussed here but they can be found in [86] .

The Cell thus has many interesting aspects. It has been made as a replacement for personal computers architectures (x86, x86_64) but is also a powerful parallel calculator. It is being currently used in the Playstation 3, but also in multiprocessor computers and soon in clusters.

Parallel computing has been a research subject for many years now, but its application to real-time systems has always been problematic as many processors often imply large power consumption. The Cell may change this as IBM clearly indicates in its official articles, [35] among others, “The Cell processor should provide extensive real-time support”. Further references to real-time support can also be found in the technical documentation of the Cell (i.e. [29]). STI has also announced, according to [80], a lighter version of the Cell for embedded hardware and real-time usage

The few research done on real-time for the Cell have been on efficient computing algorithm as MPEG compression, ray tracing, terrain rendering, but none of them has clearly answered the question of the feasibility of a real-time operating system for the Cell.

1.1 Hypothesis

The hypothesis of this project is:
"*Can an efficient hard real-time kernel be implemented on the Cell?*"

1.2 Aims

The aim of this project is to prove the hypothesis. A hard real-time kernel will be designed and then implement as a virtual kernel for the Cell, running as a Linux program.

The two criteria to prove the hypothesis are the feasibility of a real-time kernel, and its efficiency:

- The feasibility can be seen through the possibility of running processes, synchronizing and scheduling them successfully.
- The efficiency criterion is mostly the predictability of the kernel. All the common kernel tasks should have predictable runtimes.

1.3 Objectives

- Research and discuss Linux implementation on SPEs[1].
- Specify and develop a virtual kernel using Linux as a non real-time hypervisor.
- Research, design and implement a scheduling algorithm to manage the SPEs
- Research, design and implement semaphores for the SPEs
- Test system performances and compare to existing real-time kernels.
- Research and discuss possible algorithms to achieve higher performances and more generic purpose.

[1] SPE stands for Synergistic Processor Element. See Chapter 2.1.3 for more details.

1.4 Dissertation organization

In Chapter 2, the subject is introduced with an overview of the Cell Broadband Engine and of real-time kernels to define the environment of the project and ease the understanding of the work done.

In Chapter 3, a design of a hard real-time kernel for Cell Broadband Engine is proposed. The design focuses on high performance scheduling and synchronization.

In Chapter 4, the implementation of the proof of concept is discussed. An overview of the software architecture and its specificity are given.

In Chapter 5, a timing analysis of software is provided and confirmed by a practical performance tests.

In Chapter 6, the weak points of the design are studied. Solutions and enhancements are suggested.

In Chapter 7, conclusions from the previous analysis are given and new research axes are proposed.

2. Cell Broadband Engine and Real-Time Kernel

To fully understand the implication of this project, a small background on the Cell processor and real-time kernel is needed.

2.1 The Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (commonly called CBEA) has been standardized by IBM in 2005. Its first implementation is the Cell Broadband Engine (called CBE) which is the support platform of this project.

The development of the CBEA itself started in 2000 by the union of Sony, Toshiba and IBM in order to solve the common problems of current architectures (x86, Itanium). The CBE was developed at the same time as an example of the CBEA to be implemented in the Playstation 3. The following description will be focused on the CBE as it is currently the only implementation of the CBEA. The main difference, concerning this project, is that the CBEA does not specify a number of processing elements. Processor with many processing elements may appear for high-computational needs, but also smaller ones for embedded applications.

2.1.1 The Cell Broadband Engine

The Cell, as described by [32], consists of 9 cores on a single chip. One of them is the PPE (PowerPC Processor Element), and the other eight are advanced computational units called SPE (Synergistic Processor Units). All these processors are linked by a high data-rate bus called EIB (Element Interconnect Bus). The SPEs and the EIB are the keys to the Cell success as they provide a considerable amount of computational power without neglecting the memory wall problem.

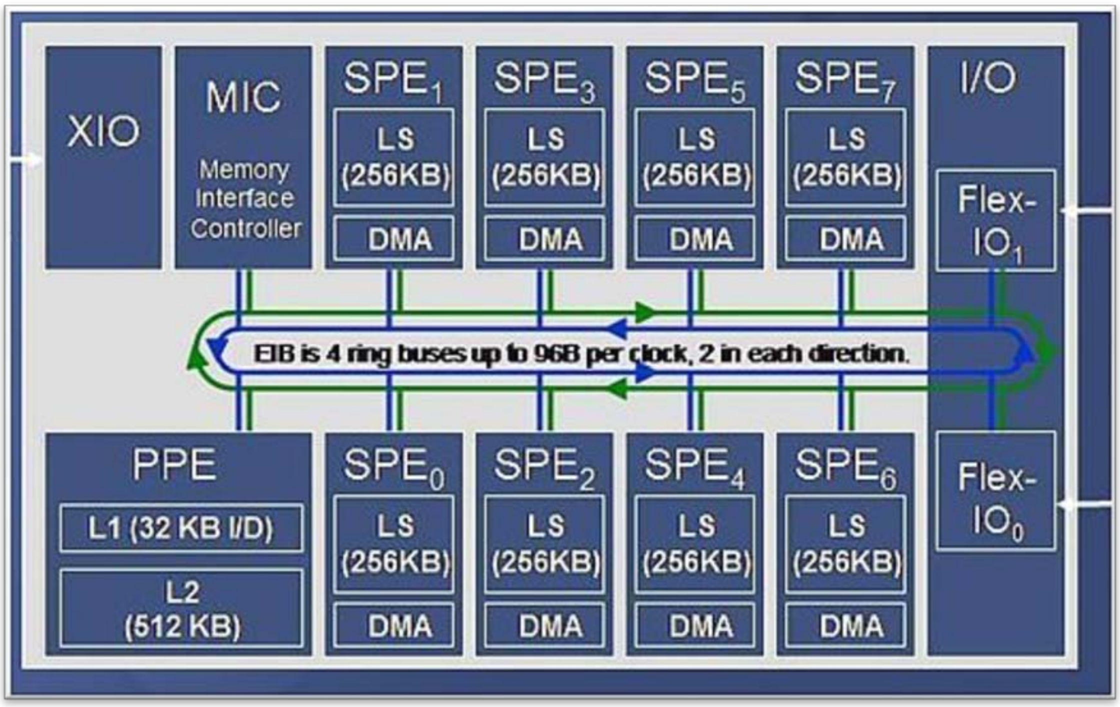


Figure 2-1 - The Cell Broadband Engine

2.1.2 PowerPC Processor Element (PPE)

According to [29], the PPE being the central element of the Cell is in fact an enhanced PowerPC 64bits core. Thus, using well known technology, STI has guaranteed some compatibility with existing softwares.

In the current Linux version running on the Cell, the PPE is the only core used by default. However softwares can choose to send tasks to the SPEs. The problem with this solution is that, although all the previous PowerPC software can be executed without any changes, a lot of the SPE computation time is wasted as few software are using them.

The main role of the PPE, fixed by the design [32], is to host the operating systems as it has full access to all hardware.

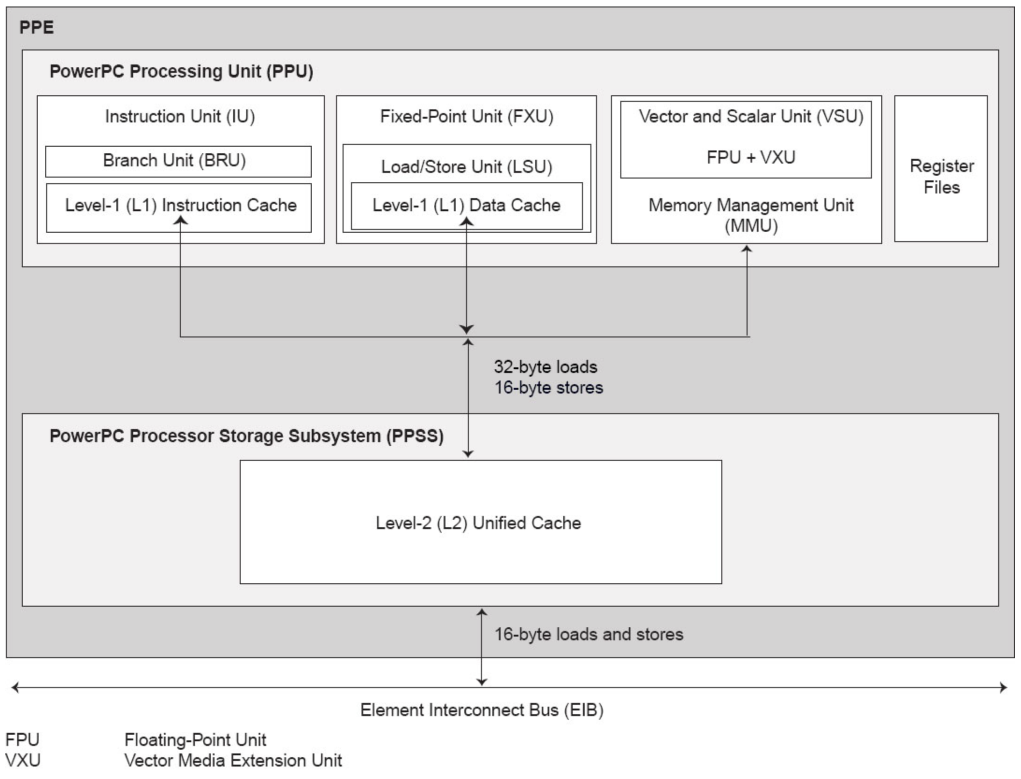

PPE
PowerPC Processing Unit (PPU)
Instruction Unit (IU)
Branch Unit (BRU)
Level-1 (L1) Instruction Cache
Fixed-Point Unit (FXU)
Load/Store Unit (LSU)
Level-1 (L1) Data Cache
Vector and Scalar Unit (VSU)
FPU + VXU
Memory Management Unit (MMU)
Register Files
32-byte loads
16-byte stores
PowerPC Processor Storage Subsystem (PPSS)
Level-2 (L2) Unified Cache
16-byte loads and stores
Element Interconnect Bus (EIB)
FPU Floating-Point Unit
VXU Vector Media Extension Unit

Figure 2-2 - Power Processing Element (PPE)

2.1.3 Synergistic Processor Elements (SPE)

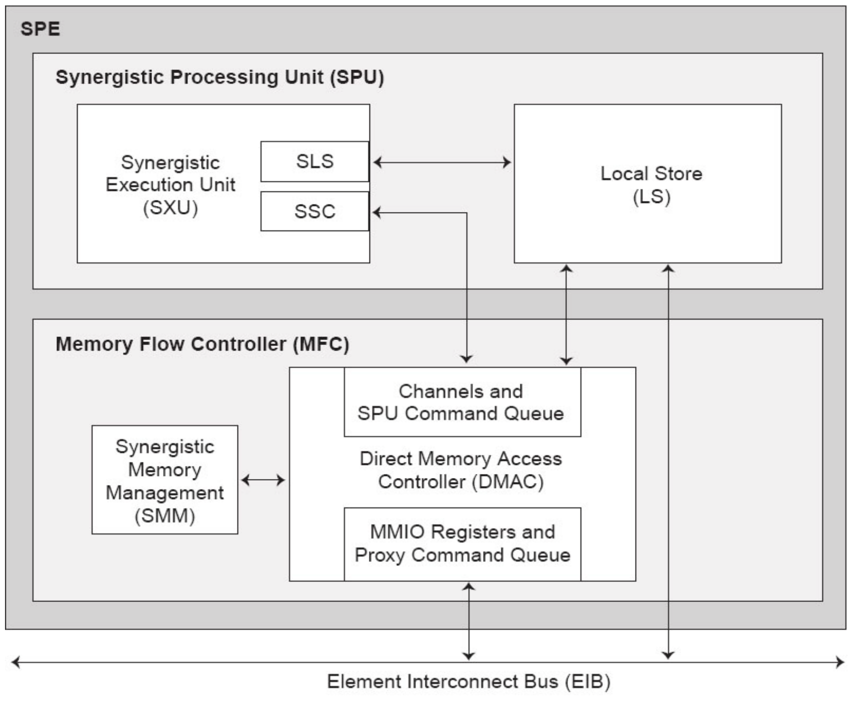
The SPE is a highly advanced 64 bits computational unit using SIMD (Single Instruction Multiple Data).

The main difference between PPE and SPE, more than the restricted instruction set of the SPE, is the way they access memory [27]. Where the PPE uses the same standard access (cache L1 and L2) as other processors, the SPE has its own local memory. As [29] indicates, each SPE owns a 256 kilobytes Local Store.

Another critical difference is that a PPE accesses the memory “on the fly” which means the data are retrieved from memory to the cache at the moment they are needed. However, the SPE owns a MFC (Memory Flow Controller) which can retrieve data asynchronously through DMA transfers. The MFC is given a list of memory area to load or store and will do it as soon as the hardware makes it possible. Therefore, the SPE can pre-fetch the data it will need before it actually needs it. This may seems really simple, but it is a solution to the memory wall as mentioned earlier. The memory latency is not a real problem anymore as the SPE does not have to wait for its data to keep running. To be more precise, data being prefetch, the program will not encounter cache misses, and then run faster. In the case of a single process running on a SPE (isolated and non-pre-empted), it is possible to calculate exactly response time within the SPE. As every instruction has a know execution time and so does the Local Store access, the execution time of a function can be calculated at a SPE cycle precision.

Moreover, it is fully possible to imagine a pre-allocation of data in the Local Store having a task ready to run as soon as a signal (intern or external event) arrives, avoiding a costly context switch. By sharing the Local Store, several processes could be stored at the same time. Context switch in this case would only require saving the different register.

For a soft real-time usage, the same principle could be applied by adding paging functions to send back some processes to the main memory if needed. Context switch would still be less costly than copying the full Local Store to the main memory and the average response time would stay quite low.


SPE
Synergistic Processing Unit (SPU)
Synergistic Execution Unit (SXU)
SLS
SSC
Local Store (LS)
Memory Flow Controller (MFC)
Channels and SPU Command Queue
Synergistic Memory Management (SMM)
Direct Memory Access Controller (DMAC)
MMIO Registers and Proxy Command Queue
Element Interconnect Bus (EIB)

MMIO	Memory-Mapped I/O Registers
SLS	SPU Load and Store Unit
SSC	SPU Channel and DMA Unit

Figure 2-3 - Synergistic Processor Element (SPE)

More than the Local Store, each SPE is provided an Atomic Cache Unit (ACU) to manage concurrent access to the main memory. The ACU acts like cache memory. If a memory area is in the ACU of a SPE, and if another SPE tries to access it, it will retrieve the value stored in the other SPE ACU. Concurrent modifications can be detected by hardware as a “dirty” bit is set when trying to store back in memory a value which has been modified by another SPE.

The SPE are also provided with three 32bits mailboxes. Two of them are outbound, one for message, the other for interruption and can send message to the PPE, other SPE or even peripherals. The third one is a four entry inbound mailbox to receive messages from these other elements.

2.1.4 Element Interconnect Bus (EIB)

To connect all the elements of the CBE, an advanced high data-rate bus has been designed. This bus, called Element Interconnect Bus or EIB, is in fact composed of 4 16-bytes wide circular unidirectional channels which counter-rotate in pairs.

These channels can execute up to three memory transactions at once if the configuration allows it. These channels work in a similar way to a token-ring network. The EIB is optimized to transfer large amount of data as each transaction transfers 128bytes (8 transfers on 16bytes wide). The concurrent accesses to the bus are managed by an arbiter. In the current CBE implementation, the first SPE has the priority on the second, the second on the third, etc.

In theory the EIB can achieve a peak data bandwidth of 204.8GB/s. A peak bandwidth of 197GB/s has been achieved by IBM [14].

