CHEMNITZ UNIVERSITY OF TECHNOLOGY

Department of Computer Science

Chair of Computer Architecture

Diploma Thesis

Execution of SPE code in an Opteron-Cell/B.E. hybrid system

Andreas Heinig

Chemnitz, March 11, 2008

Supervisor: Prof. Dr. Wolfgang Rehm
Advisor: Dipl. Inf. Torsten Mehlan

Theses

1. It is possible to establish a heterogeneous Opteron-Cell system across TCP/IP.
2. With PCIe a much better performance could be expected.
3. The SPUFS concepts are transferable to RSPUFS.
4. The concepts may could be generalized to support also other kinds of accelerators.

Abstract

It is a great research interest to integrate the Cell/B.E. processor into an AMD Opteron system. The result is a system benefiting from the advantages of both processors: the high computational power of the Cell/B.E. and the high I/O throughput of the Opteron.
The task of this diploma thesis is to accomplish, that Cell-SPU code initially residing on the Opteron could be executed on the Cell under the GNU/Linux operating system. However, the SPUFS (Synergistic Processing Unit File System), provided from STI (Sony, Toshiba, IBM), does exactly the same thing on the Cell. The Cell is a combination of a PowerPC core and Synergistic Processing elements (SPE). The main work is to analyze the SPUFS and migrate it to the Opteron System.
The result of the migration is a project called RSPUFS (Remote Synergistic Processing Unit File System), which provides nearly the same interface as SPUFS on the Cell side. The differences are caused by the TCP/IP link between Opteron and Cell, where no Remote Direct Memory Access (RDMA) is available. So it is not possible to write synchronously to the local store of the SPEs. The synchronization occurs implicitly before executing the Cell-SPU code. But not only semantics have changed: to access the XDR memory RSPUFS extends SPUFS with a special XDR interface, where the application can map the XDR into the local address space. The application must be aware of synchronization with an explicit call of the provided "xdr_sync" routine. Another difference is, that RSPUFS does not support the gang principle of SPUFS, which is necessary to set the affinity between the SPEs.
This thesis deals not only with the operating system part, but also with a library called "libspe". Libspe provides a wrapper around the SPUFS system calls. It is essential to port this library to the Opteron, because most of the Cell applications use it. Libspe is not only a wrapper, it saves a lot of work for the developer as well, like loading the Cell-SPU code or managing the context and system calls initiated by the SPE. Thus it has to be ported, too.

The result of the work is, that an application can link against the modified libspe on the Opteron gaining direct access to the Synergistic Processor Elements.

Contents

List of Figures v

List of Tables vii

1 Introduction 1

2 Opteron Cell/B.E. Hybrid System 3

2.1 Cell Broadband Engine Architecture . . . 3

2.1.1 Targets and Challenges . . . 3

2.1.2 Architecture . . . 5

2.1.3 Cell Multiprocessor Systems . . . 9

2.1.4 Hardware . . . 9

2.2 Heterogeneous Multiprocessing in an Opteron-Cell Environment . . . 11

2.2.1 Interconnections . . . 11

2.2.2 Development Environment . . . 12

2.2.3 PCIe Coupling . . . 13

2.2.4 Opteron Cell evaluation Platform . . . 14

3 SPUFS 15

3.1 Introduction . . . 15

3.2 The SPUFS Concept . . . 17

3.3 Internal Context . . . 19

3.4 User-Level Interfaces . . . 20

3.4.1 System Programing Interface . . . 21

3.4.2 SPE Runtime Management Library . . . 24

3.4.3 Toolchain . . . 27

3.5 Externally Assisted SPE Library Calls . . . 29

4 RSPUFS **32**
4.1 Targets . . . 32
4.2 Structure . . . 33
4.3 The Cell Part: *rspufsd* . . . 34
4.3.1 Network Protocol . . . 34
4.3.2 Services . . . 36
4.4 The Opteron Part: *rspufs* . . . 44
4.4.1 Internal Context . . . 45
4.4.2 Packet Dispatcher . . . 45
4.4.3 User-Level Interfaces . . . 46
System Programing Interface . . . 46
SPE Runtime Management Library . . . 48
Toolchain . . . 49
4.4.4 Software Emulated Remote Memory Access . . . 49

5 Results **51**
5.1 Implementation . . . 51
5.2 The SPUFS Concepts . . . 52
5.2.1 Virtual File System . . . 52
5.2.2 Virtualization . . . 52
5.3 First Benchmark Results . . . 53
5.3.1 Software Emulated Memory Access . . . 53
5.3.2 Adding Two Integers . . . 54

6 Conclusion and Further Work **57**

Bibliography **59**

A CD-ROM Contents **61**

List of Figures

2.1 Cell block diagram 6
2.2 Single Cell configuration 9
2.3 Cell dual processor system 9
2.4 Quad-Cell via Switch 10
2.5 Interconnect classification 11
2.6 Tightly coupled Opteron Cell evaluation platform 14

3.1 SPUFS inside the Linux Kernel 16
3.2 `struct spu_context` in linux-2.6.22 19
3.3 `struct spu_csa` in linux-2.6.22 20
3.4 The `spu_create` system call 21
3.5 The `spu_run` system call 24
3.6 SPUFS: Tool Chain 28
3.7 `struct spe_program_handle` 28
3.8 Parameter image for an assisted `write` call 29
3.9 PPE assisted `write` system call 31

4.1 RSPUFS Model 33
4.2 RSPUFS Protocol 35
4.3 RSPU_S_CREATE message 36
4.4 RSPU_S_FILELST message 37
4.5 RSPU_S_FOPEN message 38
4.6 RSPU_S_FRELEASE message 38
4.7 RSPU_S_FREAD message 39
4.8 RSPU_S_FPREAD message 39
4.9 RSPU_S_FWRITE message 40
4.10 RSPU_S_FPWRITE message 40
4.11 RSPU_S_FLLSEEK message 41

4.12 RSPU_S_SYSRUN message 41
4.13 RSPU_S_XALLOC message 42
4.14 RSPU_S_XFREE message 43
4.15 RSPU_S_XREAD message 43
4.16 RSPU_S_XWRITE message 44
4.17 The `spu_create` system call 45
4.18 RSPUFS: Tool Chain 49

5.1 Copy memory from Opteron to the Cell 53
5.2 Adding two numbers with mailbox parameter transmission 54
5.3 Adding two numbers with XDR parameter transmission 55

List of Tables

3.1 The SPUFS context (part) 22
3.2 Cell ABI stop-and-signal code definition 30

4.1 the RSPUFS context (part) 47

1 Introduction

To satisfy the needs of computing power and to place themselves at the top of the world market, the processor manufacturers developed processors with increasing clock rates. This results growing power consumptions. Thus the general trend in processor development leads to multi-core processors. That means, two or more standard processors are located on one chip (resp. die). Typical mainstream processors are the Intel Core 2 Duo/Quad or the AMD Opteron Processor. On such a multi-core the performance is not achieved by the clock ratio, but by the ability to execute threads or programs parallel.

However, the current research uses different types of cores on a single die. This is called "heterogeneous multi-core". The Cell processor is such a chip. In this diploma thesis heterogeneous has the following definition:

*The term **heterogeneous** or **hybrid** means a union from two or more different architectures. The union could be made on one chip or through an interconnect network.*

The intention of the Chair of Computer Architecture is to build a heterogeneous node for the HPC (High Performance Computing) environment. It consists of an Opteron and a Cell processor, which are tightly coupled. This hardware platform and the Cell Processor are described in chapter 2. This diploma thesis is an intermediate step towards this heterogeneous HPC node.

The objective is to write a driver implementing the direct access to the Synergistic Processing Units (SPUs) of the Cell in the GNU/Linux environment. SPUFS (Synergistic Processing Unit File System) is a Virtual File System, which implements this for the PowerPC-Linux running on the Cell. So the main work is to analyze

SPUFS (Chapter 3) and migrate it to the Opteron. The result of the migration is a project called RSPUFS (Remote Synergistic Processing Unit File System), which is introduced in Chapter 4.

The current implementation status and some benchmarks are shown in Chapter 5.

2 Opteron Cell/B.E. Hybrid System

This chapter introduces the underlying hardware for the Opteron Cell/B.E. Hybrid System, especially the relatively new Cell Broadband Engine Architecture in section 2.1. Thereafter the different hardware development stages are described, beginning with the current development platform for RSPUFS, the main part of this diploma thesis.

2.1 Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (Cell/B.E. or even Cell) is a corporate development from Sony, Toshiba and IBM ("STI"). The original intention was to develop a processor for the next generation of entertainment electronics, especially the successor of the Playstation 2 from Sony.

2.1.1 Targets and Challenges

The four developing objectives [1] for the Cell were:

- **High Performance**

  The performance of a system is limited by three barriers:

  1. Memory latency and bandwidth

     The latency and bandwidth to the main memory, especially dynamic random access memory (DRAM), is the greatest barrier for the performance. If the latency is measured in cycles, a typical single core processor will wait hundreds of cycles for a memory request. A symmetric multi processor (SMP) with shared memory could wait thousands of cycles. This phenomenon, also known as "Memory Wall"[2], implies that higher processor frequencies are not met by decreased DRAM latencies. Observations

proved this effect getting worse with each new processor generation.
If a microprocessor has a memory latency of 512 cycles and can fetch eight 128-byte cache-lines in flight, the maximum memory bandwidth is two bytes per processor cycle. This memory bandwidth is latency-induced.
However, the challenge was to find a processor organization allowing more simultaneous memory transactions to increase the memory bandwidth.

2. Power consumption

   The power consumption of processors grows more and more to a point where sophisticated cooling technologies are needed. But a system designed for consumers has some restrictions limiting the cooling capabilities for example the maximum air speed, the maximum box size and the maximum temperature of the air leaving the box.

   One reason for high power consumptions are evolved manufacturing technologies. It is possible to reduce the size of transistors to a point where the insulation between Gate, Source and Drain consists of only some atom layers. Thus tunneling through the Gate and sub-threshold leakage occurs.
   Anyway, no lower-power technology is available. So the developers had to find a way between performance and power efficiency.

3. Frequency and Pipelining

   It seems that a point is reached, where further increasing of pipeline depth and processor frequencies has diminishing returns. An increased pipeline means always more latches, which increases the instruction latencies. The penalties associated with the increased instruction execution latency have to be compensated by the increased frequency and the ability to execute more instructions. In case of a mispredicted branch instruction, more pipeline steps are junked in contrast to a short pipeline.

   The main challenge was to develop implementations and processor architectures using a possible small pipeline depth.

- **Real-time responsiveness to the user and the network**

  To achieve a high immersion, the Cell has to react as quick as possible to user input and network requests. A computer game, for example requires a continuing updated virtual environment with consistent video, audio and other

sensor feedback to the user. Any anomaly diminishes the game experience. Simultaneously the Cell should be able to handle the broadband Internet traffic. This means not only to process communication-oriented, but also Internet specific workload. Because the Internet is supporting a wide variety of standards, like many different video streaming formats, any acceleration function must be programmable and flexible.

- **Applicability to a variety of platforms**

  The primary field of usage for the Cell was originally the next-generation of entertainment systems. Its strength should be games and multimedia applications, for Sonys next video game console Playstation 3 and HD devices from Toshiba.

  In order to extend the applicability of the Cell and to start a developing community, an open software development environment based on the GNU/Linux operating system has been released.

- **Introduction in 2005**

  One intention of STI was to present the Cell Broadband Engine Architecture in the year 2005. In consequence only four years of development were available to achieve the previously mentioned targets. But for developing a new architecture more than four years are typically needed. This was one reason to use and improve the existing Power Architecture.

2.1.2 Architecture

The Cell Processor consists of one dual-threaded, dual-issue, 64-bit Power processor element (PPE) compliant to the Power Architecture. The Power Architecture is extended with eight cooperative offload processors called "Synergistic processor elements" (SPE). Additional one memory controller and two interface controllers are located on the die. Figure 2.1 shows a complete picture about the Cell. The components are explained on the following pages.

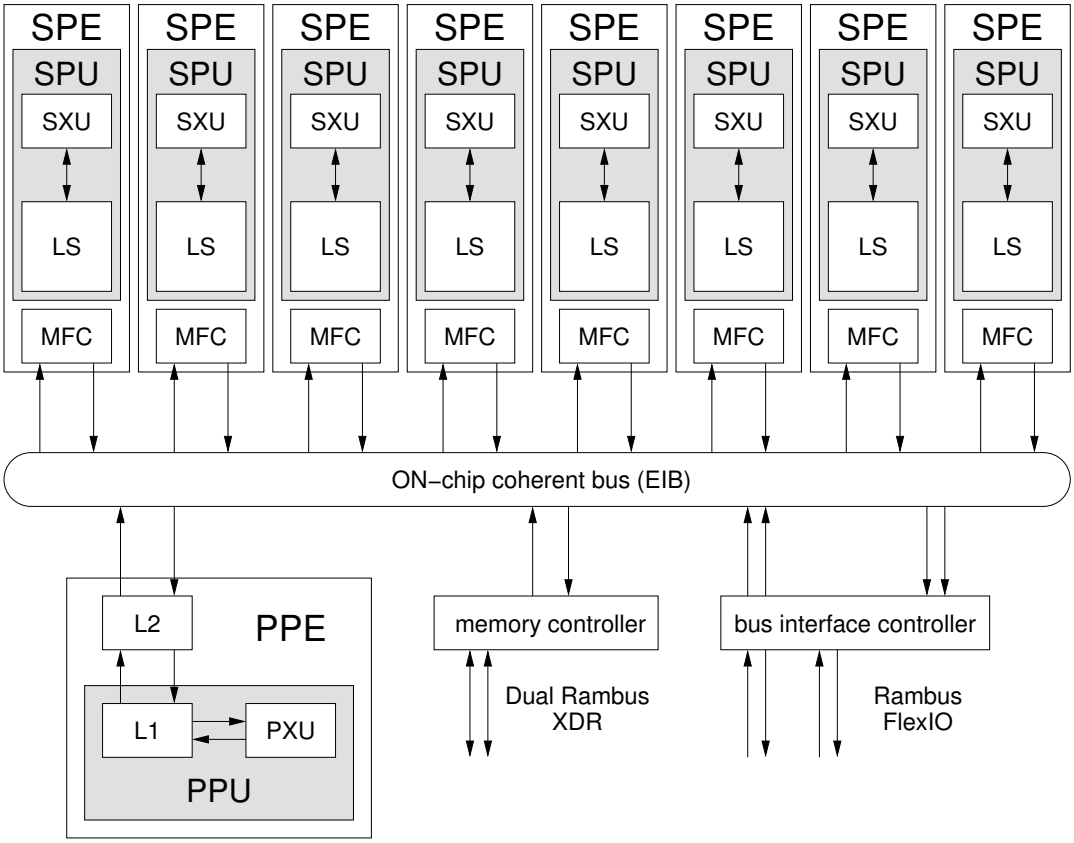

Figure 2.1: Cell block diagram

Element Interconnect Bus

The twelve elements (1xPPE, 8xSPE, 1xMC, 2xIC) are connected with a special ring bus called "Element Interconnect Bus" (EIB). This bus consists of one ring with four unidirectional channels, two have clockwise direction and two have counter-clockwise direction. Each channel is 16 Byte wide and can handle a maximum of three transactions synchronously. The EIB works with a beat[1] of the half than the system clock. This reflects in a bandwidth of 96 Byte per system clock (16 Byte * 12 transaction / 2).

Every element has one 16 Byte read and one 16 Byte write Port. So it is possible to write and read 16 Byte per beat.

[1]IBM calls the clock of the EIB "beat" for a better differentiation from the system clock

As a simplification each transfer on the bus takes always eight beats, thus one transaction is 8 * 16 Byte = 128 Byte long. Every three clocks a ring can start a new transaction [3]. The data transfer works with single steps through the ring. The maximum way length is six steps (either clockwise or counter-clockwise with twelve elements). Long ways diminish the performance of the bus, because transfers cannot overlap and so the number of parallelism is limited on the channel.

Power Processor Element

The Power Processor Element (PPE) consists of a 512 KiB Level-2-Cache and the Power Processor Unit (PPU). Among the Power Execution Unit (PXU) the PPU contains a 32 KiB Level-1-Cache.

The PXU is based on the Power Architecture executing two threads in parallel. Equipped with an Altivec unit, the PXU is able to process two floating point operations with double or eight floating point operations with single precision at the same time. Running on a clock of 3.2 GHz this is equal to 6.4 GFLOPS respectively 25.6 GFLOPS. The instruction set architecture (ISA) is nearly the same as on other 64-Bit PowerPC based processors. So it is possible to execute normal operating systems like GNU/Linux unchanged on the Cell.

The main tasks of the PPE are managing the SPEs (Synergistic Processing Elements) and executing the operating system.

Synergistic Processing Element

The Synergistic Execution Unit (SXU) is similar to a RISC (Reduced Instruction Set Computing). The register file contains 128 registers of 128 bit. The ISA includes SIMD (Single Instruction Multiple Data) operations for Integer, Float and Double. Next to the SXU a 256 KiB local-store (LS) is integrated into the SPU (Synergistic Processing Unit). The SXU always operates on this local-store, which is also mapped into the memory map of the Cell. The mapping is not coherent if it is cached in the system. The program code and all (local) data must fit into the LS. If the application accesses addresses in the main memory or other units, DMA (Direct Memory Access) requests have to be done. DMA is handled by the Memory Flow Controller

(MFC) integrated in the Synergistic Processing Element. The processing happens asynchronously to the SPU. All DMA commands are coherent and use the same protections and translations provided by the page and segment tables of the Power Architecture. Addresses can be passed between the PPE and SPE, because these tables are equal for both. The operating system is able to configure shared memory and is able to manage all resources in the system in a consistent manner.

The theoretical floating point performance of one SPE is 25.6 GFLOPS at a 3.2 GHz clock. Together with the PPE the Cell has 230.4 GFLOPS peak performance in single precision.

Memory Flow Controller

DMA is the only way for the SPU (Synergistic Processing Unit) to communicate with the rest of the Cell. Therefore the MFC is integrated into each SPE. There are three ways to program the MFC:

1. Executing instructions on the SPE, inserting DMA commands in the queue
2. Issuing a "DMA list" command with a prepared (scatter-gather) list on the local-store
3. Insertion of a DMA command in the queue by another processor element (with the appropriate privilege).

The maximum outstanding DMA commands are 16.

Memory Controller

The Memory Controller of the Cell supports Dual Rambus XDR memory. With two 32-bit memory channels the bandwidth is 25.6 Gbit per second. The EIB delivers a little bit more bandwidth, there is almost no perceivable contention on DMA transfers between the units of the Cell.

I/O Controller

The Cell has an integrated high-bandwidth flexible I/O interface (Rambus RRAC FlexIO). On the physical layer are twelve unidirectional 8-bit wide lanes. Seven are

outbound and five are inbound. The I/O-clock is independent of the system clock. The lanes can be dedicated into two separate logical interfaces, one can operate coherent (IOIF0).

2.1.3 Cell Multiprocessor Systems

Figure 2.2 shows a typical single Cell processor configuration with FlexIO configured as two interfaces.

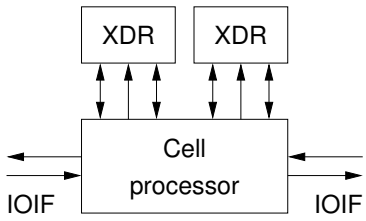

Figure 2.2: Single Cell configuration

To couple two Cells, one I/O-interface has to be configured for BIF (Broadband Interface). Only BIF provides the necessary coherence on the hardware layer. In figure 2.3 a dual Cell is shown.

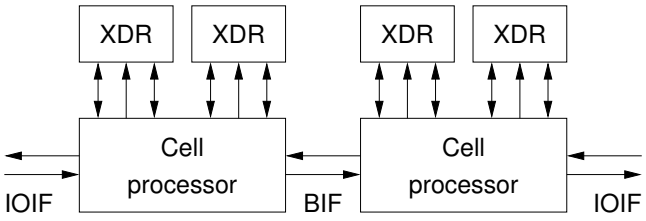

Figure 2.3: Cell dual processor system

To build a four-way symmetric multiprocessor, glue logic in form of a switch has to be used (figure 2.4).

2.1.4 Hardware

Except the previously mentioned Playstation 3 from Sony, Mercury Computer Systems manufacture a PCIe (Peripheral Component Interconnect Express) Cell accel-

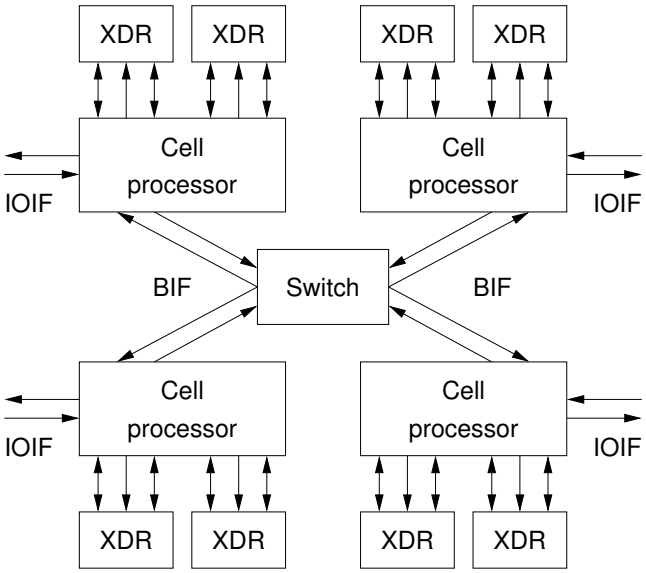

Figure 2.4: Quad-Cell via Switch

eration card.

IBM sells a dual-Cell SMP Blade (QS21). In their test series the Cell needs one watt per 1.05 GFLOPS. This is a very power efficient computer platform [4].

Toshiba has developed an acceleration card for PCIe with the name "**SpurseEngine**". In [5] it is described as:

"*The SpursEngine is a highly efficient stream processor. The SpursEngine incorporates 4 SPEs, which are the high-performance signal processing processors of Cell Broadband Engine [...], and hardware video codecs such as MPEG2/H.264. Therefore you can quickly handle operations (for example: compression/decompression of video images, physics operations, or computer-graphics) that can not be executed satisfactorily by a single processor. By using the SpursEngine, you can accelerate the processing of these operations that video applications need.*"

2.2 Heterogeneous Multiprocessing in an Opteron-Cell Environment

Within the context of heterogeneous multiprocessing the SPEs of the Cell processor are seen as accelerator units. The Opteron processor is the main processing unit. The expected performance of this combination is very high. On the one hand the system gets advantage from the high I/O throughput of the Opteron and on the other hand it owns the high computational power from the Cell. The Cell leaks the feature of parallel I/O processing, because the operating system is running on the PPE. Thus every I/O request from the SPE is routed through the PPE (confer section 3.5), which becomes a bottleneck for the SPEs.