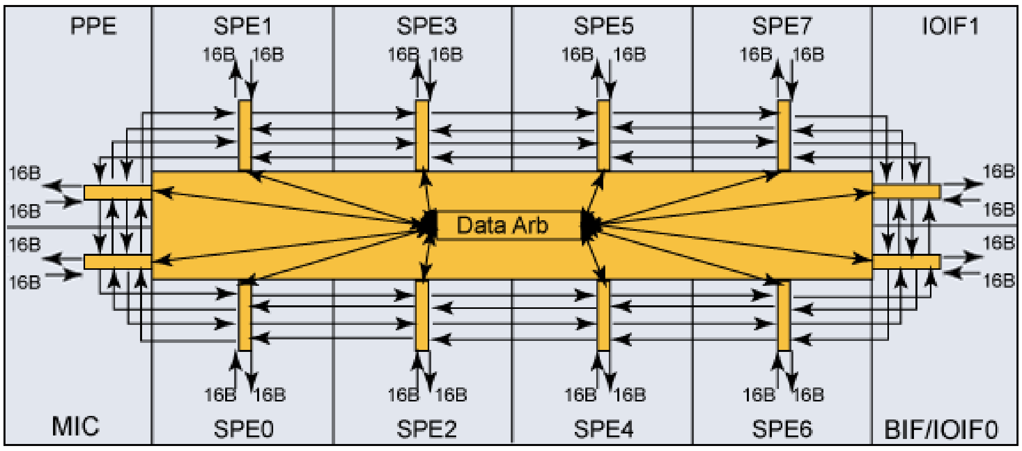


Figure 2-4 - Element Interconnect Bus Topology

2.1.5 Other Elements

For its external communications, the CBE uses two Flex IO interfaces and a XDR memory controller.

The FlexIO interface is a 48bits bus running at double clock speed. Their theoretical bandwidth is 76.8GB/s, about ten times faster than an AMD64. This bus is dynamically configurable and eases the use of the Cell in different environments. Moreover, one of the two FlexIO interface can be used to interconnect CBE. Theses connexions can either be done directly between two CBE or using a BIF (CBE Interface Protocol) switch.

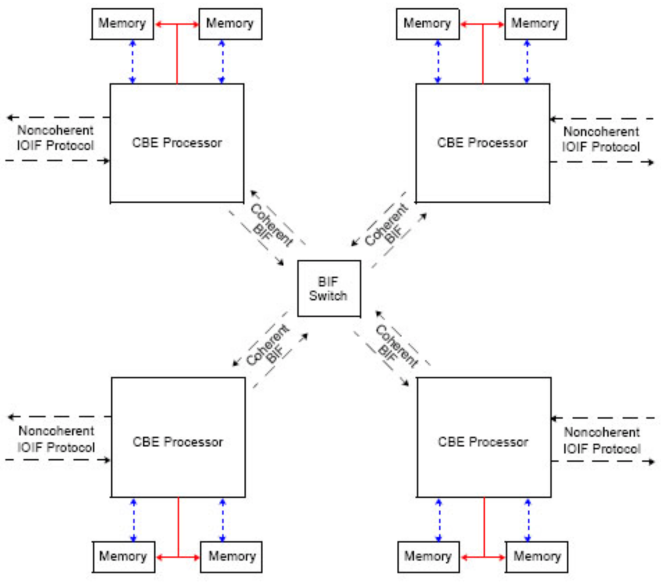


Figure 2-5 - Four CBE System

The CBE was designed to use XDR Ram as memory, although the latest CBE version, used in the Roadrunner cluster, is compatible with DDR. The advantages and drawbacks of both memory will not be discussed here.

2.1.6 Further details

The CBE and its new architecture, first produced in 2005, is still a subject of research.

More than looking for efficient algorithm using its full potential as [37],[58] and [6] , many laboratories are working on possible implementations and how to enhance the current Linux for CBE.

[39] proposes an implementation of MPI (Message Passing Interface) on the CBE. MPI is necessary to work efficiently on multiprocessor Cell computers. They have implement blocking point-to-point communications on the current Linux for that. But more than that, the techniques they used are interesting for SPE synchronisation (mutexes semaphores). It will be discussed in more details later in this review.

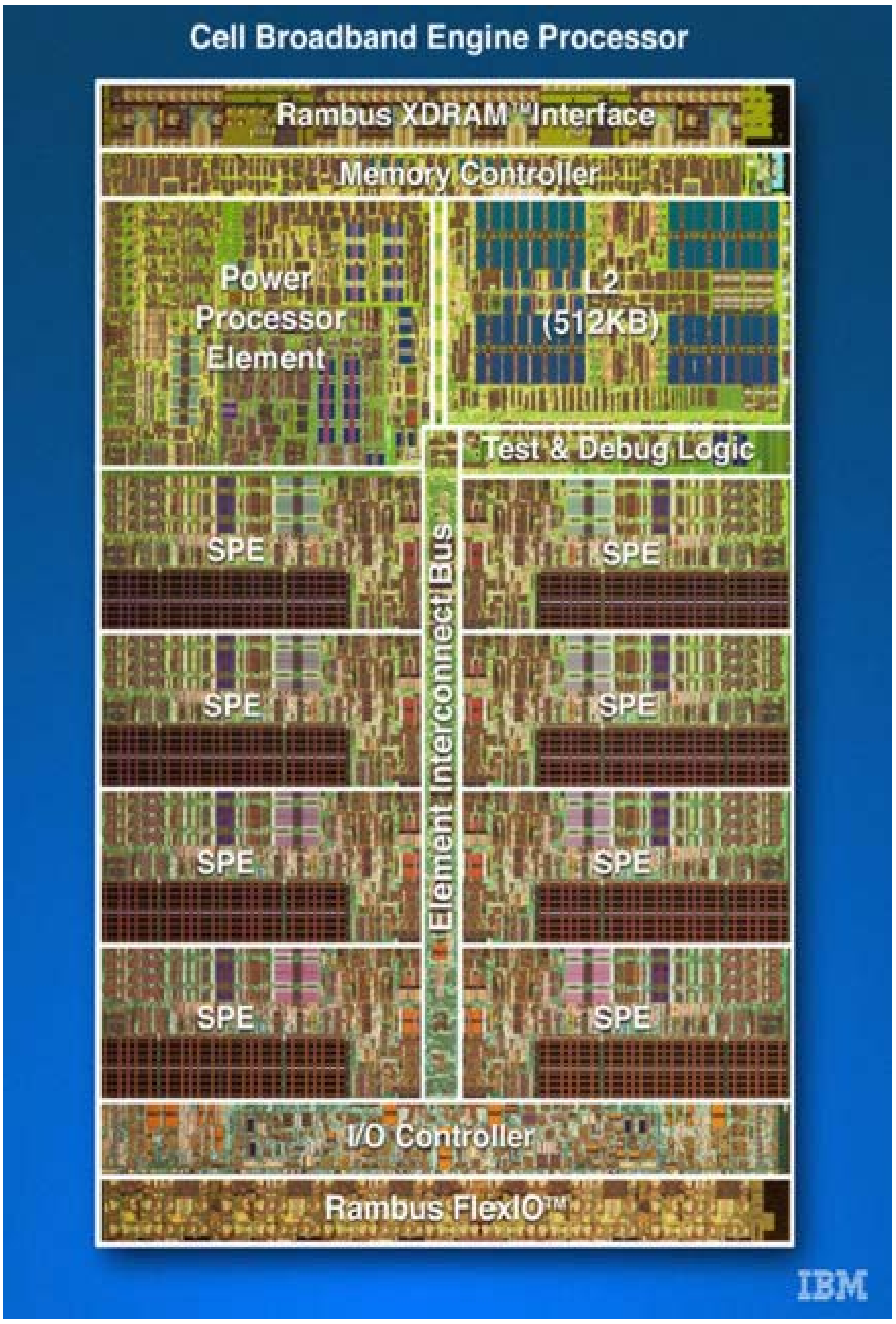


Figure 2-6 - Photo of the Cell Broadband Engine

2.2 Real-Time Kernel

Real-time kernels are a subfamily of kernels. A kernel is a transparent "program" that manages the entire system.

When a computer is started, an operating system is loaded. This system can be user-oriented as Linux, Windows or Solaris, real-time as eCos or µC/OS II, database-oriented, and so on. Linux is a specific case as some variations of the original kernel have been modified to fit real-time purposes.

Any set of systems with their own constraints may have a specific operating system fitting their needs. An operating system is not only a set of applications available to the end-user whether it is human or software. An operating system also includes a kernel.

The kernel configures the CPU, the peripherals, allows processes to communicate between each other, and allows network communication. Moreover, the kernel allows starting processes, managing, synchronizing and stopping them. A kernel is the key to have a functional multi-programmed system.

Kernels can be sorted in two categories. A kernel can be monolithic, meaning that all the functions it provides are included in the kernel itself. Or it can be a microkernel where only the most basic functions are provided by the kernel. The other ones are provided through higher-level programs. There have been endless debates about which type of kernel to use and no clear verdict. For more information about both types of kernel, the debate [79] between A.S. Tanenbaum (creator of Minix) and Linus Torvald (creator of Linux) is of the most interesting.

This is not a problem here as this choice is not necessary for the lower level of the kernel which is the main concern of this project.

There are three elements in a kernel that are necessary and sufficient to run a multiprogrammed system:

- Task management functions, to create and allocate processes
- A scheduler to manage the process ordering on the CPU
- Synchronization functions so that the processes can exchange information.

To get back to real-time kernels, they are slightly different from the other kernels. The problem is that people strongly disagree on the exact definition of a real-time kernel. But the most important point is that it should be predictable. This means that every action must be realized within a specified deadline provided by the kernel manufacturer. It allows the developers to predict the worst-case behavior of the system and ensure that even if things go as slow as they can, their task will be completed within the deadlines.

2.2.1 Task Management

To run applications, it is necessary at first to create them in the kernel. When a process is created (often by another process), it needs to be registered within the kernel and be allocated some memory space. The process data and its instructions must also be loaded in memory. This is called task or process allocation.

When these steps have been fulfilled, the process can be started. Then, the process may be running (in running state) but not actually on the CPU. This is explained in 2.2.2.

2.2.2 Real-Time scheduling

The key element of any kernel, and most critical for a hard real-time one, is the scheduler. A scheduler is a kernel function that manages the execution of multiple processes on one or more CPU.

In a regular personal computer, there is between 1 and 4 CPU, so 1 to 4 processes can run simultaneously. However, there is always dozens of processes running concurrently, the GUI, power management, applications, games… They cannot obviously all run on a CPU at the same time. This is where the scheduler plays its role. It manages the processes list, affect some to CPU, and sometimes preempt (remove it from the CPU, saving its current state) them to run other ones.

There are many scheduling algorithms depending on the type of system they will be used for. A database system (I/O oriented) will not be scheduled the same way as a system calculating Pi decimal.

Interactive systems (Windows, Linux…) tend to have fair scheduling policy. It means that they try to maximize the system performances over time without keeping a process too long away from the CPU, which would look like a really slow application from a user point of view. If a process has been running on a CPU for more than a specified time, the kernel preempt it, store it with the other processes waiting for CPU time, and choose another one to start.

In most of the real-time operating systems, interactivity is not a concern. A real-time scheduler focuses on fulfilling the processes functions within their deadlines. As deadlines and computation times are rarely available to the scheduler, the scheduler must always run the highest priority process. Thus if a high priority process runs for a few minutes, the lower priority processes will not obtain any CPU time and will seem frozen. In certain RT kernels, it is possible to run several processes with the same priority level. In these cases, they often share CPU time, using round-robin or other equivalent algorithms.

To be able to manage processes, the scheduler needs to have some information about them: its address in memory, where is store the execution code, its data, its priority, and most importantly its state. A process is not necessary running or ready to run. A process may have been created, but not yet started as its information is still being filled. A process can also be waiting for a resource, or synchronization, and thus running it would be a waste of time.

2.2.3 Synchronization function

A kernel has to provide synchronization function so processes can exchange information between each other.

On personal computers, it seems that processes are running simultaneously on the CPU as it is possible to work in a text editor, while listening to music and copying file. The sensation is due to the scheduler which allows processes to obtain a CPU time slice at a high frequency. Thus even if two processes have the same number of instructions between two synchronization points, it is highly probable that they will not pass them at the same time, except if they use synchronization functions provided by the kernel. Some kernels provide barriers where a process will stop until a specific event is received.

Another type of necessary synchronization is to manage simultaneous access to shared data in memory. The classic example of the cash point [78] illustrates perfectly this need.

Depending on the kernel theses functions can differ. Their numbers are limited but they are not necessary all available. At least, one mutual exclusion function is necessary (semaphores, mutexes or spin-lock) to be able to run several tasks with shared data.

Note that the kernel itself may use these synchronization functions, as in multi-processor systems, several access to a critical kernel code section can be attempted.

3. A Kernel Architecture for the CBE

The main focus of this chapter is to define a theoretical implementation of the kernel for the Cell. Efficient algorithms will be described although they may not be implemented in this project.

3.1 General design

Symmetrical Multi-Processing is the most common approach on multiprocessor systems. [15] has shown that it is possible to develop SMP kernel in heterogeneous environment. It may be possible to run a SMP kernel on the CBE however performances would not be so good.

The SPEs and their Local Stores are the strength of the CBEA regarding performances and predictability. However, they may introduce extremely long context switch time. In a regular SMP kernel, all the system calls are handled by the processor the task is running on, after a context switch saving the process data and loading the kernel code. On the CBE, this approach means, depending on task allocation and LS management, that the LS would have to be transferred completely to the memory and back at every system call. This means at least 512k bytes transferred on each system call, plus the kernel code that needs to be loaded. With statically defined tasks, it may not be necessary to transfer all the LS back and forth each time. The transfer size could be reduced to the used size in the LS.

The SMP approach thus does not fit the Cell at all for a hard real-time kernel.

The idea of this project is to let the SPEs run computational task while the PPE runs the kernel. The kernel being the only process running on the PPE would then have small response time, and would be able to run complex algorithm for scheduling if needed.

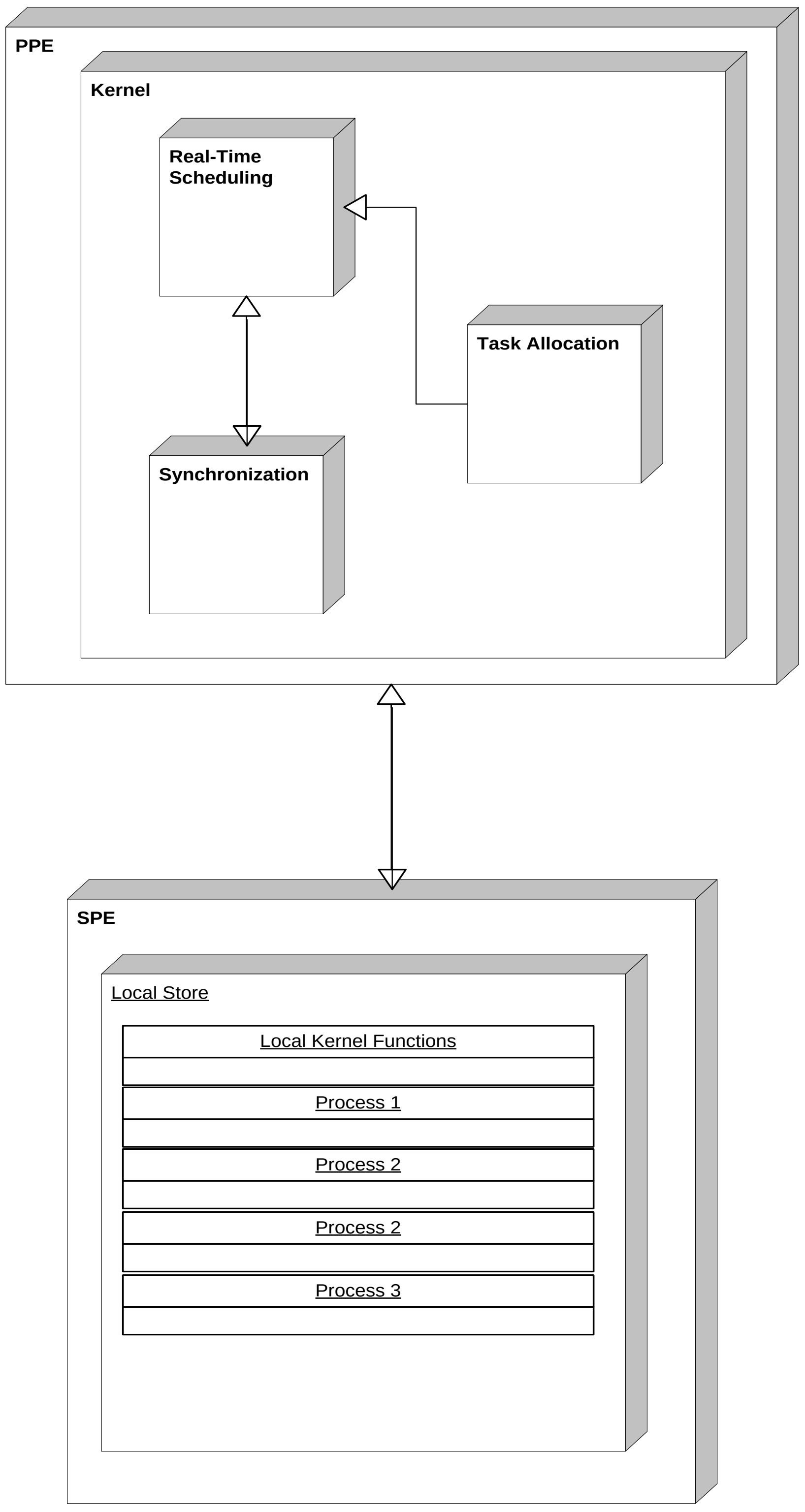


Figure 3-1 - Kernel Overview

3.2 Task Allocation

One of the first concerns in designing the kernel is the tasks or processes. The way they are designed will strongly influence the scheduling and synchronization algorithms.

As said earlier, the SPE Local Stores are problematic for memory context switch as cache memory would be on a regular multi-processor [83]. The task allocation system has thus been designed to minimize the number of memory context switch needed, and in this case, bring it down to zero.

The Local Stores were designed for the Cell to run as a high-end processor. Compared to the Gigabytes of memory available, a 256k Local Store is small and may not be sufficient to fit a single task. However, the Cell is used here as an embedded processor. The point of embedded real-time systems being efficiency, the tasks programs will, in most of the cases, be small. Thus it should fit in the Local Store and not even use it all.

The idea behind this kernel is to statically allocate processes, not to a SPE, but to a precise area in a SPE Local Store. This area would have to be statically defined during the system compilation.

At system startup, the process code would be loaded within this area. The area may be wider than the code requires to store variables, the process stack, and the register value for context switches.

By this mechanism, the memory exchange between Local Stores and the main memory are drastically reduced. Once loaded, the only exchange needed would be for run-time kernel function (system calls, semaphores…). Context switch simply consist in saving all the registers at the appropriate area in the LS, and load the new ones.

The number of task that can be allocated in a Local Store will obviously depend on the task code. Note that if these programs use all the 128 128-bit registers of the SPE, less than 128 contexts fit in the Local Store. The number of available register may be limited at compilation to limit memory usage.

Some kernel functions will also have to be stored on the Local Store. These functions will run in user mode so no context switch is needed. They will allow communication with the other processors (SPEs and PPE), and the kernel to execute certain functions not available from the PPE, such as managing semaphore. The context switch function, triggered by the PPE also resides in this area.

Following is a representation of a Local Store memory map. The kernel function is the only constant part of any LS. Depending on the system compilation, the number of task, the code, variable and task size may change.

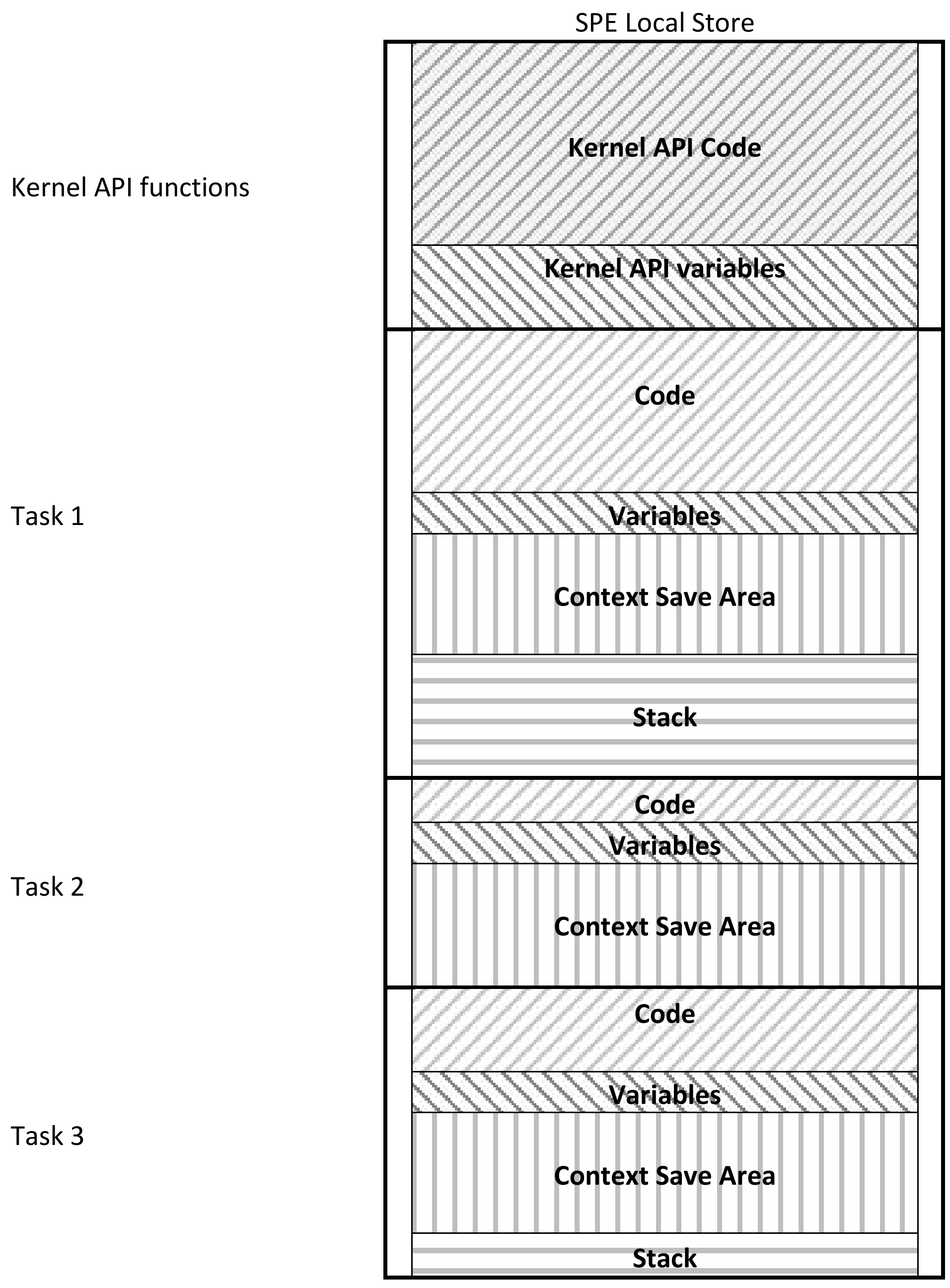


Figure 3-2 - Local Store Memory Map

Following these criteria, a task structure can be defined. To allow the kernel to manage the task, this structure should include:

- The SPE to which the task belongs to
- A pointer to the code area
- A pointer to the variable area
- A pointer to the context save area
- A pointer to the initial stack address.

The kernel functions being always loaded in the space area of the Local Store, tasks can access these functions through a direct pointer defined during the compilation.

3.3 Scheduling

Once tasks have been created and allocated to a SPE, it is necessary to schedule them. There are many possible scheduling algorithms for real-time systems. Moreover, the PPE being only used for the kernel, implementing high complexity algorithm can be a viable solution. Such algorithms are proposed in [49],[62],[36] and [55].

However, in order to keep response time as short as possible, a simple algorithm is used. One of the most common algorithms is a priority driven preemptive scheduler.

In such a scheduler, the highest ready priority task is always running. If a task with a higher priority than the one currently running becomes ready, the running task is preempted, store in memory as a ready task, and the new task is started on the processor.

By using structures equivalent to the one used by the Linux kernel [7], it is possible to achieve a O(1) priority algorithm. This means that the execution time does not depend on the number of tasks ready to run. Such algorithms achieve low and constant response time. To achieve such results, the scheduling algorithm uses a ready task queue per SPE. The task queue is stored in an array, indexed by their priority. With a priority driven scheduling algorithm, finding the next process to run is simply a matter of finding the highest priority in the array pointing to a process and execute it. The algorithm complexity thus depends on the number of priorities and not on the number of processes.

However, unlike Linux, it does not seem appropriate to introduce round-robin algorithms for processes of equal priorities. Round robin algorithms need periodical interruptions for each of the SPE which would be difficult to implement and also putting a much heavier load on the PPE, already managing 8 scheduling algorithms and the synchronization functions of the SPE. Moreover, even with the tasks being statically allocated in the Local Stores, context switching is still a long task requiring inter processor (PPE and SPE) communications, saving hundreds of registers. Thus, a round robin algorithm would slow down the overall system reactivity and render the system predictability more difficult to establish. Therefore, this kernel will never have more than one process per SPE running at a specified priority.

As tasks are statically allocated to a SPE, there is no problem of task repartition in the scheduler to homogenize processor usage, as there is on SMP kernels. Therefore, the scheduler is scheduling each SPE as an independent processor. Data structures must then be duplicated to fit the number of SPE. Each SPE will have its own task list and no other SPE can access or modify it. Although, due to synchronization, a SPE might get a task ready on another SPE but these cases will be managed by the kernel, on the PPE, and described in 3.4.

3.4 Synchronization – Semaphores

Synchronization is the most critical and difficult part of the kernel. More than requiring low response time and predictability, it influences scheduling. When a task is denied access to a critical resource or section of the code, it requires changing its state and putting it in a queue, waiting the resource to become available.

Among the many usual synchronization functions, the semaphores have been chosen to be implemented in this kernel. By successfully implementing semaphores, it will be proven that mutexes and conditions can also be efficiently implemented on the Cell.

3.4.1 Hardware implementation

The first concern specific to the Cell is about the way SPE lock and release the semaphores. On a regular mono-processor kernel, the semaphores are dealt with system calls. It means that the kernel is locking or releasing the semaphores when asked by a task. The task is blocked while the system call is executed.

The problem of this approach on the Cell is that we have only one kernel, running on the PPE. System calls, through interrupts or messages, are thus much slower than on a single processor. Moreover, with 8 SPE potentially requesting semaphores, the PPE might become overloaded, or drastically increase the response times.

Using only the SPE is also impossible. To run efficiently, the semaphores influence the scheduling using priority inversion techniques among other things.

Therefore a mixed solution has to be implemented. If no action from the kernel is necessary, the SPE will use their Atomic Cache Unit (ACU) [29] which is a small cache containing 6 lines of 128 bytes. The SPE can thus simulate atomic operation on shared-memory by verifying the cache line is not dirty when sending it back to the central memory.

Other actions that require the kernel will use blocking and non-blocking system calls. System calls can be implemented in multiple ways. The simplest one is to use the mailbox provided by the Cell to communicate between the PPE and the SPE. However, according to general opinion, this is not the fastest solution.

The most common approach is to use in parallel the Stop-and-Signal function of the SPE, and Direct Problem-State Register Access (DPSRA). Basically, the SPE saves the value of the system call in one of its user register. It then executes the stop-and-signal function which stops the SPE and sends a signal (interrupt) to the PPE. The PPE then handles the interrupts by retrieving the system call value through DPSRA, which is a DMA transfer of a SPE user-state register to the PPE cache.

The problem of this last approach is that the SPE is stopped until the system call is executed. For blocking system calls, this solution is working. But not for non blocking system call. A simple signal would not be working either. The SPE running, the user register may be rewritten before the PPE had time to access it.

Therefore both of the previous solutions shall be implemented. The stop-and-signal used for blocking system calls and the mailbox for non-blocking system calls.

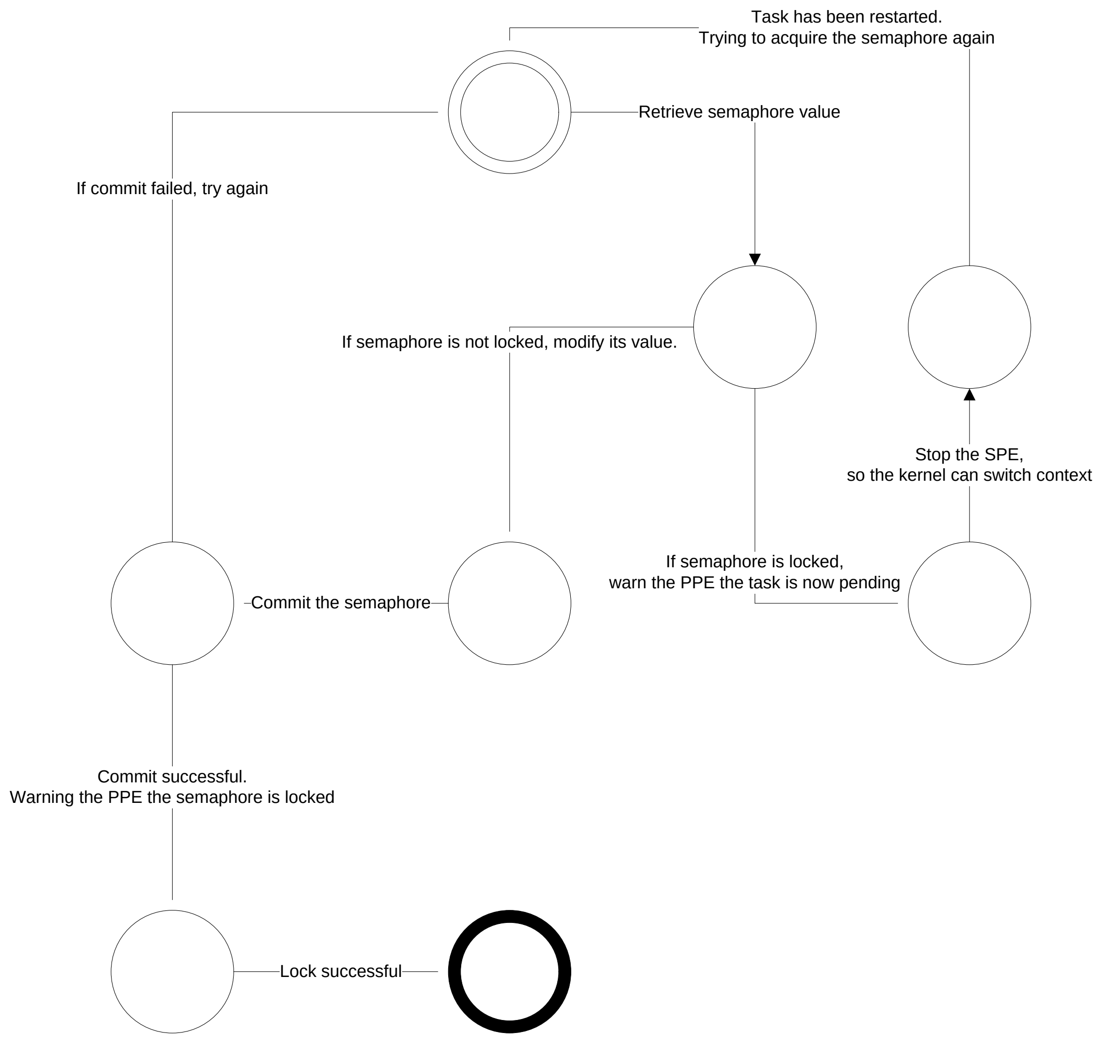

Task has been restarted.
Trying to acquire the semaphore again
Retrieve semaphore value
If commit failed, try again
If semaphore is not locked, modify its value.
Stop the SPE,
so the kernel can switch context
If semaphore is locked,
warn the PPE the task is now pending
Commit the semaphore
Commit successful.
Warning the PPE the semaphore is locked
Lock successful

Figure 3-3 - State representation of a semaphore acquisition using the ACU

3.4.2 Priority Inversion

As said earlier, the semaphores influence the scheduling. Here is why: Let's imagine a single processor running a preemptive priority driven scheduler. 3 tasks (p1, p2, p3) have been created on the kernel. We have priority of p1 > priority of p2 > priority of p3. Therefore if the 3 tasks are ready, p1 will be the one running.