Because there was no hardware available supporting the direct coupling of Opteron and Cell processor, an Ethernet-based connection has been used as intermediate step. Before taking a closer look at this development platform, different connection types must be described.

2.2.1 Interconnections

This subsection classifies different types of interconnections. Figure 2.5 shows an overview.

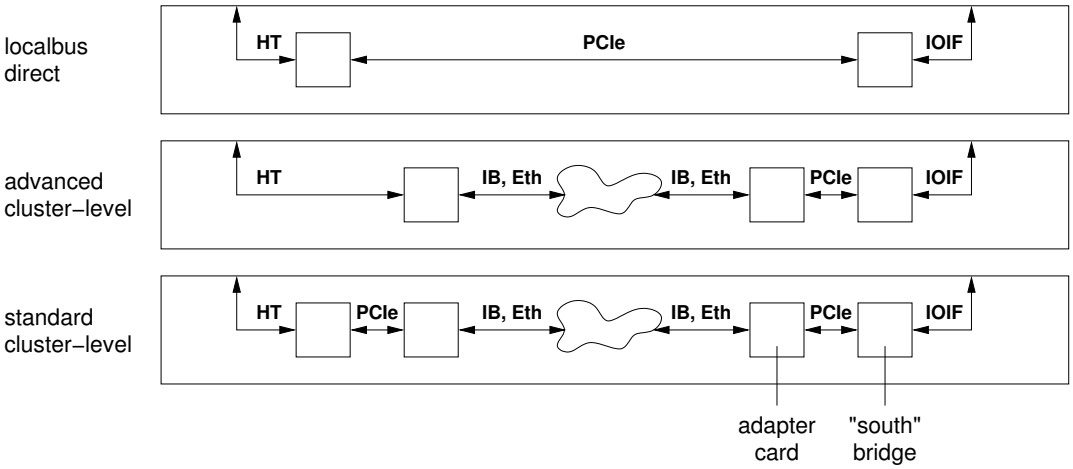

Figure 2.5: Interconnect classification

The Opteron processor is located on the left and the Cell on the right. The bandwidth is growing from bottom to top while the latency is decreasing.

Standard cluster-level

A typical representation of the *standard cluster-level* is Ethernet (Eth) or InfiniBand(IB). Both using a network to connect several hosts. The adapter cards are plugged in a PCIe-slot connected to a kind of south-bridge.

Advanced cluster-level

In the *advanced cluster-level* the "south"-bridge is omitted on the Opteron side. The interconnect adapter, which was attatched to PCIe in *standard cluster-level*, is now plugged in a direct HyperTransport (HT) slot.

Local bus direct

In *localbus direct* the Opteron and Cell are coupled directly through the PCIe-bus[2]. The Opteron serves as Host endpoint and the Cell as device endpoint.

2.2.2 Development Environment

For develop the operating system driver "rspufs", a standard cluster-level connection is used between a dual-Cell (figure 2.3) Blade and a PowerBook G3. As connection type Ethernet with TCP/IP protocol was chosen. This decision was based on the facts, that TCP/IP supports error correction, the detection of duplicate packages, the preservation of the package ordering and the point to point connection. In view of the next steps towards a tightly coupled system, these are very important features. Another reason is Firewalls and VPNs (Virtual Private Network) can handle TCP/IP traffic very well, thus it was possible to locate both systems on different places on the campus.

However, this constellation is very far from coupling the x86 Architecture with the Cell. But the advantage is the PowerBook has the PowerPC Architecture like the

[2]PCIe is a serial point to point connection and not a bus. But it is software compatible to PCI. However, it is common to say "PCIe-bus. "

PPE of the Cell, so no byte ordering problems can occur. The most important advantage is the low porting effort for the ”libspe” library (see also 3.4.2). Nearly all applications for the Cell are using this library, thus it is possible to use these applications nearly unchanged.

After the working PowerBook-Cell system the next step was the connection with a x86 system. In the first instance a Pentium III Notebook was taken to port the rspufs and the libspe.

2.2.3 PCIe Coupling

The next step after this diploma thesis would be the usage of a better interconnection via PCIe (a *local bus direct* connection type). The result would be not only a better performance, but the ability to use RDMA (Remote Dynamic Memory Access). RDMA is very important, because most parts of the Cell are memory mapped. Thus these are directly accessible from the Opteron. Unlike the current approach, where RSPUFS simulates the direct access in software, RDMA provides a hardware based solution.

The Los Alamos National Laboratories (LANL) working on a project with the name ”**Roadrunner**”[6], where the Cell and the Opteron is coupled with PCIe, too. The goal is to place the first high performance computer with more than one PetaFLOPS in the top-500 super computing list. LANL is planing a ratio of one to one between Opteron and Cell processors. According to [7], this system provides the application level library DaCSH (Data Communication and Synchronization Heterogeneous) for the communication with the PPE only. The SPEs are not directly visible by the application portion on the Opteron.

However, with the RSPUFS solution a direct access to the SPEs should be granted. But this platform is only an intermediate step before the targeted ”tightly coupled Opteron Cell evaluation platform”.

2.2.4 Opteron Cell evaluation Platform

This platform is a case study integrating the Cell Broadband Engine Architecture into an AMD Opteron Platform. Integration means tightly coupling of the processor buses via an FPGA (Field Programmable Gate Array). The FPGA has to translate the *Hyper Transport* (HT) of the Opteron to the *Global Bus Interface* (GBIF) of the Cell and vice versa.

In figure 2.6 a simplified structure is shown. Neither the Opteron nor the Cell have a real "Northbridge". On both platforms the memory controller is integrated on the processor die. So the term "Northbridge" describes the processor bus I/O interface.

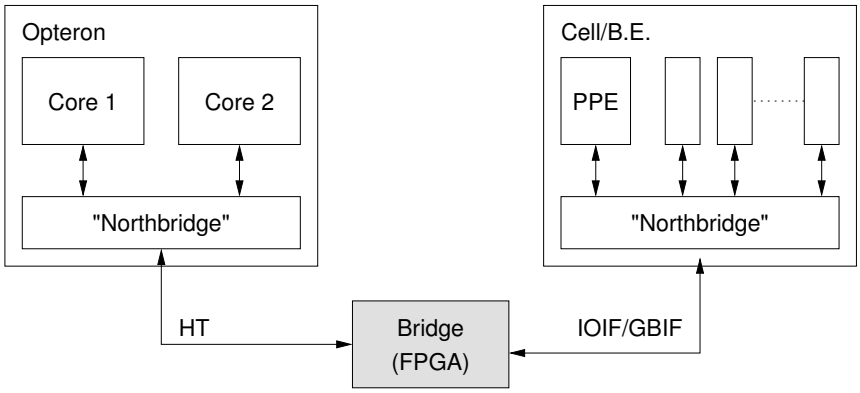

Figure 2.6: Tightly coupled Opteron Cell evaluation platform

This platform provides more performance than *local bus direct*. Its bandwidth is higher[3] and the latency is lower, because there are no Southbridges and no double protocol translation (HT to PCIe and PCIe to GBIF) necessary. However, the main reason to build this platform is the availability of coherent memory, because HT as well as GBIF are cache coherent.

All main features of RSPUFS are focused on this platform. If the PCIe coupling is available and RSPUFS is ported to PCIe, the porting effort to the *tightly coupled Opteron Cell evaluation platform* should be very low.

[3]The peak bandwidth of HT is currently specified (HT 3.0 @ 2.6GHz) at 20.8GB/s and the peak bandwidth of PCIe (v2.0) is specified at 16GB/s

3 SPUFS

The Synergistic Processing Unit File System (SPUFS) is the Linux programming model for the Synergistic Processing Units of the Cell. It is only available for the PowerPC branch of the Linux kernel. RSPUFS is built on top of SPUFS. Thus it is an important part of this thesis to analyze SPUFS.

This chapter shows an overview of the SPUFS concepts and the interface provided by SPUFS to the application.

3.1 Introduction

To integrate the SPEs into the Linux environment several basic approaches are available [8].

1. **Character Devices**

   A character device is a simple way to enable applications access to hardware resources. Each SPU would be represented as a character device. For controlling only `read`, `write` and `ioctl` system calls are required.

   However, it will be hard for an application to find unused SPUs if each is represented as a single device. Furthermore it is very difficult to virtualize the SPUs on a multi-user system.

2. **System Calls**

   With the definition of a new set of system calls and a new thread space, it is possible to abstract the physical SPU with "SPU process". The advantage is, that the kernel can schedule these SPU processes and every user is able to create them without interfering with each other.

   A possible high number of new system calls is needed to provide the necessary functionality. To manage another type of processes, kernel infrastructure

has to be duplicated. Thus changes or an alternative version of system calls manipulating the process-space like `kill` or `ptrace` are required.

3. **Virtual File System**

   Like the system call approach, a virtual file system (VFS) does not require any device drivers. All resources are stored instead in the main memory. For the communication between user- and kernel-space, system calls like `open`, `read`, `write` or `mmap` are used.

Because of the disadvantages from character devices or the new thread space, IBM has decided to implement a virtual file system with the name "SPUFS". The integration into the Linux kernel is shown in figure 3.1. The "*spufs*"[1] kernel driver uses the

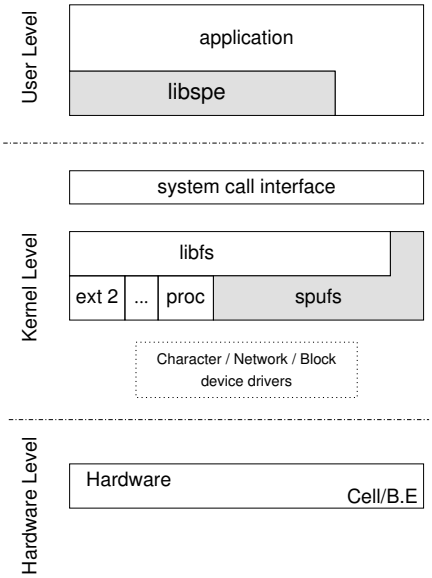

Figure 3.1: SPUFS inside the Linux Kernel

VFS interface of Linux represented by *libfs*. By convention the default mount point is "/spu". As shown in the figure, some functionalities are directly exported as system calls. They are explained in subsection 3.4.1. An application has not deal with the

[1] *spufs* in lower-case letters is the name of the Linux kernel driver.

file system parts directly. A low level library with the name ”libspe” was introduced by IBM (subsection 3.4.2).

The dotted area in the figure is the device driver layer of the Linux kernel. Even though SPUFS is not a part of it, it could be considered to belong to this layer. The reason is SPUFS manages the hardware (SPE) directly, what normally only device drivers do.

However, SPUFS is more than just a file system or a kind of a device driver. Rather it is a integration concept of the SPE as accelerator unit in the Linux system.

3.2 The SPUFS Concept

The SPE is a raw resource in the system. For example, only sixteen of these are available on a Dual-Cell Blade. This amount will may be not enough to fit the needs of all applications, if many users are working on one system. But this problem is already solved for the main processing unit, which is virtualized in the operating system environment. Their every processes is getting its own context, with an image of the real system stored in the main memory. This context includes:

- Registers
- Program Counter
- Processor Status
- Segment Pointers
- Priority
- Root Directory
- Working Directory
- File Descriptor Table
- Page Table
- ...

This is the internal view of the context. It is not possible for an application to access these entries directly. For modification and status requests the operation system provides special system calls like `getcwd` (getting the current working directory) or `nice` (setting the priority).