At the time t0, only p3 is ready, so the scheduler starts p3. P3 locks a semaphores and starts running the critical section.

At this point, p1 becomes ready. As p1 has the highest priority, it starts running. P1 then tries to lock the semaphore p3 is already using. The semaphore being used, p3 is denied the access to the critical section and is stored in a waiting queue by the kernel.

Then the task p2 becomes ready and starts running, p2 having a higher priority than p3. P3 and p1, blocked by the semaphore, will then be blocked until p2 finishes.

The result is a task indirectly blocking a higher priority task (p2 blocking p3 thus blocking p1). This is not acceptable for a real-time kernel. P2 computation time being unknown, p1 will be blocked for an unknown amount of time and will probably fail to complete within its deadline. However, p3 blocking p1 is acceptable. P3 is in a critical section so it can never be preempted by another process which attempts to access this same section.

The result we would have wanted to obtain for this scenario is p3 starts running and get into its critical section. P1 become ready and get stored in the semaphore queue. When p2 gets ready, it stays in the ready queue and does not preempt p3. When p3 exits the critical section, p1 is freed will run until completion. Then p2 will be able to complete too.

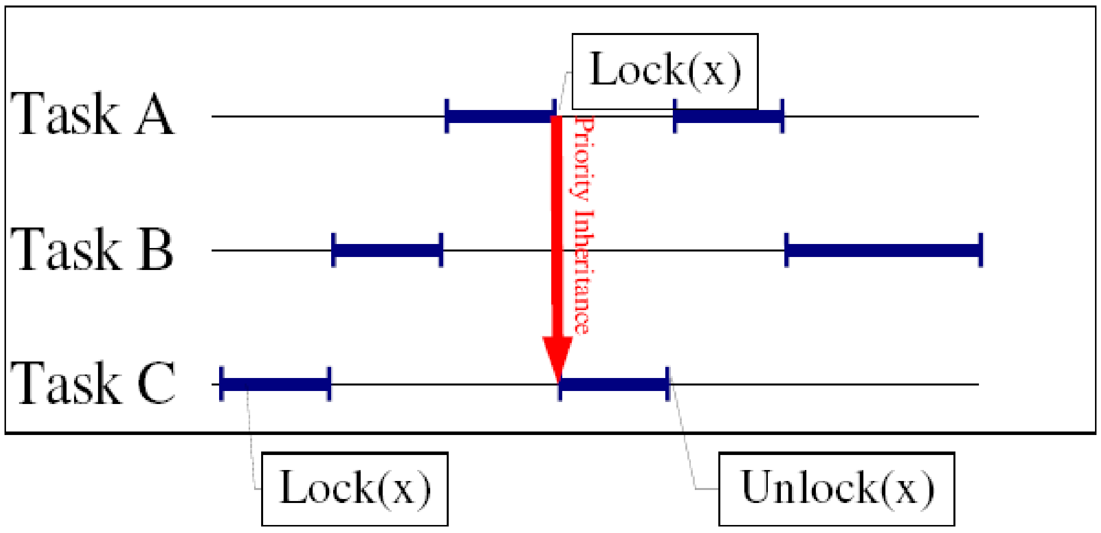


Figure 3-4 - Priority Inheritance

This solution seems to be contrary to a priority driven kernel. But by introducing two priorities per task, a static and a dynamic one, this problem can be solved. This solution is called the priority inversion protocol [66].

The static priorities are the one defined earlier. They are allocated at the task creation and will never change. Dynamic priorities may change. When a process is not in a critical section, its dynamic priority is equal to its static priority. However, when a process is in a critical section, its dynamic priority is equal to the highest static priority of the tasks in the pending queue of this semaphore. Therefore, no task can be blocked by a lower priority task.

On a multi-processor kernel, the problem is more difficult. Although such an algorithm can be used, it presents some problems. A solution has been proposed in [61], the multiprocessor priority ceiling protocol. More than avoiding unwanted preemptions, it avoids deadlocks and minimizes the blocking times. The multiprocessor priority ceiling protocol fits perfectly the requirements of a hard real-time kernel for the Cell.

3.4.3 Task dispatching

Another complex problem linked to semaphores is the task dispatching problems. When a task tries to acquire a semaphore which is already fully used, the task is stored into a pending queue attached to the semaphore. When the semaphore is released, one of the pending tasks has to be released. The task dispatching problem consists of choosing which task will be freed.

An interesting solution is proposed in [50].It uses an heuristic solution to analyze the tasks behavior and give them a priority in the pending queue depending on this result. However such an algorithm seems unfit for the Cell as the PPE would have to run heuristic algorithms to analyze the tasks from 8 processors using semaphores.

A simplest approach to the problem would be to free the task with the highest priority. Although it may be working on a single processor system, it is not adapted to multiprocessor systems. While the freed process may have the highest priority in the pending queue nothing ensures that its priority will be higher than the current process running on the SPE. Thus the process will not be allowed to run and will be put in the ready process queue. All the tasks still waiting in the semaphore queue will be stuck until the freed task is allowed to run and complete.

Another simple solution that fit the CBE is proposed here. Each semaphore will use 8 pending queues, one per SPE. Each of these pending queues will be a priority sorted list. Once the semaphore is released, the kernel will search for the pending process having the highest priority and could run immediately if freed. This means that the sleeping process priority (eventually the dynamic priority he would have if freed) has to be higher than the priority of the process currently running on its SPE. If such process does not exist, the highest priority pending process is released.

Such an algorithm is fast (O(1) complexity), easy to implement and predictable.

3.5 Interfacing with Linux

Although the kernel just discussed might be more efficient, a complex kernel is not necessary to prove the hypothesis. Therefore a simpler version of the kernel will be implemented. The data structures and the global organization will not change but some algorithm will be simplified to ease the development process.

A more detailed description of the software is available in Chapter 4.

4. Software Implementation

In this part, the implementation of Virt-K (Virtual Kernel) is described. Virt-k is a proof of concept of the ideas proposed before. Due to the limited time for this project, the kernel has not been implemented as low-level drivers, hardware management function and compilers modules would have to be written. Therefore, these ideas have been implemented as a virtual kernel running over Linux. As Linux does not schedule the SPE (once a task is started on the SPE it is never preempted unless the task asks to), the response times on the SPE themselves are significant. However, the response times on the PPE are not. The PPE part of the kernel will be executed as a user task within Linux. As the Linux distribution that is used is not real-time and running over tasks, the response times it provides may not be significant at all.

Only the global architecture and features of Virt-K will be described here. However a more complete documentation of the structure and functions is available with the sources on the Sourceforge project [56] under the GPLv2 license.

Virt-K has been developed on an x86 Fedora Core 7 using the IBM Cell Development Kit. All the functionality testing has been realized on the IBM Cell simulator (systemsim), though the final and performances tests have been done on a Playstation 3 running Ubuntu.

4.1 Software architecture

The architecture of Virt-K will be detailed into two parts: the PPE part, which is the kernel, and the SPE part, which is where the users have their tasks running.

4.1.1 PPE – Kernel

As Figure 4-1 shows, the kernel runs multiple threads. Such an approach would not be realistic for a real kernel as it would be necessary to implement a scheduler for the PPE. However, it simplifies the development task for this proof of concept.

The first set of threads is necessary in the current Linux implementation to run a program on a SPE. To start a task on a SPE, it is necessary to call a function which only returns once the program stops, hence the need of a thread per used SPE.

The other two threads are the real core of the kernel. The scheduler thread obviously realizes the SPE scheduling. The algorithm is quite simple:

```
While (true)
     For each SPE
          If Need Reschedule Flag is set for this SPE
               Acquire lock on SPE
               Reschedule the SPE
               Release the lock
          End if
     End for each
End while
```

The scheduling algorithm itself has been described in chapter 3.3.

The last thread entitled Mailbox management manages communications between the PPE and the SPEs. For optimum performance, these functions should not be done in a thread but in an interrupt handler triggered by the reception of a message from a SPE. It has been done as a separate thread in Virt-K as it simplifies synchronizations between the scheduler and the communication parts.

What the mailbox management thread realizes is reading all the inbound messages, sent from the SPE, modifying some data linked to the task or semaphore if necessary and sets flags.

For example, when a SPE sends a message to the PPE to make it aware it is pending for a semaphore, the mailbox thread attaches the SPE task to the semaphore, calculates the new dynamic priority of the process currently owning it, and sets a flag so the SPE will be rescheduled.

4.1.2 SPE – User-space

The SPE software architecture is simpler than the PPE one.

The first set of tasks available here is the user tasks plus the null task, a task which is always ready so the scheduler can always find a task to run. Except for the null task, all of them are user-defined, depending on their needs.

The other task is the switch task. It is used during context switching.
When the PPE needs to reschedule a SPE, it sends the SPE a message containing a task identifier. The reception of a message triggers an interrupt on the SPE. The interrupt handler saves the registers, the stack pointer value and the program counter (except for the null task). It then returns from the interrupt, not in its original task, but in the switch task. The switch task reads the message sent by the PPE and triggers a context restore depending on the task (null task, new task, task already ran).
The switch task has its own context but it is smaller than a usual one. As the interrupt handler "call" the switch function as returning from the interrupt, it is not necessary to save the register of the exact stack pointer value. Switching to switch task simply requires setting the stack pointer to the kernel stack value and jumping to the right address in the LS.

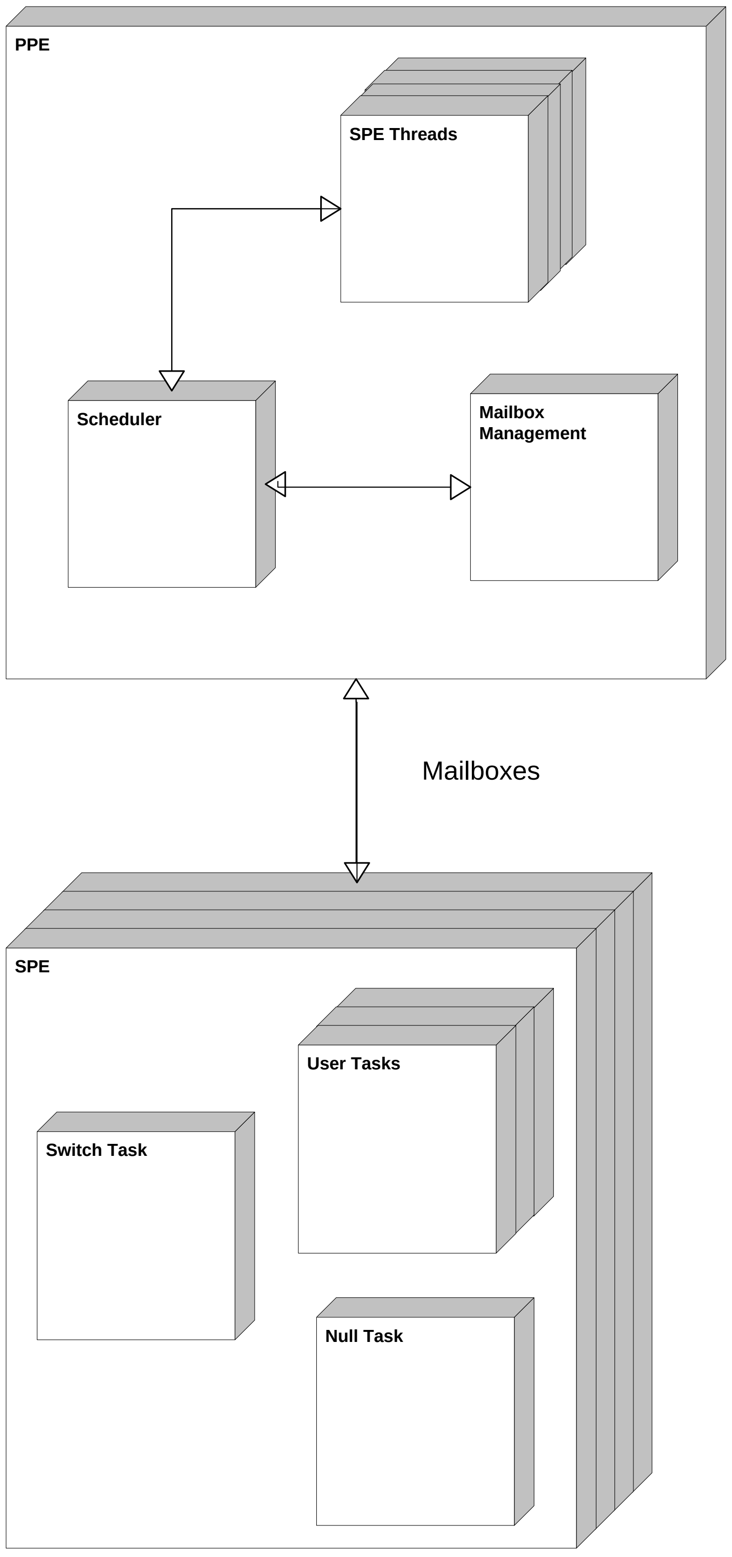


Figure 4-1 - Software architecture

4.2 Implementation specifics

As explained earlier, Virt-K is a proof of concept. Therefore, it differs from the ideas detailed in Chapter 3.

For example, as said in chapter 4.1.1, the kernel is divided into two threads which are incompatible with a non-virtual kernel.

4.2.1 LS Memory organization

As Chapter 3.2 describes, tasks are statically allocated to a SPE local Store. However, due to the usage of gcc compiler, and Linux kernel loader, the position of a task in the SPE is not easily predictable. It could be retrieved from the object file generated for the SPE but it would be unpractical.

Therefore, few modifications have been made to Virt-K:

- Tasks on a SPE are identified by a byte identifier
- The SPE stores a structure indexed by task identifier to retrieve stack pointer, program counter and the task status (new / ran)
- Holes cannot be let in the LS to fit the stacks, so tasks are statically allocated at compilation as global arrays.
- In PPE/SPE communication, the task ID is used instead of a task pointer

4.2.2 PPE/SPE communications

In chapter 3.4.1 was discussed the best implementation for PPE/SPE communications. It was said that mailboxes are better for non blocking system calls but that stop-and-signal with DPRSA would be better for blocking system calls.

To simplify the implementation, only the mailbox systems have been used though for blocking system calls, the SPU is stopped, but the signal is not used.

This results in a slightly slower response time for blocking system calls but it is not necessary significant compare to the SPU stop and restart time.

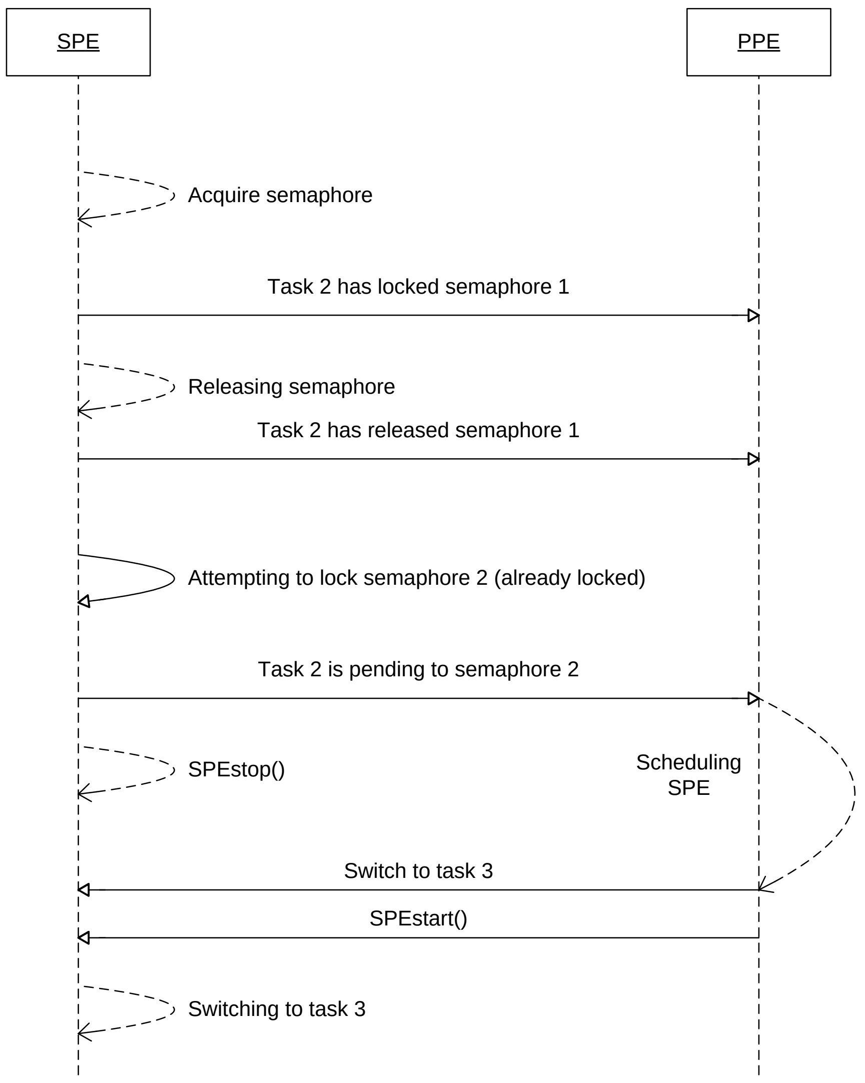

SPE
PPE
Acquire semaphore
Task 2 has locked semaphore 1
Releasing semaphore
Task 2 has released semaphore 1
Attempting to lock semaphore 2 (already locked)
Task 2 is pending to semaphore 2
SPEstop()
Scheduling SPE
Switch to task 3
SPEstart()
Switching to task 3

Figure 4-2 - Example of PPE/SPE communications through mailboxes

4.2.3 Conditions

To ease the realization of functional tests, conditions have also been implemented. They use the same mechanisms as semaphores except that there is no priority inversion, and when a task signals a condition, all the pending tasks are set back into ready mode. All the SPU are also rescheduled.

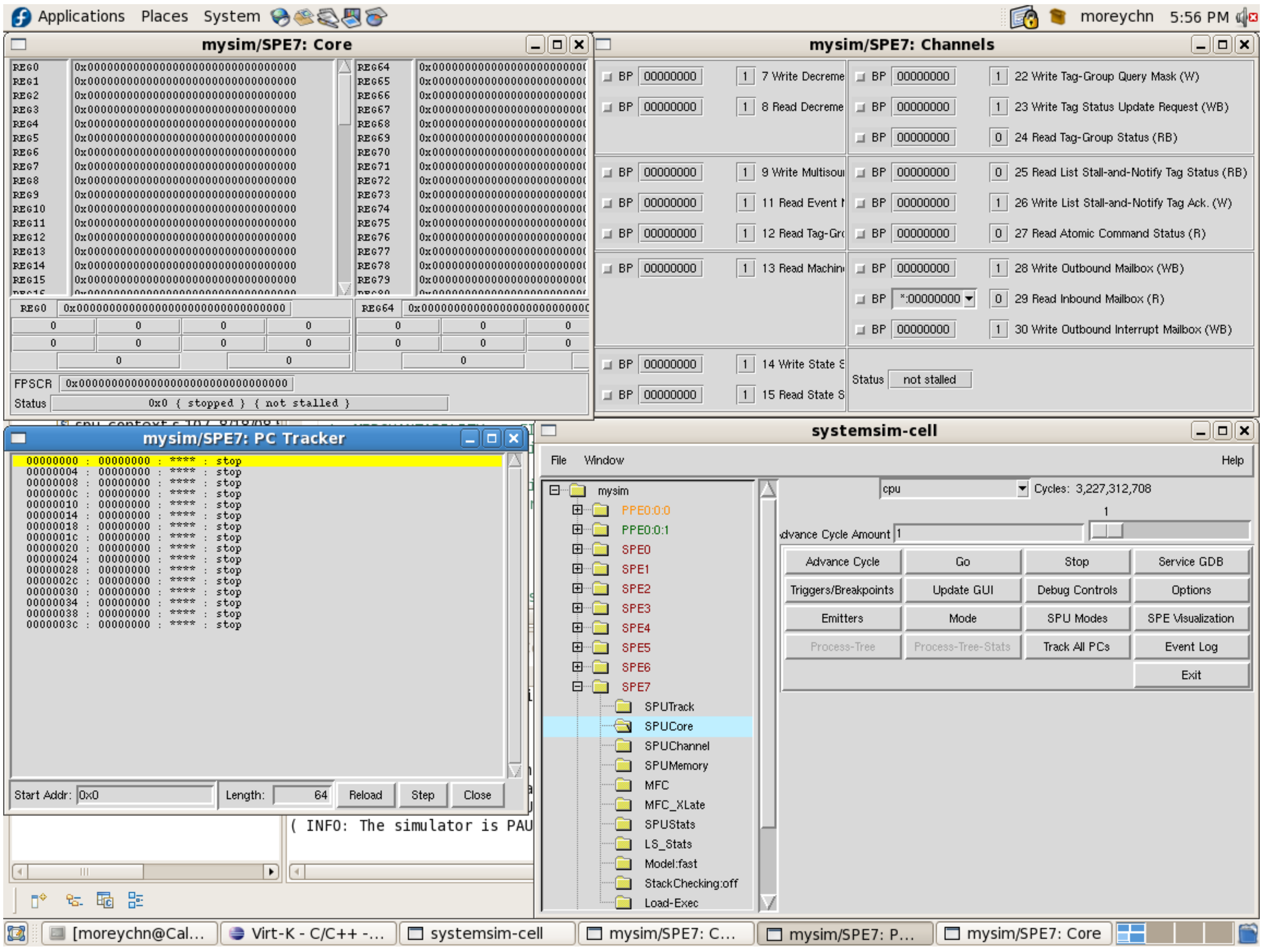
4.3 *Software status*

4.3.1 Development Environment

Virt-K has been developed under GPLv2 License using C language, except for the context saves and restores functions which have been written in assembler. Source and header files are fully documented using Doxygen. The development was managed and backuped using Subversion.

All the development has been realized on Eclipse with the Cell SDK plug-in, running on an x86 Fedora Core 7.

The final execution platform is a Playstation 3 running Linux Ubuntu.



```
REG0    0x00000000000000000000000000000000    REG64   0x000000000000000000000000
REG1    0x00000000000000000000000000000000    REG65   0x000000000000000000000000
REG2    0x00000000000000000000000000000000    REG66   0x000000000000000000000000
REG3    0x00000000000000000000000000000000    REG67   0x000000000000000000000000
REG4    0x00000000000000000000000000000000    REG68   0x000000000000000000000000
REG5    0x00000000000000000000000000000000    REG69   0x000000000000000000000000
REG6    0x00000000000000000000000000000000    REG70   0x000000000000000000000000
REG7    0x00000000000000000000000000000000    REG71   0x000000000000000000000000
REG8    0x00000000000000000000000000000000    REG72   0x000000000000000000000000
REG9    0x00000000000000000000000000000000    REG73   0x000000000000000000000000
REG10   0x00000000000000000000000000000000    REG74   0x000000000000000000000000
REG11   0x00000000000000000000000000000000    REG75   0x000000000000000000000000
REG12   0x00000000000000000000000000000000    REG76   0x000000000000000000000000
REG13   0x00000000000000000000000000000000    REG77   0x000000000000000000000000
REG14   0x00000000000000000000000000000000    REG78   0x000000000000000000000000
REG15   0x00000000000000000000000000000000    REG79   0x000000000000000000000000

REG0  0x00000000000000000000000000000000     REG64  0x000000000000000000000000
   0      0      0      0                        0      0      0
   0      0      0      0                        0      0      0
       0             0                                0

FPSCR  0x00000000000000000000000000000000
Status           0x0 { stopped } { not stalled }
```

```
BP  00000000   1  7 Write Decreme     BP  00000000   1  22 Write Tag-Group Query Mask (W)
BP  00000000   1  8 Read Decreme      BP  00000000   1  23 Write Tag Status Update Request (WB)
                                      BP  00000000   0  24 Read Tag-Group Status (RB)
BP  00000000   1  9 Write Multisou    BP  00000000   0  25 Read List Stall-and-Notify Tag Status (RB)
BP  00000000   1  11 Read Event N     BP  00000000   1  26 Write List Stall-and-Notify Tag Ack. (W)
BP  00000000   1  12 Read Tag-Gro     BP  00000000   0  27 Read Atomic Command Status (R)
BP  00000000   1  13 Read Machin      BP  00000000   1  28 Write Outbound Mailbox (WB)
                                      BP  *:00000000 0  29 Read Inbound Mailbox (R)
                                      BP  00000000   1  30 Write Outbound Interrupt Mailbox (WB)
BP  00000000   1  14 Write State S    Status  not stalled
BP  00000000   1  15 Read State S
```

```
00000000 : 00000000 : **** : stop
00000004 : 00000000 : **** : stop
00000008 : 00000000 : **** : stop
0000000c : 00000000 : **** : stop
00000010 : 00000000 : **** : stop
00000014 : 00000000 : **** : stop
00000018 : 00000000 : **** : stop
0000001c : 00000000 : **** : stop
00000020 : 00000000 : **** : stop
00000024 : 00000000 : **** : stop
00000028 : 00000000 : **** : stop
0000002c : 00000000 : **** : stop
00000030 : 00000000 : **** : stop
00000034 : 00000000 : **** : stop
00000038 : 00000000 : **** : stop
0000003c : 00000000 : **** : stop

Start Addr: 0x0    Length: 64
```

```
( INFO:  The simulator is PAU
```

```
cpu    Cycles:  3,227,312,708
dvance Cycle Amount  1
```

Figure 4-3 - IBM Cell SystemSim Simulator

4.3.2 Development Status

Virt-K is currently available as a release candidate. Though it is incomplete and still includes some bugs, it has reached a step where it can be used by developers as a SPE framework.

4.3.3 Optimizations

Virt-K trying to prove it is possible to run an efficient hard real-time kernel on the Cell, it has been highly optimized. These optimizations are described in the source comments and in the Doxygen documentation.

For example, a semaphore only needs a single bit to store its value. Therefore, by using bit fields, up to 8 semaphores per byte could be stored. However, to retrieve a semaphore, SPE use the ACU which transfer 128bytes. Then if multiple semaphores are stored in a byte or even in the surrounding bytes, another SPE reading or modifying this other semaphore would result in removing previous reservation by SPE other SPE trying to lock other semaphores. Thus, semaphore values are stored in 128bytes array, decreasing the average semaphore acquisition time.

4.3.4 Functional test

In parallel of the development process, intensive testing has been realized. All the individual pieces (scheduler, context switch, mailbox management, semaphores) have been tested individually before being merged.

The first stages of the functional testing have been done on IBM Cell Simulator (SystemSim), which allows cycle by cycle execution and full access to the Cell registers. The final tests had to be realized on a Playstation 3 due to the low performances of the simulator.

The entire performance testing has been realized on a Playstation 3.

4.4 Known problems

Although Virt-K is running there are still few problems left. Some of them are bugs which have not been found yet, but mostly unsolved problems.

4.4.1 Context Restore

This problem is due to the fact that tasks are not statically allocated and the SPE is managing the context values instead of the PPE.

When a context is being restored, the stack pointer is restored first; all the registers values are retrieved from the stack. The process needs then to jump back where it was in the task. However, the SPE instruction set doesn't provide a function to jump to an address stored in the LS. This address has to be loaded in a register first, therefore overwriting the value the register had before being switched. For this purpose, R127 (128th register) has been sacrificed.

An analysis of object files generated by the compiler has never shown this register being used.

This problem has not been solved because it is a problem met only in this proof of concept. On a non-virtual kernel, with statically allocated memory area, the PPE would be storing the context pointers. Thus when a SPE has restored its stack and register, it would stop and signal the PPE that it is ready. The PPE would then run the SPE from the right address, keeping the context untouched.

4.4.2 Semaphores and ACU

As described in chapter 3.4.1, to acquire a semaphore, the ACU is used. Through the ACU, the SPE acquire a 128-bytes line from the memory through DMA transfer. The SPE is then free to read/modify its local copy of the value. However, when the SPE tries to commit it, the ACU check a cache table to see if the value used by the SPE is still valid (not read or written by any other SPE). In the case of such an event, the semaphore lock function tries to acquire the cache line again.

```
Init:
     Retrieve cache line from memory
     If semaphore is locked
          Send message to kernel
          Stop
     Else
          Semaphore = 0
          Commit semaphore to memory
          If commit was successful
               Return
          Else
               Goto Init;
End
```

The problem of such an approach is that it is not predictable. In a worst case scenario, several SPE could be trying to lock the same semaphore at the same time. It would probably results in failure to commit for all of them and have them trying again to acquire the cache line. This problem will be discussed further in Chapter 6.1.2.

4.4.3 Shared variables

In its current implementation, Virt-K does not manage shared variables. It could probably be implemented as a user task, though some change in the kernel would be necessary to allocate memory for these variables.

For the Barber Problem test (see Chapter 5), a quick solution has been implemented: semaphores are used as shared variables. New functions have been added to allow reading and saving the variable to the main memory, however there is no protection against concurrent access. User programs have to lock access to these variables through a semaphore.

5. Performances

In this chapter, the performances of Virt-K are detailed. Virt-K being a virtual kernel and running over Linux, the response times of the kernel are not significant. Therefore, the performance tests have been focused on the SPE side, measuring, semaphore acquisition, release and context switch times. In the first part, a theoretical approach of these times is provided. Then, a practical test has been implemented to retrieve these values.

5.1 Timing Analysis

In this first part a theoretical timing analysis of the critical times in Virt-K is provided. This analysis is solely based on the software architecture and provides a worst-case response time. Some hardware dependant (interruption propagation time, DMA transfer delay) values necessary for these analyses being not available, constants will be used to provide an approximation.

5.1.1 Context switch

Context switches can be split into 3 parts: saving the context, updating kernel and hardware status, restoring a new context.

Context saving and restoring have the same organization. After testing which kind of task has to be saved or restored (null, new or ran task), the necessary registers are saved/restored.

For the null task, no values are saved. Restoring it simply jumps to the beginning of the function with the matching stack pointer restored. This stack pointer is never saved.

For a new task, there is no need to restore a context. The kernel simply loads the matching stack pointer and jump to the start address.