However, the SPUFS provides a similar solution. The only difference is the way of modifying the internal context. Instead of introducing a high amount of new or changed system calls, a VFS approach is used. The context components (see section 3.3) are accessible through normal file system calls.

To sum up, the SPUFS concept has two parts: The virtualization of the SPUs and the export of the context through a VFS interface to the user-space.

3.3 Internal Context

```
struct spu_context
{
        struct spu *spu;
        struct spu_state csa;
        spinlock_t mmio_lock;
        struct address_space *local_store;
        struct address_space *mfc;
        struct address_space *cntl;
        struct address_space *signal1;
        struct address_space *signal2;
        struct address_space *mss;
        struct address_space *psmap;
        struct mutex mapping_lock;
        u64 object_id;

        enum { SPU_STATE_RUNNABLE, SPU_STATE_SAVED } state;
        struct mutex state_mutex;
        struct mutex run_mutex;

        struct mm_struct *owner;

        struct kref kref;
        wait_queue_head_t ibox_wq;
        wait_queue_head_t wbox_wq;
        wait_queue_head_t stop_wq;
        wait_queue_head_t mfc_wq;
        struct fasync_struct *ibox_fasync;
        struct fasync_struct *wbox_fasync;
        struct fasync_struct *mfc_fasync;
        u32 tagwait;
        struct spu_context_ops *ops;
        struct work_struct reap_work;
        unsigned long flags;
        unsigned long event_return;

        struct list_head gang_list;
        struct spu_gang *gang;

        /* scheduler fields */
        struct list_head rq;
        struct delayed_work sched_work;
        unsigned long sched_flags;
        unsigned long rt_priority;
        int policy;
        int prio;
};
```

Figure 3.2: `struct spu_context` in linux-2.6.22

The internal context reflects itself in the `struct spu_context` shown in figure 3.2. It contains the address configurations of the mappable parts of the SPE, the message boxes und `mfc` entries and other housekeeping stuff like reference counters and syn-

chronization primitives. Scheduling parameters are included in this structure, too. But the most interesting entries are `struct spu *spu` and `struct spu_state *csa`.

Before the SPU code could be executed, the corresponding context has to be bind to a physical SPU by the SPUFS scheduler. A binded context has a valid `*spu` entry pointing to the physical SPU executing it. This structure is the device representation of the physical SPU.

If a context has no physical SPU, it will be stored in the `csa` (context safe area) represented by `struct spu_state` (confirm figure 3.3).

```
struct spu_state
{
        struct spu_lscsa *lscsa;
        struct spu_problem_collapsed prob;
        struct spu_priv1_collapsed priv1;
        struct spu_priv2_collapsed priv2;
        u64 spu_chnlcnt_RW[32];
        u64 spu_chnldata_RW[32];
        u32 spu_mailbox_data[4];
        u32 pu_mailbox_data[1];
        u64 dar, dsisr;
        unsigned long suspend_time;
        spinlock_t register_lock;
};
```

Figure 3.3: `struct spu_csa` in linux-2.6.22

The structure contains all the data for suspending and later resuming the SPU program execution. This includes the register file, local-store, problem state, privileged areas, channel status and the data of the inbound and outbound mailboxes.

3.4 User-Level Interfaces

This section discusses in a bottom-up way the interface exported from SPUFS to the user-space. In 3.4.1 the system programing interface will be introduced and after this, a library abstracting the low level parts is depicted. At the end of the section the toolchain for creating PPE programs with embedded SPE code is presented.

3.4.1 System Programing Interface

The first step for applications is to create a context for each SPE they require. Creating contexts with the system call `spu_create` (figure 3.4) is the only operation available on the *spufs* root. With the appropriated flag it is possible to influence the affinity of the SPUs. However, this diploma thesis deals not with affinity or the gang principle, because it is not (yet) supported by RSPUFS.

```
#include <sys/types.h>
#include <sys/spu.h>

int spu_create(const char *pathname, int flags, mode_t mode);
```

Figure 3.4: The `spu_create` system call

Each context is represented by a directory. The file descriptor for this is the value returned by `spu_create`. If the application closes this descriptor, the context gets destroyed.

The content of the directory are regular files. A subset is listed in table 3.1. The first column points to the file name, the second to the access permissions and the last to a short description. Through these files the application gets access to the single portions of the Cell. The most important entries are:

- **mem**

  The whole local-store is represented by the ”mem” file. Simple I/O system calls can be used to place data into it.

  It is possible to map the local-store into the address space of the application by calling `mmap`. If more than one is mapped, it will be possible to move data between them via *memcpy*. This results internally in a DMA transfer. Moreover, the mapping has the advantage, that the operating system is not involved by each access. Besides, this circumstance avoids copying the data twice (first time in the read/write buffer and the second time from the read/write buffer to the local-store memory by the Linux kernel).

File	Perm		Description
decr	r	w	SPU Decrementer
decr_status	r	w	decrementer status
event_mask	r	w	event mask for SPU interrupts
fpcr	r	w	floating point status and control register
mbox	r	-	the first SPU to CPU communication mailbox
mbox_stat	r	-	length of the current queue
ibox	r	-	the second SPU to CPU communication mailbox
ibox_stat	r	-	length of the current queue
wbox	-	w	CPU to SPU communication mailbox
wbox_stat	r	-	length of the current queue
mem	r	w	local-store memory
npc	r	w	next program counter
psmap	r	w	problem state area
regs	r	w	register file
signal1	r	w	signal notification channel 1
signal1_type	r	w	behavior of the signal1 (replace or ”OR”)
signal2	r	w	signal notification channel 2
signal2_type	r	w	behavior of the signal2
srr0	r	w	interrupt return address register

Table 3.1: The SPUFS context (part)

- **regs**

  The register file of the SPU is accessible via this entry. Any operation causes the SPE making a complete halt. But normally there is no need to access the registers directly during the program execution.

- **mbox**

  ”mbox” is the first SPU to CPU communication mailbox. It can only be read in units of four bytes in non-blocking mode. If there is no data available, `read` returns -1 and *errno* is set to EAGAIN.

- **ibox**

  The second SPU to CPU communication mailbox is ”ibox”. The only difference to ”mbox” is, that this file can be read in blocking mode.

- **wbox**

  For the communication from the CPU to SPU "wbox" has to be used. Only units of four bytes can be written to this file. If the mailbox is full, `write` will block until it is becoming empty. If the file is opened with O_NONBLOCK, the return value is -1 and *errno* is set to EAGAIN.

- **wbox_stat / mbox_stat / ibox_stat**

  These three files contain the length of the current mailbox queue: how many words can be written to wbox or read from mbox or ibox without blocking. `read` on these files is only possible in units of four bytes. The result is a binary integer number in big-endian format.

- **npc**

  It is possible to `read` and `write` values to the next program counter without stopping the execution of the SPU. The value is an ASCII string with the numeric value of the next instruction.

- **mfc**

  The Memory Flow Controller is accessible through this file. With `write` it is possible to insert a new DMA requests into the MFC. The MFC Tag Status Register is returned by reading the file.

- **psmap**

  The whole problem-state area is mappable via `mmap` the "psmap" file. The problem-state area is the memory representation of special parts of the SPE:

  - MFC DMA setup and status registers
  - PPU and SPE mailbox registers
  - SPE run control and status register
  - SPE signal notification registers
  - SPE next program counter register

After loading the program into the local-store memory the `spu_run` system call (Figure 3.5) executes the SPU code.

```
#include <sys/types.h>
#include <sys/spu.h>

int spu_run(int fd, unsigned int *npc, unsigned int *event);
```

Figure 3.5: The `spu_run` system call

The execution is synchronous. Thus the thread calling `spu_run` is blocked, while the SPU code is running. After returning it is possible to execute the system call again without updating the pointers, because the SPU instruction pointer is written back to *npc*. Returning has one or more of the following reasons:

- Wrong usage of `spu_run`
- SPU was stopped by **stop-and-signal**
- SPU was stopped by **halt**
- SPU is waiting for a channel
- SPU is in single-step mode
- SPU has tried to execute an invalid instruction
- SPU has tried to access an invalid channel

It is also possible to start the SPU-code execution by writing the appropriate values into the ”run control” register, located into the problem-state area. However, it is recommended to use the `spu_run` system call.

3.4.2 SPE Runtime Management Library

To provide a further abstraction to the application, a SPE Runtime Management Library called ”libspe” is provided by IBM. The library interface has changed from libspe1 to libspe2 since the Cell/B.E. SDK 2.1. The main differences are the way of executing the SPE code and the context creation. While the execution is synchronous in libspe2, it is asynchronous in libspe1. The context creation and execution are one step in libspe1 and several steps in libspe2. In the rest of this thesis ”libspe” always means the libspe2 interface.

Libspe is a user-space library, it does not manage physical SPEs. Hardware resources can not be manipulated directly. The library requests SPE resources from the operating system, taking no concern about how the operating system implements this. It is also allowed to schedule multiple contexts on one SPE. The operating system can suspend execution at any point. But all this happens transparently to the libspe and the user application. Only diminished performance is noticed in cause of excessive context switches. According to the past sections the *spufs* Linux kernel driver does exactly those things.

The last part of this subsection shows an overview of the functionalities provided by libspe. For a closer look please refer to [9].

SPE context management

Like in SPUFS the first action for an application is to create the number of contexts it needs. Therefore `spe_context_create` has to be called. While creating, a library internal structure gets initialized and the `spu_create` system call is executed. To destroy the context `spe_context_destroy` has to be called.

CPU information

The function `spe_cpu_info_get` provides an application some basic information about the system. This includes the number of physical PPEs and SPEs and the usable SPEs. Usable SPEs can actually be scheduled to run for the application and are not ”pinned” by the operation system.

SPE program image handling

Before execution, the SPU code has to be loaded. There are two possibilities of loading the program. The first possibility is loading from an external SPE ELF with the function `spe_image_open`. The second possibility is to embed the image into the PPE executable via an *Embedder* as described in the toolchain section (section 3.4.3). The image is loaded into the local-store by `spe_program_load`.

SPE run control

The function `spe_context_run` executes the SPE code loaded by `spe_program_load`. This function internally executes the system call `spu_run` provided by SPUFS. Because this happens synchronously, the calling thread is stopped. In case the program needs more than one SPE (assume N), it is common to use $N+1$ threads. N threads are used for the SPEs and one thread for the main program itself.

When `spu_run` gives the execution control back, the libspe does not return immediately. Depending on the return code, callback handlers are executed. This mechanism is described in detail in section 3.5.

SPE event handling

To receive information caused by the asynchronously running SPE threads, it is possible to create (`spe_event_handler_create`) and register event handlers (`spe_event_handler_register`). The supported events are:

- SPE_EVENT_OUT_INTR_MBOX (data available in the mailbox)
- SPE_EVENT_IN_MBOX (space for new mailbox message available)
- SPE_EVENT_TAG_GROUP (a SPU event tag group signaled completion)
- SPE_EVENT_SPE_STOPPED (SPU program stopped)

SPE MFC proxy command

Libspe provides an interface to initiate DMA transfers from the PPE side. Various commands, like `spe_mfcio_put` or `spe_mfcio_get`, are available to write data in or out of the local-store directly by the MFC on the SPE.

SPE MFC multi-source synchronization

A multi-source synchronization of a specified SPE is started with the function `spe_mssync_start`. The MFC tracks all uncompleted transfers at this SPE. If all transfers across the local-store and main memory are finished, the `spe_mssync_status` function will return zero.

SPE MFC proxy tag-group completion functions

The completion of DMA requests can be checked with `spe_mfcio_tag_status_read`.

SPE mailbox functions

Sending and receiving short messages (mails), each with a size of 32-bit, is supported by libspe, too. The mailbox functions match exactly the entries of the SPUFS virtual file interface. This shows clearly that libspe is a wrapper for the most parts of the SPUFS functions.

SPU signal notification functions

The function `spe_signal_write` can be used to write in the signal notification registers of the SPU.

Direct SPE access

The local-store is memory mapped by libspe. The application can get access to it by calling `spe_ls_area_get`. With `spe_ps_area_get` it is possible to modify the problem state area directly.

PPE-assisted library facilities

With the help of the PPE, the SPE can use system calls or other libraries which are too large for the local-store. The mechanism named "Externally Assisted SPE Library Calls" will be explained in section 3.5.

3.4.3 Toolchain

The PPU and SPU instruction set architectures are not compatible. In consequence the application has to be split into two binaries. It is desirable to combine both into one single file. Therefore the toolchain from figure 3.6 can be used. The SPE executable becomes embedded by the "SPE Embedder" (`ppu-embedspu`) into a PPE object file. Then this object can be linked with other PPE objects to a single PPE executable file. The libspe library can load those embedded images into the local-store of the SPE.

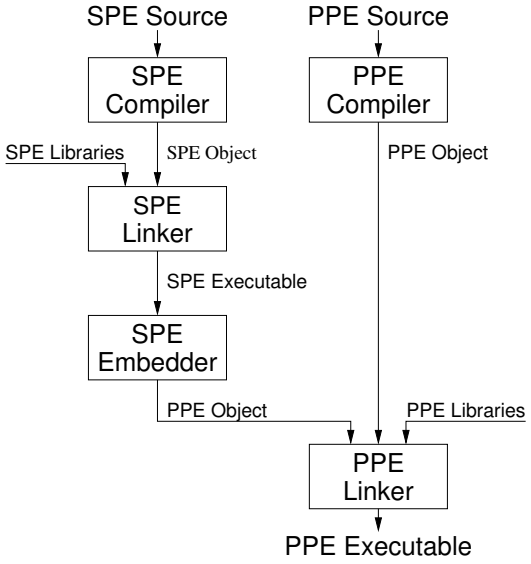

Figure 3.6: SPUFS: Tool Chain

The embedder is a shell script building the embedded SPU image. This process consists of three steps. The first step is copying the whole SPU Elf file into a read-only section. The second step is to filter special "_EAR_" symbols and write them in form of a table into a seperate data section. With those symbols it is possible to access corresponding symbols in the PPU program. The last step is the creation of "struct `spe_program_handle`" (figure 3.7).

```
typedef struct spe_program_handle
{
        unsigned int    handle_size;
        void            *elf_image;
        void            *toe_shadow;
} spe_program_handle_t;
```

Figure 3.7: `struct spe_program_handle`

This structure is a handle for the SPU image in the PPU program. The pointers `elf_image` and `toe_shadow` are set to the sections created in step one and two. This structure could be directly loaded from libspe with `spe_program_load`.

3.5 Externally Assisted SPE Library Calls

The SPE is designed to handle computational workload. However, outside the compute kernels functionalities are needed like allocating memory or printing status informations. The operating system is running on the PPE. Thus a direct call of the necessary system calls is not possible. To not abandon these functionality, the Application Binary Interface (ABI) [10] specifies PPE assisted library calls.

This section describes the interface with the help of the example in figure 3.9 on page 31. In line 40 the string "Hello PPE!" has to be written to standard output. Therefore a stub for the `write` system call is provided (line 27). To perform the assisted call the SPE has to construct a local-store memory image with the input and output parameters. Each parameter must be padded to a quadword boundary. For the `write` system call the image has the format shown in figure 3.8.

0	int fd	pad	pad	pad
16	char * buf	pad	pad	pad
32	size_t count	pad	pad	pad

Figure 3.8: Parameter image for an assisted `write` call

The structure in line five implements this. After creating and filling it with the parameters, `__send_to_ppe` is called. The arguments of this function are *signalcode*, *opcode* and a pointer to the previously created local-store memory image holding the parameters. The pair *signalcode* and *opcode* are selecting the library function. All possible signal codes are shown in table 3.2. The system call `write` is included into POSIX.1 with the Opcode 27 (table 3-7 in [10]). To signal the PPE the "stop-and-signal" machine instruction [11] has to be executed by the SPE. The ABI defines that the next 32-bit are the assisted call message. This message is a combination of the *opcode* in the upper byte and the address of the local-store memory image in the lower byte (line 15). To avoid changing the program code at runtime, a helper function is created on the stack (lines 16-20). The machine code of "stop-and-signal" has only zeros in the instruction bits. Thus the *signalcode* could be directly used as command. Subsequently the assisted call message, a "nop" and a "return" is written.

Signal Code	Description
0x0000	Data executed as an instruction.
0x2000-0x20FF	Return from main or exit. The return code is encoded in the least significant byte.
0x2100	ISO/IEC C99 library callback.
0x2101	POSIX.1 library callbacks.
0x2102	POSIX.1b library callbacks.
0x2103	Operating-System-Dependent system calls.
0x2105	Libea callbacks.
0x2106-0x21FF	User defined callbacks.
0x2200-0x220F	SPE isolation mode errors.
0x3FFE	Stack overflow detected.
0x3FFF	Debugger breakpoint.

Table 3.2: Cell ABI stop-and-signal code definition

By the execution of this stack function (line 24) the SPE stops and the PPE gets signaled.

On the PPE this reflects a return from `spu_run`. In the case of the example, the return value is 0x2701. The Runtime Management Library (libspe) calls the POSIX.1 library handler, which is reading the assisted call message to get the desired function and the appropriate parameter list. The pointer to the parameter list can be found by adding the value of *npc* to the start address of the local-store. After executing the library function, the *npc* has to be incremented by 4 Byte. The return values must be copied into the local-store memory image. If the assisted call generates errors by setting *errno*, this value has to be stored in the third word element of the quadword. The SPE execution is resumed by calling `spu_run` again.

This little example has shown the way for assisted callbacks implemented in the libspe. But it is also possible to define own functions. The callback handler could be registered with `spe_callback_handler_register`.

```
1  #include <sys/types.h>
2  #define SPE_POSIX1_CLASS        0x2101
3  #define SPE_POSIX1_WRITE        27
4  int errno;
5  typedef struct {
6          int fd;
7          unsigned int pad0[3];
8          char * buf;
9          unsigned int pad1[3];
10         unsigned int count;
11         unsigned int pad2[3];
12 } write_t;
13 void __send_to_ppe (unsigned int signalcode, unsigned int opcode, void * data)
14 {
15         unsigned int combined = ((opcode << 24)|((unsigned int)data & 0x00FFFFFF));
16         vector unsigned int stopfunc = {
17                 signalcode,
18                 combined,
19                 0x4020007F,
20                 0x35000000
21         };
22         void (*f)(void) = (void *)&stopfunc;
23         asm("sync");
24         f();
25         errno = ((unsigned int *)data)[3];
26 }
27 ssize_t write(int fd, char *buf, size_t count)
28 {
29         ssize_t * ret;
30         write_t data;
31         data.fd = fd;
32         data.buf = buf;
33         data.count = count;
34         ret = (ssize_t * ) &data;
35         __send_to_ppe(SPE_POSIX1_CLASS, SPE_POSIX1_WRITE, &data);
36         return *ret;
37 }
38 int main()
39 {
40         write(1, "Hello PPE!\n", 11);
41 }
```

Figure 3.9: PPE assisted `write` system call

4 RSPUFS

The integration of the SPE as accelerator component into the PowerPC Linux environment was demonstrated in the last chapter. This integration was based on SPUFS, which contains the concepts virtualization of the SPEs and the functionality export via a VFS approach. However, this solution was only available for the Cell system.

Within the migration process of SPUFS from the PPE to the Opteron as main processing unit, the RSPUFS project was founded. RSPUFS stands for "Remote SPU File System". The name was chosen, because of the remote location of the SPEs and the usage of the original SPUFS on the remote host.

The first part of this chapter introduces RSPUFS. In the next parts, a closer look into the internals is provided.

4.1 Targets

The main target of RSPUFS is the support of SPUs in the Opteron Linux kernel. To minimize the porting effort of existing applications the same interface like on the native Cell has to be provided. This means adapting the concepts of SPUFS to RSPUFS.

However, this is only one side. The other is to port the runtime management library "libspe", too. Because of the recommendation from IBM to use this library to program the Cell, it is required by most of the applications.

4.2 Structure

RSPUFS consists of two parts: A daemon running on the PPE of the Cell with the name "*rspufsd*" and a virtual file system driver called "*rspufs*"[1] in the Opteron Linux kernel. These components are displayed in figure 4.1.

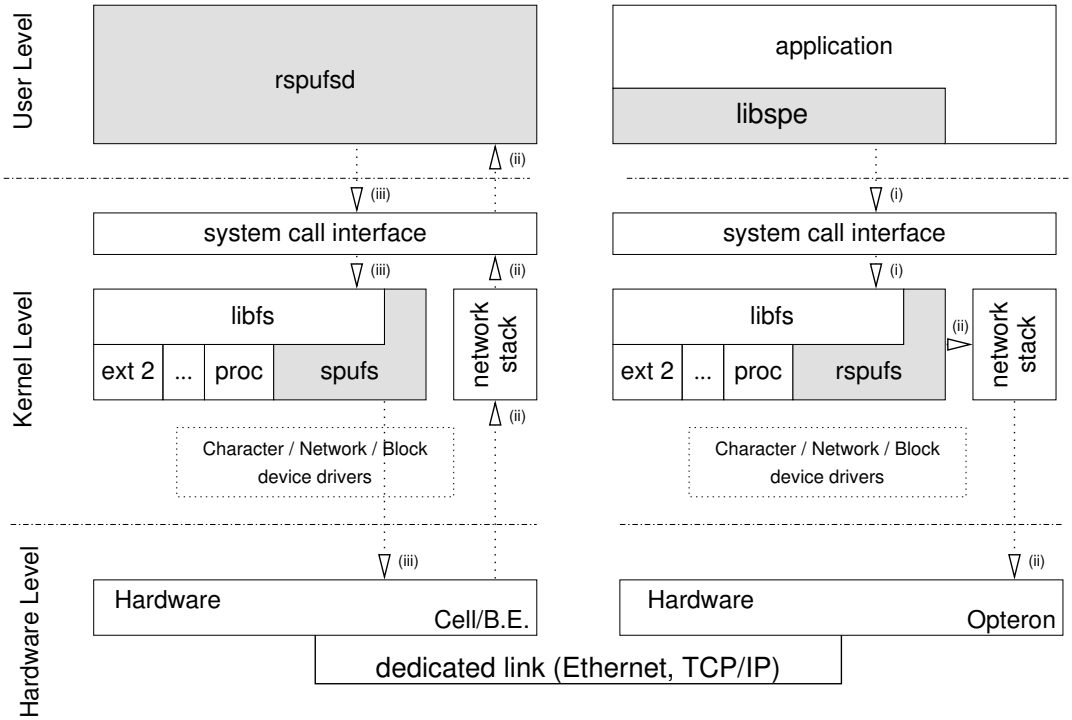

Figure 4.1: RSPUFS Model

A complete Linux system is running on the Opteron as well as on the Cell. Both are connected through an Ethernet link. It is possible to use the Cell in the normal way. The only difference is a further user-space process, the *rspufsd*, which uses the unmodified SPUFS. The reason for the decision not modifying SPUFS on the Cell side is the better assignment of errors, which occur during the development. All errors are based on the daemon, which is very easy to debug, because it is running in the user-space, where normal debuggers are working. In contrast, a fatal error in the kernel-space always results in a "kernel oops". If the kernel detects a problem, it kills

[1] According to "spufs", "rspufs" in lower-case letters means always the Linux kernel driver.

the involved processes and prints an oops message used by kernel developers to fix programing errors. In the oops state some internal resources are no longer available to avoid damage on the system. The whole VFS may not work and no data can be stored or loaded anymore. The only option is to reboot the system. Thus it is very uncomfortable to develop in the kernel-space.