For a task which had already run, all 128 registers are stored on the task stack, with the program counter. The stack pointer is saved in the task management structure in the Local Store.

Thus execution time of context saves and restores are:

- $Context\_save(task) = Checking\_type\_of(task) + \begin{cases} 0\ if\ null\ task \\ Save\ 128\ regs + PC \end{cases} +$

  $Restore\_switch\_SP + Jumpto\_switchtask$

- $Context\_restore(task) = Checking\_type\_of(task) + Restore\_sp(task) +$
  $\begin{cases} 0\ if\ null\ \ or\ new\ task \\ Restore\ 128\ regs + PC \end{cases} + Jumpto\_pc(task)$

Moreover, the switch task resets the interrupt mask and flags, reads the message from the PPE with the new task ID and retrieves the stack pointer from the Local Store.

Its execution time is then:

- $Switch\_task = Reset\_interrupt + Read\_mailbox + Retrieve\_PC$

Therefore we obtain a global execution time of a context switch of

- $ContextSwitch\_time(newtask) = Context\_save(current\_task\ ) + Switch\_task + Context\_restore(newtask)$

These values will be discusses in Chapter 6.1.1

5.1.2 Semaphore Acquisition

As described in Chapter 4.4.2, semaphore locks are problematic due to their unbounded response time.

A formula for semaphore lock response time, when the semaphore is not locked, is:

- $Semaphore\_lock\ = Acquire\_line + Modify\_semaphore + Commit\_line + \begin{cases} 0\ if\ successful \\ Semaphore\_lock\ if\ not \end{cases}$

As the formula shows, the semaphore lock response time is a recursive formula. The issue is that the success or failure of the commit action is unpredictable in the case of a semaphore used on multiple SPE. Moreover, acquiring and committing cache lines depends on the use of the EIB. The ACU thought being atomic, uses DMA transfer over the EIB. Therefore, its speed depends on the EIB usage, which means the more semaphores are used, the slower a lock is. But also the SPE could already own a clean version of the semaphore in its cache. There would be thus no data transfer required to retrieve the semaphore’s value.

This issue will be further detailed in Chapter 6.1.2.

5.1.3 Semaphore Release

Semaphore releases are a much simpler problem than semaphore locks. The issue of semaphore locks comes from the need of “memory transactions” which can fail and need to be executed again. However, releasing a semaphore is a truly atomic action. This action consists of writing the semaphore value in the Local Store, and committing it to the main memory. The value is not read from the main memory before committing it, thus removing the need of a transaction system.

The formula for semaphore releases is:

- $Semaphore\_release = Writing\_sem\_to\_LS + Commiting\_cache\_line$

However, as for semaphore locks, the time needed to commit the information to the memory depends on the EIB usage.

5.2 Performances Tests

Timing analysis is a useful tool for schedulability analysis. However, the formulas presented earlier include much uncertainty, mainly due to unknown hardware response time. Moreover, even if such times were available, timing analysis focuses on the worst-case scenario.

Therefore, practical performance tests have been realized. On the one hand it gives an approximation of these hardware response times. On the other hand, it provides the average values, which is much more efficient for performance analysis.

5.2.1 Test Case

To test the performances of Virt-K, a well known synchronization problem has been implemented. It is called the Sleeping barber problem [78].

This problem simulates a barber shop. The barber shop has X barber working, X barber chairs and Y chairs for waiting customers.

A barber cuts hair as long as there are waiting customers. If there are no waiting customers, a barber falls asleep on his chair.

When a customer arrives, he sits on a waiting chair. If he is the only customer, he wakes the barbers. If the customer arrives and there are no waiting chairs, he leaves.

This problem is simulated by implementing one thread per barber, and one thread per customer. In this test case, we use 6 SPE. Three of them program one barber and one customer each. The other three program one customer each. The shop has 2 waiting chairs available.

This specific problem is particularly adequate as it requires semaphores to access the number of waiting customers, but also conditions to wake up the sleeping barbers.

To execute such a problem on Virt-K, functions to read and write variables from the main memory have been added. They are not part of the regular implementation, but were necessary to run this test case.

It is also important to note than the implementation of the sleeping barber problem has not been optimized. This test case focuses on kernel function response time and not on overall performances.

5.2.2 Timing solution

To acquire precise timing of the kernel function, the SPE decrementers have been used. They are hardware register decreased at regular interval without any software requirement.

Therefore, measuring these function times consist of setting the decrementer value to 2^{32}at the beginning of the function, reading its value at the end and taking the difference. The rate at which the decrementer decreases is available within Linux kernel information.

5.2.3 Results

Figure 5-1 presents the results for semaphore lock times. As expected, the responsive repartition shows two steps:

- One around 125ns which matches cases where the semaphore is already in the local ACU.
- One around 31µs which matches regular semaphore access from distant memory

It also shows that there are much higher values. Only the smallest values are shown but the test cases reported response time over 5ms.

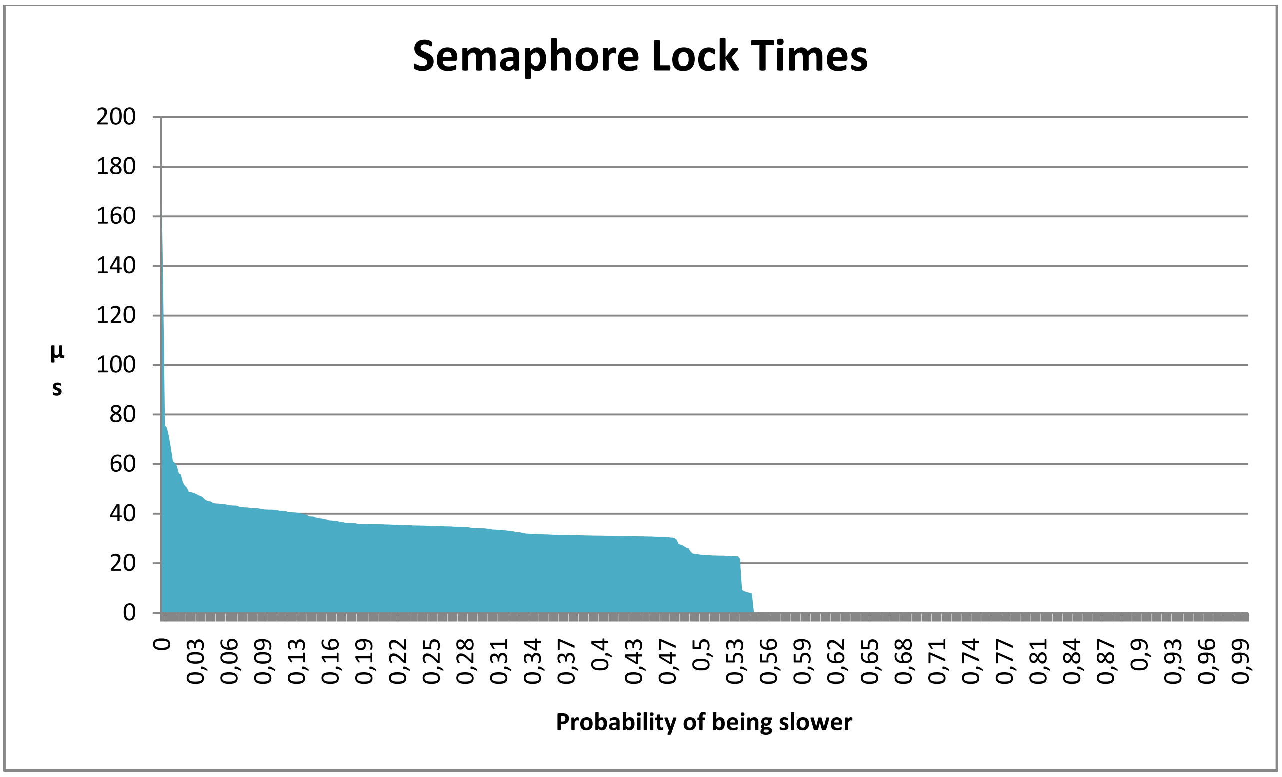


Figure 5-1 - Semaphore Lock Times

A semaphore release matches a semaphore lock from the local ACU as it consists of sending a message to the PPE and committing the value to the ACU. Its execution time is thus around 125ns.

For context switch, values are similar for equivalent switches (no null or new task). The maximum gap between two context switches is around 20ns and is due to reading and writing channels (for mailbox and interruption management).

In average, a context switch requires 1.6ms.

These results prove that the Cell Broadband Engine can achieve high performances as a real-time processor. However, they also highlight some problems due to the Cell architecture:

-Due to the fact SPE are distributed processors, constant semaphore acquisition time is extremely hard to achieve.

-Due to the large number of registers, context switches are slow compared to usual embedded processors. Virt-K however achieves much lower context switching time than Linux.

These problems will be discussed in details in the next chapter.

6. Weak Points and Possible Enhancements

The proof of concept, Virt-K, has shown some flaws in the kernel designs, and also provided some useful performance analysis which could lead to performance enhancement.

6.1 Complex scheduler

The design being focused on minimizing response times, it was decided to restrict the scheduler to a preemptive priority-driven scheduling algorithm. However, the sleeping barber test case has shown that even with Linux kernel and over software running concurrently running on the PPE, its workload is still low.

Therefore, complex scheduling algorithms, as discussed in chapter 3.3 could be implemented. Response time would probably be slightly increased, but the overall performances of the system would be greatly increased.

6.1.1 Context Switches

As the performance results shown, a context switch requires about 1.6ms. Although this is much faster than a full context switch (saving and restoring all the LS to the main memory), it is still slow.

The number of register to save and restore is what is slowing down context switches. On an x86 processor, there are only 4 main registers plus a couple for flags, program counter… The SPE have 128 128bits registers. A solution would be to limit the number of registers used by a task or at least use this number to save and restore only the required number of registers.

For example, in the sleeping barber test case, no tasks use more than 13 registers. By using this parameter, Virt-K could save much memory on the stack (leaving space for only 13x128bits = 208 bytes instead of 128*128bits = 2048 bytes), but more importantly nearly divide context switch time by 10.

This flaw in Virt-K is only due to its implementation. In a non-virtual implementation, a restricted context switch algorithm could be easily implemented.

6.1.2 Semaphore Locks

As explained in chapter 5.1.2, and confirmed by the performance analysis, semaphore acquisition presents a predictability problem. Although in most cases, semaphore locks are fast, the execution time is not bounded.

Therefore, Virt-K does not fit the requirements for a hard real-time kernel. Hard real-time kernel requires predictable performance to perform schedulability analysis and ensure that the tasks will complete within their deadlines.

However, Virt-K and the idea it is demonstrating are still eligible for a soft real-time kernel. The average low response time for semaphore acquisition allows high performance, and unpredictability is not critical for a soft real-time system.

A solution to this problem would be to implement semaphore lock through system calls. Performance would be drastically reduced in average but the execution time would be bounded and thus predictable. A mix of both solutions could also be implemented. The SPE would try to acquire the semaphore through the ACU. If the commit action fails, the acquisition would be done through a system call.

This solution provides the same low execution time as the current implementation for successful cases, but bound the worst case execution time. However, it may be difficult to implement as the value is still shared between PPE and SPE.

6.2 Shared variables

A critical point which appeared during the test case implementation is the necessity to provide shared variables. The solution used in the test case is not optimal and should not be used in production.

The Cell and more precisely the SPE provide efficient hardware to solve this problem. Each SPE owns a MFC which allows asynchronous DMA transfers. Which means a SPE task can request a data area to be transferred from the main memory and continue computing other data while it is retrieved in the LS.

An efficient implementation would be to request these data so they are retrieved when the task needs them. However concurrent access to these data makes it more difficult.

As these data are shared variables, other tasks or processor may use them between the moment the transfer is requested and the values are sent back to the main memory. Therefore it is necessary to protect this critical section by a semaphore. But then, by requesting an early DMA transfer to limit waiting time on the SPE, the length of a critical section is increased which is slowing down the overall performances of the kernel.

An interesting solution would be to adjust the time at which the DMA transfer is requested depending on the static task priority. A high priority task needs to be completed as fast as possible whatever the cost are on the overall performance. So the DMA request, and the semaphore acquisition, should be done early to ensure the data will be there when needed. For a low priority task, the request should be done at the last moment to limit the impact on the system performances.

In the case of Virt-K, it would be necessary to provide shared variables allocation functions on the PPE, and a mean for the SPE to retrieve those variables address. However in a non-virtual implementation, as physical address as used directly (there is no need to use the TLB), address could be hard encoded in the SPE executables.

7. Conclusion

To conclude, an efficient real-time system can be implemented on the Cell Broadband Engine. Through Virt-K, it has been proven that running and synchronizing multiple tasks on the SPE is possible, and that it is mostly predictable.

However, Virt-K failed in being completely predictable and therefore did not answer the hypothesis. Due to the unbounded execution time of semaphore acquisition, Virt-K, and the design it is implementing, do not fit the requirements for a hard real-time kernel. Enhancements have been proposed to bind the upper execution time; but these improvements either sacrifice performances for predictability, or provide an extremely wide bound, not significant in most of the cases.

It seems that shared-memory semaphore approach is not adapted to the Cell architecture. Further research should look at an implementation of message passing semaphores on the Cell.

An important point is that though the Cell can run soft real-time kernels and probably hard real-time kernels too, it appears that its purpose is not for pure real-time systems. Even in achieving extremely high performances by running eight real-time processors in a single chip, it is only using a small part of its possibilities.

An interesting approach, which seems popular among the Cell community, is to run both Linux and a real-time kernel on the Cell. As in Linux RT, a real-time kernel would be running on the PPE and reserve a number of SPE. A regular Linux kernel would be then running as the lowest priority task in the real-time kernel and could use the unreserved SPE.

The ideas proposed in this dissertation would fit such an approach though a scheduler would have to be added on the PPE to allow Linux to run.

For a simple soft real-time system, an implementation like Virt-K would be sufficient. Running the process within Linux kernel with a high priority would ensure real-time response time (except for few system calls). Moreover, Linux kernel not interfering with the SPE execution, allows real-time capabilities too.

To finish, the Cell has real-time capabilities but is much more than a real-time processor and to fully use its potential, mixed solutions should be considered.

8. BIBLIOGRAPHY

[1] T. E. Anderson, "Performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, pp. 6-16, January 1992.

[2] T. E. Anderson, D. D. Lazowska, and H. M. Levy, "The performance implications of thread management alternatives for shared-memory multiprocessors," in *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Oakland, California, 1989, pp. 49-60.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the 10th annual ACM symposium on Parallel algorithms and architectures*, Puerto Vallarta, Mexico, 1998, pp. 119-129.

[4] R. Bealkowski and E. B. Fernandez, "A heterogeneous multiprocessor architecture for workstations," in *Proceedings of IEEE Southeastcon '91*, Williamsburg, Virginia, 1991, pp. 258-262.

[5] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems,* vol. 2, pp. 39-59, 1984.

[6] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parralel programming*, San Jose, California, 2007, pp. 90-100.

[7] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 1st edn. ed. Farnham: O'Reilly, 2000.

[8] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier, "A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility," in *Proceedings of EurOpen Spring 1991 Conference*, Tromso, Norway, 1991, pp. 13-32.

[9] P. Brucker, *Scheduling Algorithms*, 2nd ed. Berlin: Springer, 1999.

[10] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Transactions on Computer Systems,* vol. 15, pp. 412 - 447, November 1997.

[11] R. J. A. Buhr and D. L. Bailey, *An introduction to real-time systems : from design to multitasking with C/C++*. Upper Saddle River: Prentice Hall, 1999.

[12] A. Burns, *Real-time systems and programming languages: Ada 95,real-time Java and Real-time POSIX*, 3rd edn. ed. Harlow: Addison-Wesley, 2001.