On the Opteron side, the *spufs* driver is exchanged with *rspufs*, which only provides the interface. Any resource request is transmitted to the daemon on the Cell.

The dotted arrows in the figure 4.1 are an example for a communication, where the application running on the Opteron write to the Cell. The numbers will be explained in the next section.

The parts of RSPUFS are not just two program pieces, but a differentiation of the functionality. The *rspufsd* implements the hardware access (bottom half) to the SPEs and the *rspufs* kernel driver provides the user interface (top half).

4.3 The Cell Part: rspufsd

The whole RSPUFS concept works like a proxy. In figure 4.1 the application requests a resource from the *rspufs* kernel driver (i), which translates this to a network package and send it to the Cell (ii). The *rspufsd* decodes the package and reexecute the request on the Cell (iii). The results are transmitted back to the Opteron and the application (not displayed in the figure).

The main tasks of *rspufsd* could be described as: provide a service infrastructure and handle SPU requests.

In the next subsections the network protocol and the provided services are depicted.

4.3.1 Network Protocol

The RSPUFS network protocol was build on top of TCP/IP. In contrast to the stream-orientation of TCP/IP, RSPUFS is message based. The structure of the

message is shown in figure 4.2. The white parts are the header and the grey is the payload.

Figure 4.2: RSPUFS Protocol

The header fields have the following meaning:

- **Flags**

  A message must not, but can have one or more of these flags:

RSPU_F_OK	The request was processed without errors. The result can be found in the data field.
RSPU_F_ERROR	An error occurs while the execution of the request. The 4-Byte error is placed into the first four data bytes.
RSPU_F_FIN	This is a marker for the last package of the current session. Thereafter the connection gets closed.
RSPU_F_NOREP	The message is only a notification. No answer is expected.
RSPU_F_ASYNC	If a message is outside the normal data flow, it will be marked with this flag.

  If neither RSPU_F_OK nor RSPU_F_ERROR is set, the message will be service request. Otherwise it is a response.

- **Service**

  Each message has to belong to exactly one service. The services are shown in section 4.3.2. In the normal data flow the Opteron requests services from the *rspufsd*, which is responding to this. The daemon is free in the decision, which service it responds first. It is only important to respond with the right Service- and Tag-number.

  However, that does not mean the messages can overlap. Both sides have to ensure, that it is send as a whole and not in parts.

- **Tag**

  Every request gets an own Tag. Thus it is possible to assign the incoming answers to the appropriate request.

- **Size**

  The Size field contains the length of the total message. To get the length of the payload, the size of the header (8 byte) has to be subtracted.

4.3.2 Services

As mentioned in the past section, the communication inside RSPUFS is service orientated. Thereby the *rspufsd* has the server and *rspufs* the client role.

The client requests a service. The server is processing this request and is sending the response back. If the flag RSPU_F_OK is set, the execution will be successful. If an error occurres, RSPU_F_ERROR will be set. By now only the client is allowed to initiate requests (this is called ”normal” or ”synchronous” data flow). But the protocol contains all necessary features to support asynchronous requests from the server.

In the rest of this subsection all services provided by *rspufsd* are described.

RSPU_S_CREATE

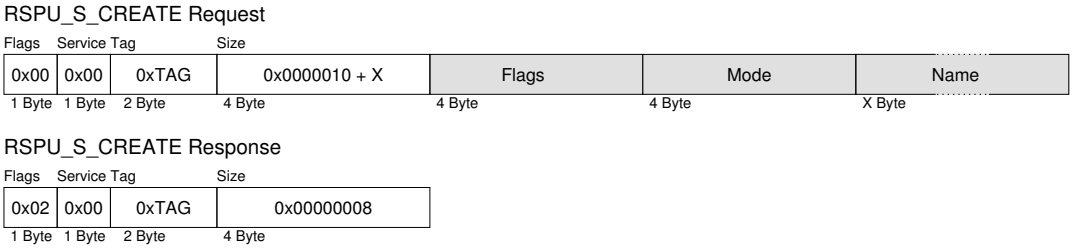

Figure 4.3: RSPU_S_CREATE message

By convention, RSPU_S_CREATE is the first message after the established TCP connection. ”Flags”, ”Mode” and ”Name” (figure 4.3) are the parameters of the

`spu_create` system call (confirm figure 3.4 on page 21) submitted by the application.

The daemon uses these parameters to execute the system call on the Cell-side again. If the context creation is successful, the response message will set only RSPU_F_OK without any data. Possible error scenarios are a malformed or already existing context name or invalid flags for `spu_create`.

Every context gets its own TCP/IP connection. Thus RSPFUS can differ the contexts just with the port, where the requests are coming in.

After the creation of the SPU context all other services are available.

RSPU_S_FILELST

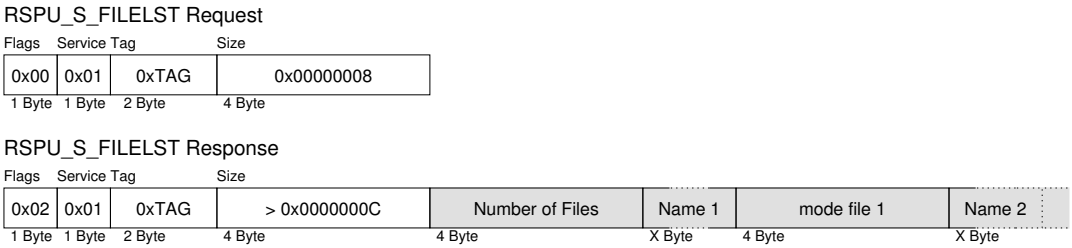

Figure 4.4: RSPU_S_FILELST message

In order to be flexible, if changes in SPUFS are made, the context entries would not be hard-coded into RSPUFS. The client has to request instead (figure 4.4) the list of all file entries. This list is generated on demand by the server for the context directory.

The respond contains the number of files with their access modes and names. The end of each name could be found by a '\0' character.

RSPU_S_FOPEN

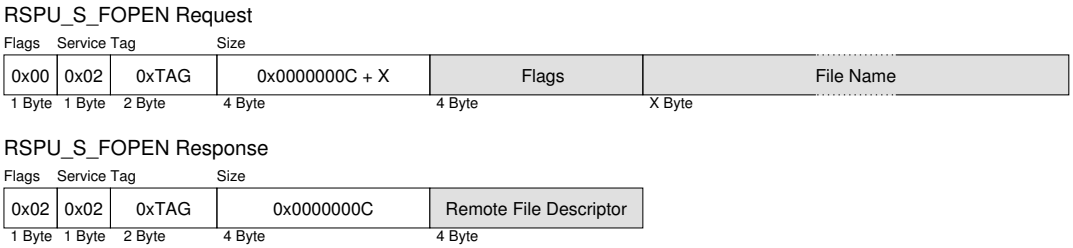

Figure 4.5: RSPU_S_FOPEN message

To open a context entry the service RSPU_S_FOPEN (figure 4.5) has to be used. This is generic for all SPUFS context items. The submitted file name must include the context name.

If the file is available and can be opened with the requested flags, the descriptor, returned by the `open` system call, will be the response.

RSPU_S_FRELEASE

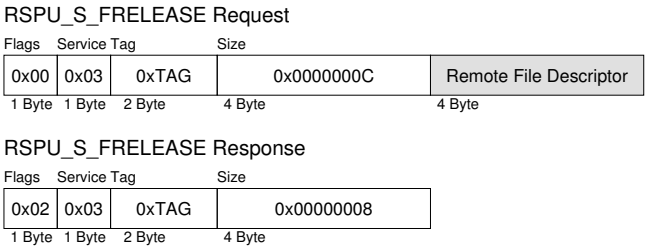

Figure 4.6: RSPU_S_FRELEASE message

Context entries opened with RSPU_S_FOPEN could be closed by RSPU_S_FRELEASE (figure 4.6). To avoid, that private descriptors of *rspufsd* get closed, a list is used tracking all file descriptors opened by the Opteron.

RSPU_S_FREAD

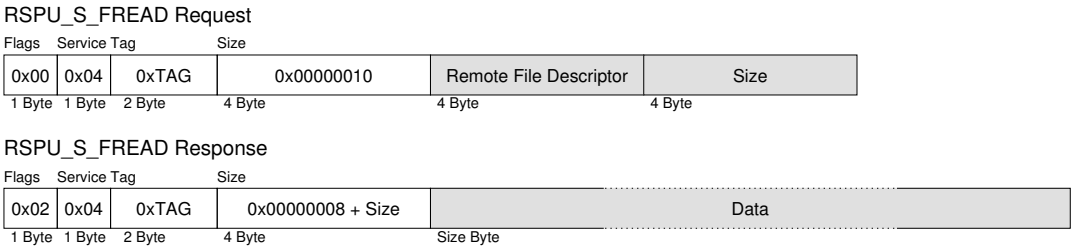

Figure 4.7: RSPU_S_FREAD message

RSPU_S_FREAD (figure 4.7) is a generic service to read from any file provided by SPUFS. Of cause, this file must have the ”r” permission (confirm table 3.1 on page 22). The `read` is performed on the requested file descriptor with the specified size. All restrictions have to be respected, for example a read from the mailboxes is only possible in a quantity of four bytes.

RSPU_S_FPREAD

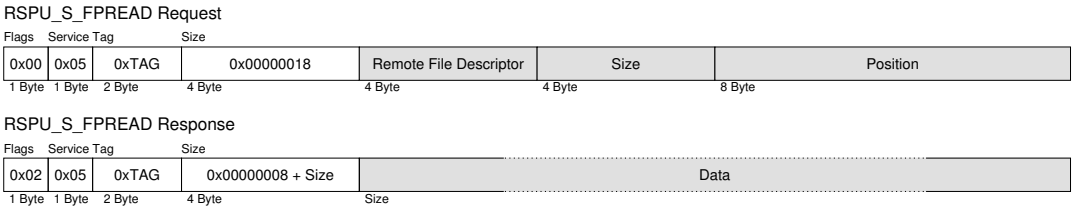

Figure 4.8: RSPU_S_FPREAD message

RSPU_S_FPREAD (figure 4.8) is nearly the same as RSPU_S_FREAD. The only difference is the read on a specific position in the file.

RSPU_S_FWRITE

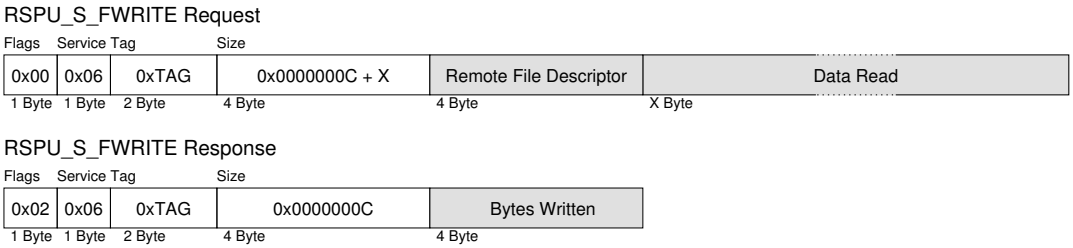

Figure 4.9: RSPU_S_FWRITE message

This service is the analogue to RSPU_S_FREAD, but the `read` system call is exchanged with `write`. Again, all restrictions on the files within the context has to be respected.

The amount of the written data is returned by this service (figure 4.9).

RSPU_S_FPWRITE

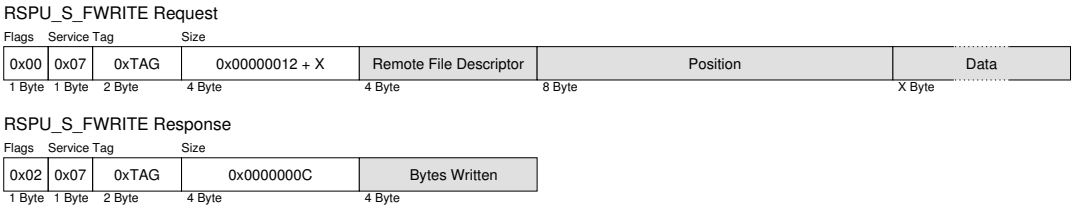

Figure 4.10: RSPU_S_FPWRITE message

RSPU_S_FPWRITE (figure 4.10) is the `pwrite` system call version of RSPU_S_FWRITE. As in RSPU_S_FPREAD it is possible to specify the position for writing within the file.

The services RSPU_S_FPREAD and RSPU_S_FPWRITE are intensively used by RSPUFS for reading and writing whole local store pages within the software emulated DMA support.

RSPU_S_FLLSEEK

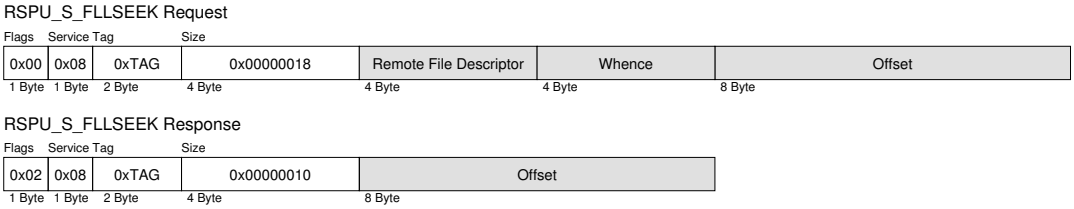

Figure 4.11: RSPU_S_FLLSEEK message

With RSPU_S_FLLSEEK (figure 4.11) it is possible to change the offset of the file descriptor.

This service is not often used. It is only provided for compatibility reasons.

RSPU_S_SYSRUN

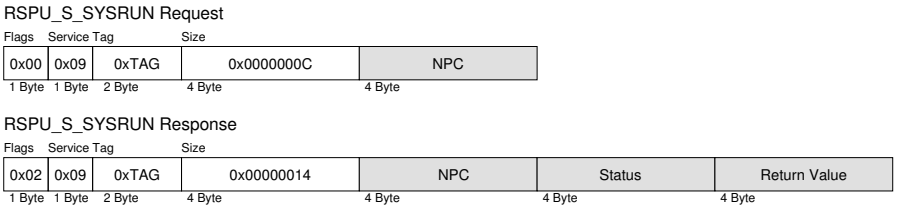

Figure 4.12: RSPU_S_SYSRUN message

RSPU_S_SYSRUN (figure 4.12) starts the execution of the SPU code. The parameters and return values match exactly those of the `spu_run` system call.

The processing of this service is completely different to the other. Because the system call is blocking, it has to be executed in a separate thread to avoid the stall of the whole daemon. If `spu_run` returns, the thread will be destroyed and the results will be transmitted back to the Opteron. In the meantime other services can be served by *rspufsd*.

RSPU_S_XALLOC

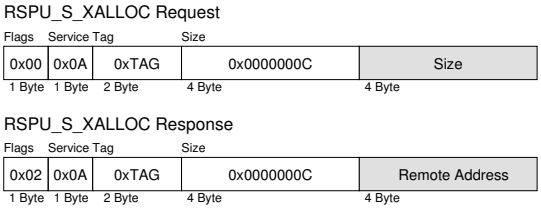

Figure 4.13: RSPU_S_XALLOC message

RSPU_S_XALLOC (figure 4.13) is the first service that does not match any functionality provided by SPUFS. The purpose of this service and the next three services is to provide an Opteron application the ability to access the main memory of the Cell system.

To allocate the XDR memory, RSPU_S_XALLOC has to be used. This service is associated with the current context. If the context becomes destroyed, the memory will be freed. The binding to the context is a simplification to keep track of the usage of the memory. The allocation is available for all SPU contexts created by RSPFUS. Thus the memory can be shared by multiple SPEs, which is needed for collective operations like matrix multiplication. The disadvantage is, that SPEs from other Opteron applications can use this memory, too, if they get the pointer to it. This behavior is not possible on SPUFS, because each application has their own address space. But in RSPUFS every program of the Opteron is represented by the daemon with only one address space. This problem could be solved with forking after each new connection. Thereby each child process gets its own address space with the disadvantage, that the SPEs cannot use the main memory together.

The only way to really solve this problem is the migration of *rspufsd* into the kernelspace.

RSPU_S_XFREE

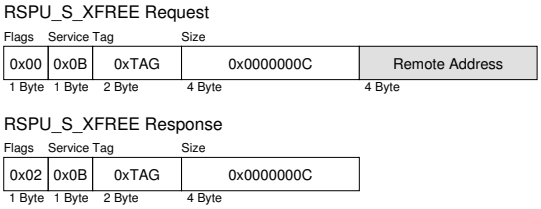

Figure 4.14: RSPU_S_XFREE message

The whole allocated memory is freed by using the RSPU_S_XFREE service (figure 4.14). It is not possible to free only a part.

The application programmer has to ensure, that no SPE is using this memory anymore. Any access to the released memory results in a segmentation fault.

RSPU_S_XREAD

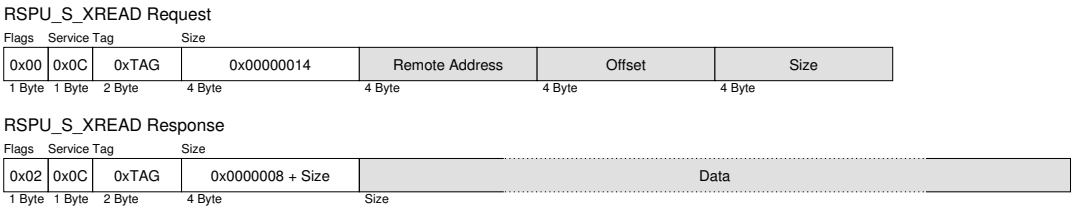

Figure 4.15: RSPU_S_XREAD message

With RSPU_S_XREAD (figure 4.15) the allocated memory can be written from the Opteron. If the combination of ”Offset” and ”Size” was outside the allocated region, the daemon would return an error.

RSPU_S_XWRITE

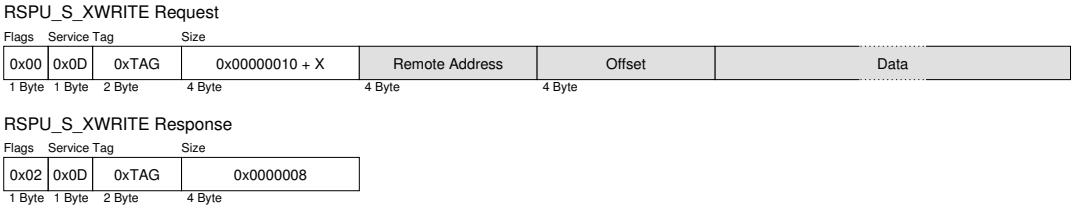

Figure 4.16: RSPU_S_XWRITE message

Last but not least, the service RSPU_S_XWRITE (figure 4.16) allows the client to write to the allocated memory.

4.4 The Opteron Part: rspufs

As top half of RSPUFS the kernel driver *rspufs* provides the VFS interface to access the SPEs from the Opteron. The main task is to provide the same behavior as SPUFS on the Cell.

In contrast to the daemon, which can use the Cell like any other application, the only way for the Opteron to use the SPEs is through the provided services from *rspufsd*. However, the big challenge is the different byte order of both systems. The Opteron has Little- and the Cell Big-endian. Thus the bytes have to be swapped after receiving and before sending inside the kernel driver.

Because the kernel cannot make any assumptions on the type of the data the application wants to access, the application has to order the bytes itself.

In this section the internal context and the management of network packages are depicted first, followed by the changes/extensions of the SPUFS interface. Thereafter the software emulated Cell memory access is described in subsection 4.4.4.

4.4.1 Internal Context

```
struct rspu_context
{
        struct socket *          sock;
        struct semaphore         comm_sem;
        struct kref              kref;
        spinlock_t               tag_lock;
        uint16_t                 tag_next;
        uint8_t                  want_close;

        struct mem_mmap
        {
                void *                      buffer;
                unsigned long               offset;
                struct rspufs_file_info *   fi;
                spinlock_t                  lock;
                int                         syncd;
                struct inode *              inode;
        } mem_mmap;

        struct xdr_mmap
        {
                void *                      buffer;
                unsigned long               offset;
                uint32_t                    size;
                spinlock_t                  lock;
                int                         syncd;
                struct inode *              inode;
                int32_t                     remote_addr;
        } xdr_mmap;

        int                      dispatcher_id;
        struct semaphore         dispatch_sem;
        struct list_head         dispatch_list;
        struct completion        dispatch_complete;
};
```

Figure 4.17: The `spu_create` system call

Most of the internal *rspufs* context (figure 4.17) consists of network specific parts like the TCP/IP socket ”sock” or the Packet Dispatcher entries. Other important entries are ”mem_mmap” and ”xdr_mmap”, which are used to support memory mapping.

4.4.2 Packet Dispatcher

One core component is the Packet Dispatcher, which is running as a separate kernel thread. Its task is to receive and deliver messages from the *rspufsd* as fast as possible. Receiving a message in the stream orientated TCP/IP is not straight forward. The `recv` operation on a socket does not always return the requested size. In most

cases it is necessary to recall it to get the rest of the bytes. The explanation of this phenomenon is, that this kernel returns back the current buffer content. When there are not all bytes arrived or the message is just to large to fit completely into the buffer, only the existing bytes are returned.

Thus the receiving implementation has the following steps:

1. Receiving until the eight header bytes are arrived.
2. Transform the values in the host byte order.
3. Receiving until the message is completed.

If the message is a response to a previous request, it will be put in a queue implemented as a simple list. The function sending the request traverses this list, searching for the matching *Tag* and *Service* (confirm figure 4.2). If the message is found, it will be deleted from the queue. These functionalities are implemented in the function `comm_getpack`, which is called by `comm_send`.

4.4.3 User-Level Interfaces

System Programing Interface

The system programming interface is nearly the same as in SPUFS. An application has to create the contexts with `spu_create` and executes them with `spu_run`, too. The creation results in a new context directory in the *rspufs* root with the content shown in table 4.1. One difference to the SPUFS entries (confirm table 3.1) is a new entry called “xdr“. Another is the semantic of “mem“. In the next points they are explained:

- **mem**

  The usage is exactly the same as in SPUFS, but the memory mapping is software emulated (confer section 4.4.4), because TCP/IP provides no RDMA features.