[13] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Techniques for reducing consistency-related communication in distributed shared-memory systems," *ACM Transactions on Computer Systems,* vol. 13, pp. 205-243, August 1995.

[14] CellMicroprocessor, "Cell Microprocessor", 2007.
http://en.wikipedia.org/wiki/Cell_microprocessor

[15] J. Chen and J.-H. Liu, "Developing embedded kernel for system-on-a-chip platform of heterogeneous multiprocessor architecture," in *Proceedings of the12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, 2006, pp. 246-250.
[16] D. R. Cheriton, "The design of a distributed kernel," in *Proceedings of the ACM '81 conference*, 1986, pp. 46-52.
[17] S. P. Dandamudi and P. S. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, pp. 1-16, Jan 1995.
[18] E. W. Dijkstra, "The structure of the “the”-multiprogramming system," in *Proceedings of the 1st ACM symposium on Operating System Principles* 1967, pp. 10.1-10.6.
[19] B. P. Douglass, *Real Time UML : advances in the UML for real-time systems*, 3rd edn. ed.: Addison-Wesley, 2004.
[20] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, vol. 21, pp. 9-21, 1988.
[21] I. Englander, *The architecture of computer hardware and systems software : an information technology approach*, 3rd edn. ed. Chichester: Wiley, 2003.
[22] R. Finkel and D. Hengsen, "YACKOS on a shared-memory multiprocessor," *ACM SIGARCH Computer Architecture News*, vol. 16, pp. 31-36, 1988.
[23] B. Frey, *PowerPC Operating Environment Architecture*, 2003.
http://www.ibm.com/developerworks/power/library/pa-archguidev1
[24] B. Frey, *PowerPC User Instruction Set Architecture*, 2003.
http://www.ibm.com/developerworks/power/library/pa-archguidev1
[25] B. Frey, *PowerPC Virtual Environment Architecture*, 2003.
http://www.ibm.com/developerworks/power/library/pa-archguidev1
[26] A. Gheith and K. Schwan, "CHAOSarc: kernel support for multiweight objects, invocations, and atomicity in real-time multiprocessor applications," *ACM Transactions on Computer Systems*, vol. 11, pp. 33-72, 1993.
[27] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, pp. 10-24, 2006.
[28] P. Hofstee, "Power Efficient Processor Architecture and The Cell processor," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, San Francisco, California, 2005, pp. 258-262.
[29] IBM, *Cell Broadband Engine Programming Handbook*, 2006.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F
[30] IBM, *Celll Broadband Engine Programming Tutorial*, 2006.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/FC857AE550F7EB83872571A80061F788
[31] IBM, *IBM Full-System Simulator User's Guide*, 2006.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/B494BF3165274F67002573530070049B

[32] IBM, *Cell Broadband Engine Architecture*, 2007.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA
[33] IBM, *Cell Broadband Engine Registers*, 2007.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/6ED822DD7E97D889872570B200607EEC
[34] IBM, *SPU Instruction Set Architecture*, 2007.
http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44
[35] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, pp. 589-604, July/September 2005.
[36] A. Khemka and R. K. Shyamasundar, "Multiprocessor scheduling of periodic tasks in a hard real-time environment," in *Proceedings of the 6th International Parallel Processing Symposium*, Beverly Hills, California, 1992, pp. 76-81.
[37] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, pp. 10-23, 2006.
[38] H. Kopetz, *Real-time systems : design principles for distributed embedded applications*. Boston; London: Kluwer Academic, 1997.
[39] A. Kumar, N. Jayam, A. Srinivasan, G. Senthilkumar, P. K. Baruah, S. Kapoor, M. Krishna, and R. Sarma, "Feasibility study of MPI implementation on the heterogeneous multi-core cell BE architecture," in *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architecture*, San Diego, California, 2007, pp. 55-56.
[40] B. L. Kurtz and J. J. Pfeiffer, "A Course project to design and implement the kernel of a real-time operating system," in *Proceedings of the 18th SIGCSE technical symposium on Computer science education table of contents*, St. Louis, Missouri, 1987, pp. 115-119.
[41] K. Kwangsik, K. Dohun, and P. Chanik, "Real-time scheduling in heterogeneous dual-core architectures," in *Proceedings of the 12th International Conference on Parallel and Distributed Systems* Minneapolis, Minnesota, 2006, pp. 91-96.
[42] J. J. Labrosse, *MicroC/OS-II : the real-time kernel*, 2nd edn. ed. Great britain: CMP Books, 2002.
[43] K. Langendoen, R. Bhoedjang, and H. Bal, "Models for Asynchronous Message Handling," *IEEE Concurrency*, vol. 3, pp. 28-38, Apr-Jun 1997.
[44] B. Lee, "Optimizing heterogeneous architecture," *EDN*, vol. 51, pp. 65-69, 2006.
[45] LeProcesseurCell, "Le Processeur Cell", 2005.
http://www.presence-pc.com/tests/Le-processeur-Cell-366/
[46] J. Levesque, J. Larkin, M. Foster, J. Glenski, G. Geissler, S. Whalen, B. Waldecker, J. Carter, D. Skinner, H. He, H. Wasserman, J. Shalf, H. Shan, and E. Strohmaier, "Understanding and Mitigating Multicore Performance Issues on the AMD Opteron Architecture," *Lawrence Berkeley National Laboratory*, pp. Paper LBNL-62500, 2007.

[47] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM symposium on Operating systems principles*, Copper Mountain, Colorado, 1995, pp. 237-250.

[48] J. Liedtke, "Toward real microkernels," *Communications of the ACM,* vol. 39, pp. 70-77, 1996.

[49] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM,* vol. 20, pp. 46-61, January 1973.

[50] V. B. Lortz and K. G. Shin, "Semaphore Queue Priority Assignment for Real-Time Multiprocessor Synchronization," *IEEE Transactions on Software Engineering,* vol. 21, pp. 834-844, 1995.

[51] M. Maniecki, "Universal Real-Time kernel," *Microprocessing and Microprogramming,* vol. 14, pp. 161-163, 1984.

[52] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors," *ACM Transaction on Computer Systems,* vol. 11, pp. 146-178, 1993.

[53] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems,* vol. 9, pp. 21-65, 1991.

[54] L. Molesky, K. Ramaritham, C. Shen, J. Stankovic, and G. Zlokapa, "Implementing a predictable real-time multiprocessor kernel - the Spring kernel," in *Proceedings of the 7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, 1990, pp. 20-26.

[55] O. Moreira, F. Valente, and M. Bekooij, "Scheduling Multiple Independent Hard-Real-Time Jobs on a Heterogeneous Multiprocessor," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, Salzburg, Austria, 2007, pp. 57-66.

[56] N. Morey-Chaisemartin, "Sourceforge project : Virt-K", 2008.
http://sourceforge.net/projects/virt-k/

[57] A. Morton and W. M. Loucks, "A hardware/software kernel for system on chip designs," in *Proceedings of the 2004 ACM symposium on Applied computing*, Nicosia, Cyprus, 2004, pp. 869-875.

[58] H. Muta, M. Doi, H. Nakano, and Y. Mori, "Multilevel parallelization on the cell/B.E. for a motion JPEG 2000 encoding server," in *Proceedings of the 15th international conference on Multimedia*, Augsburg, Germany, 2007, pp. 942-951.

[59] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Amora, and S. Kesavarapu, "Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers," *IBM Journal of Research and Development,* vol. 51, pp. 573-582, September 2007.

[60] B. Parhami, *Computer architecture : from microprocessors to supercomputers*. New York; Oxford: Oxford University Press, 2005.

[61] R. Rajkumar, L. Sha, and J. P. Lehocczky, "Real-Time Synchronization Protocols for Multiprocessors
" in *Proceedings of the Real-Time Systems Symposium*, Huntsville, AL, USA, 1988, pp. 259-269.

[62] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 1, pp. 184-194, 1990.

[63] M. W. Riley, J. D. Warnock, and D. F. Wendel, "Cell Broadband Engine processor: Design and implementation," *IBM Journal of Research and Development*, vol. 51, pp. 545-557, Sept 2007.

[64] Z. Salcic, P. Roop, D. Hui, and I. Radojevic, "HiDRA: A new architecture for heterogeneous embedded systems," in *Proceedings of the International Conference on Embedded Systems and Applications*, Las Vegas, Nevada, 2004, pp. 164-170.

[65] T. Samuelsson, M. Akerholm, J. Starner, and L. Lindh, "A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software," in *International Workshop on Advanced Real-Time Operating System Services*, Porto, Portugal, 2003.

[66] L. Shal, R. Rajkumar, and J. P. Lehocczky, "Priority Inheritance Procotols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, pp. 1175-1185, 1990.

[67] K. Shimizu, S. Nusser, W. Plouffe, V. Zbarsky, M. Sakamoto, and M. Murase, "Cell broadband Engine processor security architecture and digital contant protection," in *Proceedings of the 4th ACM international workshop on Contants protection and security*, Santa Barbara, California, 2006, pp. 13-18.

[68] A. l. Shimpi, "Understanding the Cell Microprocessor," in *Anandtech*, 2005.

[69] K. G. Shin and C.-J. Hou, "Design and Evaluation of Effective Load Sharing in Distributed Real-Time Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 704-719, 1994.

[70] S. G. Shiva, *Advanced computer architectures*. London: CRC/Taylor & Francis, 2006.

[71] M. Singhal, *Advanced concepts in operating systems : distributed, database, and multiprocessor*. New York; London: McGraw-Hill, 1994.

[72] M. S. Squillante, "Issues in shared memory multiprocessor scheduling: a performance evaluation," in *Department pf Computing Science and Engineering*. vol. Ph.D. Washington: University of Washington, 1990.

[73] W. Stallings, *Operating Systems: Internals and Design Principles*, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2004.

[74] J. A. Stankovic and K. Ramamritham, "The Spring kernel: a new paradigm for real-time operating systems," *ACM SIGOPS Operating Systems Review*, vol. 23, pp. 54-71, 1989.

[75] P. Stenstrom, E. Hagersten, D. J. Lilja, M. Martonosi, and M. Venugopal, "Trends in shared memory multiprocessing," *Computer*, vol. 30, pp. 44-50, December 1997.

[76] A. Suksompong, "Real-TIme Systems on Multicore Platforms " in *Department of Computer Sciend and Electronics*. vol. Ph. D. Eskilstuna: Mälardalen University, 2007.

[77] A. S. Tanenbaum, *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice Hall International, 1995.

[78] A. S. Tanenbaum and S. Andrew, *Modern operating systems*, 2nd edn. ed. Upper Saddle River, NJ: Prentice Hall International, 2001.

[79] A. S. Tanenbaum and L. Torvald, "LINUX is obsolete", 1992. http://groups.google.com/group/comp.os.minix/browse_thread/thread/c25870d7a41696d2/f447530d082cd95d?tvc=2

[80] TheCellRoadmap, "The Cell Roadmap", 2005. http://www.ppcnux.com/modules.php?name=News&file=article&sid=6666

[81] H. Thielemans, L. Demeestere, and H. Van Brussel, "HEDRA: heterogeneous distributed real-time architecture," *Control Engineering Practice,* vol. 4, pp. 187-193, 1996.

[82] A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," in *Proceedings of the 12th ACM symposium on Operating systems principles*, Litchfield Park, Arizona, 1989, pp. 159-166.

[83] R. Vaswani and J. Zahorjan, "The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors " in *Proceedings of the 13th ACM symposium on Operating systems principles*, Pacific Grove, California, 1991, pp. 26-40.

[84] W. Walker and H. G. Cragon, "Interrupt processing in concurrent processors," *Computer,* vol. 28, pp. 36-46, June 1995.

[85] B. Wilkinson, *Computer architecture : design and performance*, 2nd ed. London: Prentice Hall, 1996.

[86] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the 3rd conference on Computing frontiers*, Ischia, Italy, 2006, pp. 9-20.

[87] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: the kernel of a multiprocessor operating system," *Communications of the ACM,* vol. 17, pp. 337-345, June 1974.

[88] W. Wulf and S. McKee, "Hitting the memory Wall," *ACM Computer Architecture News,* vol. 23, pp. 20-24, 1995.

[89] J. Zahorjan, E. D. Lazowska, and D. L. Eager, "The effect of scheduling discipline on spin overhead in shared memory parallel systems," *IEEE Transactions on Parallel and Distributed Systems,* vol. 2, pp. 180-198, April 1991.

[90] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, "Software Write Detection for a Distributed Shared Memory," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, Monterey, California, 1994, pp. 87-100.

[91] S. Zhou, M. Stumm, K. Li, and D. Wortman, "Heterogenous Distributed Shared-Memory," *IEEE Transactions on Parallel and Distributed Systems,* vol. 3, pp. 540-554, 1992.

[92] K. M. Zuberi, P. Pillai, and K. G. Shin, "EMERALDS: a small-memory real-time microkernel " in *Proceedings of the 17th ACM symposium on Operating systems principles*, Kiawah Island, South Carolina, 1999, pp. 277-299.

9. Appendix A – Glossary

Cell	Short name for the Cell Broadband Engine.
DMA	Direct Memory Access. DMA transfers are transferred that do not involve the CPU once they are started.
DPSRA	Direct Problem State Register Access. Stands for access to the SPE registers by the PPE through DMA transfers.
EIB	Element Interconnect Bus. Connect all the elements of the Cell Broadband Engine.
FlexIO	Input/output interface of the Cell Broadband Engine.
Itanium	Intel architecture used for high computational power servers or clusters.
LS	Local Store. It is a 256KB memory owned by each SPE.
MFC	Memory Flow Controller. Manage all memory transfer between the LS and other LS or main memory.
MPI	Message Passing Interface. It is an efficient communication protocol often used in clusters.
PC	Program Counter. Address of the instruction currently being executed
SMP	Symmetrical Multi Processing. Common approach to multi-processing where a copy the kernel is executed on each processor
SP	Stack Pointer.
SPE	Synergistic Processor Elements.
PPE	PowerPC Processor Element.
X86	32bit processor architecture introduced by Intel and used in most of the personal computers.
X86_64	64bits architecture. Successor of the x86 architecture.
XDR	Extreme Data Rate. XDR is a type of Random Access Memory.
Virt-K	Virt-K is the implementation developed during this project.

10. Appendix B – Terms of reference

10.1 Background Information

In the late years, microprocessor founders have had trouble following the Moore Law. Not only they have had they to face physical problems due to the size of transistors but also what is called the “Memory Wall” [88] due to the memory latency and cache misses. Numerous new ideas have been tried to solve this problem but only few have. Classical multicore architectures, like Intel Core Duo, have managed to achieve greater performances but memory accesses are being more critical than ever.

However as described by [35], STI, the union of Sony, Toshiba and IBM, have worked since 2000 on this problems and have found a working solution: the Cell Broadband Engine, commonly called Cell. The results of the Cell will not be discussed here but they can be found in [86].

The Cell thus has many interesting aspects. It has been made as a replacement for personal computers architectures (x86, x86_64) but is also a powerful parallel calculator. It is being currently used in the Playstation 3, but also in multiprocessors computers and soon in clusters.

Parallel computing has been a research subject for many years now, but its application to real-time systems has always been problematic as many processors often imply large power consumption. The Cell may change this as IBM clearly indicates in its official articles, [35] among others, “The Cell processor should provide extensive real-time support”. Further references to real-time support can also be found in the technical documentation of the Cell (i.e. [29]). STI has also announced, according to [80], a lighter version of the Cell for embedded hardware and real-time usage

The few research done on real-time for the Cell have been on efficient computing algorithm as MPEG compression, ray tracing, terrain rendering, but none of them has clearly answered the question of the feasibility of a real-time operating system for the Cell.

10.2 Project Outline

During this project, it will be tried to answer the question of the feasibility of a real-time kernel for the Cell, which is a heterogeneous multicore processor. More than proposing an implementation, further performances tests will be used to judge about the Cell's efficiency as a real-time processor.

Our research hypothesis will be:

"*Can an effective hard real-time scheduler be implemented for the Cell Broadband Engine?*"

The main criteria of a hard real-time scheduler, which will be used to test its effectiveness, are:

- It has to allow task creation.
- It must be able to schedule task for concurrent processing.
- Its performances should be predictable and bounded.
- It should provide primitives for safe concurrent processing (mainly synchronisation primitives)

It would be really difficult, even impossible to write a complete kernel within the project deadlines. It would be possible to use an existing kernel, and adapt it to the Cell, but it would require a lot of time to port and most of its code would have to be rewritten to fit the requirements. Therefore, Linux, which has already been ported on the Cell, will be used as a development platform, to run a "virtual kernel". Linux drivers and few low-level functions will be used to simplify the development process and be able to focus on scheduling and synchronisation. The main problem of this approach is that Linux kernel is not real-time. Some part of our virtual kernel will not thus real-time but results might still be available.

The kernel that will be developed will focus on achieving high performances (i.e. better than Linux and existing real-time kernel on other architectures) using new scheduling and synchronisation functions. A hard real-time scheduler will be implemented, and be provided semaphores. Further scheduling algorithms and synchronisation means (mutexes, conditions) may be discussed in the report.

The implementation and its effectiveness will be tested through simulation by running case studies. It will be judged effective if the case study run successfully and the achieved latencies are acceptable for a real-time kernel.

To fully understand the implication of this project, a bit of background on the Cell processor and real-time kernel is needed.

10.3 Review of the Cell Broadband Engine

10.3.1 Heterogeneous multicore architecture

Heterogeneous multicore processors are quite new on the processor markets. Before them, except a few exceptions, only the regular single core processor, the homogeneous multicore processors like the AMD Opteron [46] and heterogeneous multi processors were available.

Multi-processors (and parallel systems) have played quite a big role in the past. Even with the computing power of a processor following the Moore Law or so, the need for power will always exists. Even with the last generation of processors being thousands times more powerful than their predecessors 20 or 30 years ago, they still do not fit every purpose. Modelising climate or physicals phenomena requires much more power than a processor can bring. Parallel processing was and still is the only way to reach such computing power requirements. Moreover, as described in [77], parallel architectures have many advantages as reliability, low cost (compare to an eventual equivalent single processor.

Some of these advantages have been lost when moving to multi-processor parallel architectures to multi-core architectures, but the computation power and reliability are still there.

As said earlier, parallel architectures have been a research subject for many years. Many results may be used in this project as multi-core processors can be seen as a multi-processor parallel architecture on-a-chip.

10.3.2 The Cell Broadband Engine Architecture

The Cell, as described by [32], consists of 9 cores on a single chip. One of them is the PPE (PowerPC Processor Element), and the other eight are advanced computational units called SPE (Synergistic Processor Units). All these processors are linked by a high data-rate bus called EIB (Element Interconnect Bus). The SPEs and the EIB are the keys to the Cell success as they provide a considerable amount of computational power without neglecting the memory wall problem.

10.3.2.1 PowerPC Processor Element (PPE)

According to [29], the PPE being the central element of the Cell is in fact an enhanced PowerPC 64bits core. Thus, using well known technology, STI has guaranteed some compatibility with existing softwares.

In the current Linux version running on the Cell, the PPE is the only core used by default. However softwares can choose to send tasks to the SPEs. The problem with this solution is that, although all the previous PowerPC software can be executed without any changes, a lot of the SPE computation time is wasted as few software are using them.

The main role of the PPE, fixed by the design [32], is to host the operating systems as it has full access to all hardware (Interrupt Controller, EIB Management...).

10.3.2.2 Synergistic Processor Elements (SPE)

The SPE is a highly advanced 64 bits computational unit using SIMD (Single Instruction Multiple Data).

The main difference between PPE and SPE, more than the restricted instruction set of the SPE, is the way they access memory [27]. Where the PPE uses the same standard access (cache L1 and L2), the SPE has its own local memory. As [29]indicates, each SPE owns a 256 kilobytes Local Store.

Another critical difference is that a PPE accesses the memory “on the fly” which means the data are retrieved from memory to the cache at the moment they are needed. However, the SPE owns a MFC (Memory Flow Controller) which can retrieve data asynchronously. The MFC is given a list of memory area to load or store and will do it as soon as the hardware makes it possible. Therefore, the SPE can pre-fetch the data it’ll need before it actually needs it. This may seems really simple, but it is a solution to the memory wall as mentioned earlier. The memory latency is not a real problem anymore as the SPE does not have to wait for its data to keep running. To be more precise, data being prefetch, the program will not encounter cache misses, and then run faster. In the case of a single process running on a SPE (isolated and non-pre-empted), it is possible to calculate exactly response time within the SPE. As every instruction as a know execution time and so does the Local Store access, the execution time of a function can be calculated at a SPE cycle precision.

Moreover, it is fully possible to imagine a pre-allocation of data in the Local Store having a task ready to run has soon as a signal (intern or external event) arrives, avoiding a costly context switch. By sharing the Local Store, several processes could be stored at the same time. Context switch in this case would only require saving the different register.

For a soft real-time usage, the same principle could be applied adding paging functions to send back some processes to the main memory if needed. Context switch would still be less costly than copying the full Local Store to the main memory and the average response time would stay quite low.

10.3.2.3 Current Researches

The Cell and its new architecture, first produced in 2005, is still a subject of research.

More than looking for efficient algorithm using its full potential as [37],[58] and [6], many laboratories are working on possible implementations and how to enhance the current Linux for Cell.

[39]have worked on implementing MPI (Message Passing Interface) on the Cell. MPI is necessary to work efficiently on multiprocessor Cell computers. They have implement blocking point-to-point communications on the current Linux for that. But more than that, the techniques they used are interesting for SPE synchronisation (mutexes semaphores). It will be discussed in more details later in this review.

10.4 Real-Time Operating Systems

What real-time operating system is will not be discussed here; instead we will focus on an efficient implementation of it on the Cell

The concept of heterogeneous multicore is new, so there have not been many research done on it. However, even if all the cores are in a single chip, the Cell can be seen has a heterogeneous multiprocessor, thus, giving access to many results from completed researches. Most of them will not be cited here as the ideas they bring are really technical and will not be used before the programming stage.

According to [78] there are only three ways to modelise multiprocessor systems: shared-memory multiprocessors, message-passing multicomputer and wide area distributed systems. The third one cannot be considered for the Cell as it is a single chip; however the two other approaches are valid.

Although[15] and their AsymOS operating system have proven it is possible and efficient to run different part of the operating system on each core, using MPI, it is not appropriate for the Cell. The SPE have not been designed to run an operating system due to their limited access to the hardware and the high-delay context changes.

To consider the Cell as a shared-memory multiprocessor seems to be the most efficient approach.

10.4.1 Kernel

As it was said earlier, SPEs are not supposed to run any part of the operating systems, including the kernel, hence the kernel has then to run entirely on the PPE. It seems also acceptable, in certain condition, to run a micro kernel on a SPE to manage local scheduling and synchronisation. It would increase context switches but decrease substantially communication between the PPE and SPE which may prove useful in specific applications.

A problem Linux faces is the lack of usage of the SPE, as they are only used by Cell specific application through threads, the father process always running on the SPE. In this case, to maximise the Cell usage, the SPEs may be used fully.

An interesting approach is to run a real on the PPE, and all the time-constrained software on the SPEs. Some unconstrained software could also be run on the PPE if needed.

The main problem of this approach is the synchronisations and the access to Operating System resources. However it allows the kernel to run power consuming scheduling algorithms as most of the PPE computing time is reserved to the kernel. Such optimized algorithm may not be proven useful to a hard real-time kernel where a pre-emptive round-robin may be preferred. But for soft real-time, power-consuming algorithm, it may improve the overall performances.

10.4.1.1 Synchronisation

As described by [38], in any multi-tasked OS, it is necessary for the kernel to provide synchronisation means, so that the different softwares executing at the same time can exchange information and be prevented from writing a shared memory area at the same time.

These functions are usually operated by the kernel using specific CPU instructions [78]. The Cell provides such function support but only on the PPE.

In the current Linux implementation, the SPEs execute each of these functions by sending system calls to the PPE which do the real treatment. This solution has drawbacks, as it needs the PPE execution to be interrupted each time one of the eight PPE needs access to these functions.

It is important to understand that synchronisation is critical in a multicore system and that every single micro-second gained will have a huge impact on the overall performances.

The work of [39] mentioned earlier gives new solutions to these problems. In specific case, blocking point-to-point communications between SPEs can be used. It will then only uses CPU time of the concerned SPE and achieve higher performances.

[1], and many others, propose other efficient techniques for locking access to part of the memory. Some of them may be used to enhance the previous algorithm.

Some research will be done on the Atomic Cache Unit of the SPE which provide an atomic way to write in the memory and be instantly shared with the other SPE.

In this kernel, we will only implement semaphores as they also cover mutexes. However, with a few tweaks to scheduling, the same algorithms could be used for conditions.

10.4.1.2 Operating System Resources

More than synchronisation, processes need access to resources: mainly access to the peripherals and their interrupts, timers, etc.

Linux here once again uses system calls to have the PPE doing the work. And once again, it is working but not in the most efficient way.

According to [32] indicates, the Cell architecture does not provide hardware support to route interruptions to a SPE directly. Therefore it is necessary that the PPE deals with them. However, the PPE does not have to treat more than the interrupts. The Cell integrates communication channels between the SPE and the PPEs and can thus “forward” the interrupt to the concerned SPE. It is important that only one SPE is linked to an interrupt as a peripheral cannot be accessed by more than one core at the same time.

This system allows deferring the peripheral treatment only to the concerned core, leaving the other unchanged and being more efficient. This is only possible when one single core needs access to the peripheral. If the peripheral is shared, the PPE would have to do the treatment as Linux does already.

In this kernel, we will not implement a real-time access to resources. Linux functions and drivers will be used to access the peripherals. Therefore, the kernel I/O will not be real-time.

10.4.2 Real-time Scheduling

Scheduling is the most critical part of an operating system and even more critical in the case of a real-time one. The scheduler has to choose an execution order of the processes depending on their priority, and their deadlines.

Scheduling on a single CPU has now been used for years and there are many optimal algorithms like the round-robin described by [42]. However, scheduling on a multi-processor is a much difficult problems.

Most of the multi-processor (or multicore) systems like the one used by [15] uses Symmetrical Multi-Processing (SMP) kernels which means each processor has its own kernel and each of them does it own scheduling, falling back to the single CPU scheduling case.

As [83] described, scheduling on shared-memory multiprocessor implies taking in account the cache (here the local stores), as it is really time-consuming to reload all the data when a processes is moved to another SPE.

[62] propose an interesting approach of scheduling: instead of taking only processor requirements into consideration, they consider the whole resource consumption (CPU, peripherals, mutexes). They provide a heuristic algorithm which can dynamically schedule a set of tasks. As said earlier, the software would run only on SPE and the kernel on the PPE. This algorithm is thus really interesting as the PPE has unused computation power which can be used for an efficient scheduling. As said earlier, such algorithms may not prove useful in a hard real-time kernel; therefore they will not be implemented in this kernel. However they may be discussed as an improvement for a more generic purpose of the Cell.

10.5 Research areas

- Heterogeneous Multicore Architectures
- Distributed Operating Systems
- Scheduling for Multicore Processors
- Shared-memory Multiprocessor Synchronisation

10.6 Aims

Implementing a hard real-time kernel for the Cell Broadband Engine

10.7 Objectives

- Research and discuss Linux implementation on SPEs.
- Specify and develop a virtual kernel using Linux as a non real-time hypervisor.
- Research, design and implement a scheduling algorithm to manage the SPEs
- Research, design and implement semaphores for the SPEs
- Test system performances and compare to existing real-time kernels.
- Research and discuss possible algorithms to achieve higher performances and more generic purpose.

10.8 Relationship to course

This project covers the major area taught in the courses. It requires a good understanding of real-time kernel as well as concurrent programming.

It will involve conception, programming in C and Assembler, scheduling analysis and much more different skills acquired during the two taught semesters.

10.9 Resources constraints

As the Cell is not accessible directly, the whole project will be done with a free simulator provided by IBM. The development and testing of the software will be realised on a Linux workstation running Fedora Core 7.

Due to the power requirements of the simulator, the work will be realised on a rack server I own. Thus, it may prove difficult to show a live demonstration of my kernel during the viva. A solution will be searched later, in agreements with both markers.

10.10 Proposed Outline of Dissertation

1. Introduction
2. Subject overview
    - 2.1. Parallel computers
    - 2.2. Cell Broadband Engine
    - 2.3. Real-Time Kernel
3. Research and design: Theoretical implementation
    - 3.1. Interfacing with Linux
    - 3.2. Task allocation
    - 3.3. Scheduling
    - 3.4. Semaphores
4. Results of the implementation
5. Performances
        - 5.1.1. Results
        - 5.1.2. Analysis of weak points
        - 5.1.3. Comparison with Linux
6. Discussion of possible enhancements
        - 6.1.1. Achieving higher performances
        - 6.1.2. Achieving a more general purpose
7. Possible usage
8. Conclusion

Note: Sub points have been used in this outline to regroup points matching a same topic. However, depending on the project, some of them may become proper chapters.

10.11 Schedule of activities

Task#	Task	Duration	Beginning	End	Predecessors
1	**Get ready for development**	**6 days**	**19/05/08**	**26/05/08**	
2	Install Fedora core 7, Cell SDK, Development tools and Simulator	2 days	23/05/08	26/05/08	
3	Read IBM Documentation about the Cell	4 days	19/05/08	22/05/08	
4	**Interfacing with Linux**	**8 days**	**27/05/08**	**05/06/08**	**1**
5	Analyse Linux implementation on the SPEs	3 days	03/06/08	05/06/08	
6	Analyse how to implement the virtual kernel within Linux	5 days	27/05/08	02/06/08	
7	**Virtual kernel structures**	**10 days**	**06/06/08**	**19/06/08**	**4**
8	Research and design the kernel structures	7 days	06/06/08	16/06/08	
9	Implement the basic structures	3 days	17/06/08	19/06/08	8
10	**Scheduling**	**15 days**	**20/06/08**	**10/07/08**	**7**
11	Research and design task allocation and scheduling algorithms	10 days	20/06/08	03/07/08	
12	Implement scheduling	5 days	04/07/08	10/07/08	11
13	**Synchronisation**	**15 days**	**11/07/08**	**31/07/08**	**10**
14	Research and design semaphore algorithm	5 days	11/07/08	17/07/08	
15	Implement semaphores	10 days	18/07/08	31/07/08	14
16	**Tests (includes program writing)**	**5 days**	**01/08/08**	**07/08/08**	
17	Test SPE Scheduling	3 days	05/08/08	07/08/08	12
18	Test SPE synchronisation	2 days	01/08/08	04/08/08	15
19	**Performance Tests**	**7 days**	**08/08/08**	**18/08/08**	**13;16**
20	Run performance tests	3 days	08/08/08	12/08/08	
21	**Analyse test**	**4 days**	**13/08/08**	**18/08/08**	**20**
22	*Find weaknesses*	2 days	13/08/08	14/08/08	
23	*Compare to Linux implementation*	2 days	15/08/08	18/08/08	
24	**Overall analysis**	**10 days**	**19/08/08**	**01/09/08**	**19**
25	Analyse Cell potential for real-time application	3 days	19/08/08	21/08/08	
26	Discuss possible usage of the Cell	2 days	22/08/08	25/08/08	25
27	Research and discuss possible enhancements	5 days	26/08/08	01/09/08	
28					
29	**Write report**	**76 days**	**19/05/08**	**01/09/08**	