  In consequence the behavior is a little bit different to a hardware based support. First there is no coherency of the mapping. That means, if the Opteron is writing on the same local-store region as the Cell, the next write back will overwrite all the changes made by the Cell. This cannot be influenced by

File	Perm		Description
decr	r	w	SPU Decrementer
decr_status	r	w	decrementer status
event_mask	r	w	Event mask for SPU interrupts
fpcr	r	w	floating point status and control register
mbox	r	-	the first SPU to CPU communication mailbox
mbox_stat	r	-	length of the current queue
ibox	r	-	the second SPU to CPU communication mailbox
ibox_stat	r	-	length of the current queue
wbox	-	w	CPU to SPU communication mailbox
wbox_stat	r	-	length of the current queue
mem	r	w	local-store memory
npc	r	w	next program counter
regs	r	w	register file
signal1	r	w	signal notification channel 1
signal1_type	r	w	behavior of the signal1 (replace or "OR")
signal2	r	w	signal notification channel 2
signal2_type	r	w	behavior of the signal2
srr0	r	w	interrupt return address register
xdr	r	w	main memory interface

Table 4.1: the RSPUFS context (part)

the application, because it happens implicitly before each `spu_run` or when accessing a page different the current one.

However, the application should only operate with the local-store memory mapping, when the SPU is not running.

- **xdr**

  Running on the Opteron, an application cannot access the main memory of the Cell. Thus a special interface is provided by RSPUFS.

  After opening the "xdr" entry, it is possible to `mmap` it in shared mode into the application address space. With the *size* parameter of this system call, the amount of memory can be set. There is no limitation, but any request higher than the physical amount of memory on the Cell results in swapping.

  Of cause, the address returned by `mmap` is only the address on the Opteron. To get the Cell equivalent an `ioctl` with the parameter "RSPU_GETADDR" has

to be performed.

Unlike the implicit synchronization of the local-store mapping, the application can do it explicitly by calling `ioctl` with the parameter "RSPU_SYNC".

The main memory access is software emulated (section 4.4.4), too. Thus there is also the restriction, that Cell and Opteron could not work on the same page without the loss of data.

- "**generic**"

  All the other entries are handled by generic functions inside the VFS implementation of RSPUFS. They are transforming the parameters into an RSPUFS message requesting the appropriate service. On the Cell side the *rspufsd* delegates it to the SPUFS.

SPE Runtime Management Library

To support the XDR mapping the library interface was improved with a new set of functionalities:

1. `void * xdr_mmap(spe_context_ptr_t spe, void * start, size_t length, int prot)`

   XDR memory could be requested with this function. The return value points to the mapping in the application address space, which is not equal to the address on the Cell. The Cell address could be requested with `xdr_getaddr`.

   The mapping could neither be extended nor reduced later. When another amount is required, it has to be freed and newly mapped. It is only allowed to have one mapping per context.

2. `int xdr_unmap(spe_context_ptr_t spe)`

   If the memory is not used anymore, the application had to call `xdr_unmap`.

3. `int xdr_getaddr(spe_context_ptr_t spe)`

   The address of the XDR memory mapping on the Opteron differs from the address of the allocated memory on the Cell. The address on the Cell can be requested by calling `xdr_getaddr`. If this address is passed to the SPE program, it will be able to access the XDR memory, too.

4. `int xdr_sync(spe_context_ptr_t spe)`

   A call of `xdr_sync` synchronizes the mapping explicitly.

Toolchain

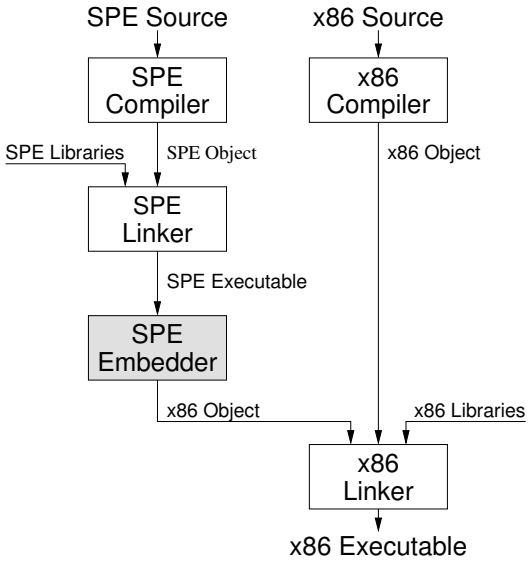

Figure 4.18: RSPUFS: Tool Chain

In figure 4.18 the toolchain is displayed, which produces an x86 executable with integrated SPU code. Therefore the original Embedder (figure 3.6 on page 28) was modified to produce x86 object code.

4.4.4 Software Emulated Remote Memory Access

The possibility to map the local-store into the local address space is very important, because the libspe uses this feature extensively. It is not only desirable to memory map the local-store, but the XDR main memory, too. This is necessary to make the operands, for example a huge matrix, accessible through the Cell. With TCP/IP no hardware support is available to provide this functionality on the Opteron. Thus it has to be emulated in software by RSPUFS.

Therefore the internal RSPUFS context has two structures: `struct mem_mmap` and

`struct xdr_mmap` (confirm figure 4.17 on page 45). If the application calls the `mmap` system call, the page tables will be configured, but without memory assignment. This is called "Demand Paging" [12]. Demand Paging follows a lazy evaluation (or lazy loading) strategy. If a page fault occurs, the page will bounded on physical memory. RSPUFS uses this mechanism. The physical memory is represented by *buffer* with exactly the length of one page. When a page fault occurs, the *buffer* is mapped to the faulting page and the previous mapping becomes destroyed. Thus only one page is physically represented all the time. This behavior was chosen, because it is the only possibility to minimize the amount of Cell memory mirrored on the Opteron. This is very important, because in view of the Cell the *buffer* is like a none coherent cache. Any data written to the memory region the Opteron is currently working on, results in the loss of this data.

The write back of the *buffer* happens, when:

- local-store mapping:
  - accessing a page other than the current one,
  - `spu_run` is called,
- XDR mapping:
  - accessing a page other than the current one,
  - the application calls the RSPU_SYNC `ioctl`.

Thereafter the buffer gets unmapped. Thus any access to the memory results in a page fault. This has the advantage, that in meantime the Cell can modify it without the loss of data. If the next page fault occurs, the Opteron will get the actual version of the memory.

5 Results

This chapter shows the results of the diploma thesis. The implementation status of RSPUFS will be described in 5.1. Whether the SPUFS concepts could be respected, will be depicted in 5.2. Last but not least, some benchmarks are shown in section 5.3.

5.1 Implementation

The migration process is nearly completed in the current RSPUFS implementation. Missing features are the `poll` system call and the memory mapping support of the problem-state area and the MFC multi-source synchronization facility, but these are not necessary for the libspe, because alternative code inside the library will be used, when it is not possible to memory-map these areas. Ony the function `spe_read_tag_status_noblock` is not available, because `poll` is used their.

The porting from libspe is far away from completion. Currently the following parts are ported to the Opteron:

1. Program loading
2. Program execution
3. Mailboxing
4. POSIX.1 assisted Callbacks

   All calls using the main memory (confer table 3-7 [10]) are not supported, because it is not possible to access the Opteron memory from the Cell side. For example, if the `mmap` is mapping a file into the main memory, the Cell could never access it.

However, the current implementation allows to execute simple SPU-codes on the Opteron.

5.2 The SPUFS Concepts

This section examines the compliance of the SPUFS concepts to RSPUFS.

5.2.1 Virtual File System

As shown in table 4.1 on page 47 the VFS interface was ported to the Opteron. Except the functions mentioned in the last section, the RSPUFS provides the fully operational SPUFS interface.

Moreover, a method for accessing the main memory of the Cell is available, which is very important for most of the applications to exchange the operands.

5.2.2 Virtualization

Even though virtualization is an important concept of SPUFS, it was never mentioned in the RSPUFS chapter. The simple reason is, that RSPUFS is not dealing with it. Nevertheless this concept is supported by RSPUFS implicitly.

The virtualization is completely done by the native SPUFS on the Cell. Double virtualization would happen if RSPUFS would virtualize the SPEs, too. Thus scheduling issues can occur, when the Opteron suspends a context, which the Cell is resuming.

However, the current approach creates the illusion, that any desired amount of physical SPEs is present.

5.3 First Benchmark Results

5.3.1 Software Emulated Memory Access

To benchmark the software emulated memory access, a program executed on the Opteron copies an array from the main memory into the mapped Cell memory. In figure 5.1 the result is shown. The axis of abscissas shows the copied size in bytes and the axis of ordinates the required time in micro seconds. The time measurement includes only the time for the `memcpy` function call.

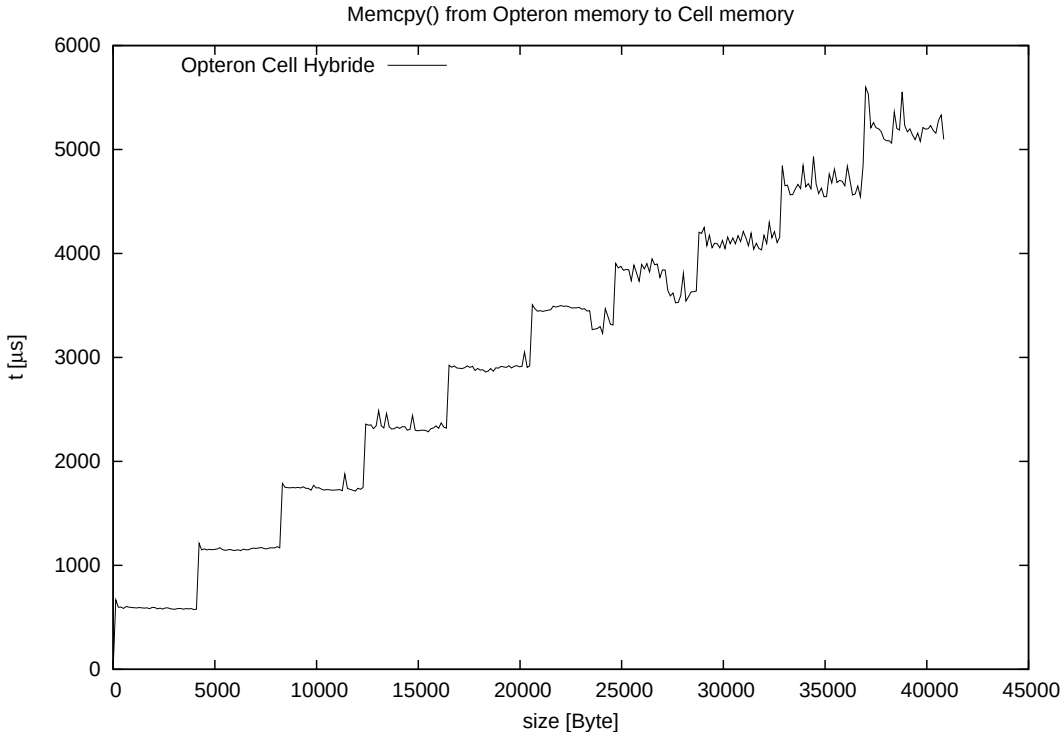

Figure 5.1: Copy memory from Opteron to the Cell

The graph possesses steps of approximately $0.55ms$ every 4096 Byte. This is the time for one page fault.

5.3.2 Adding Two Integers

In figure 5.2 and figure 5.3 the results of the addition of two integers ($a + b = z$) are shown. The addition was executed 1000 times (axis of abscissas) on the Opteron (black line) and 1000 times on the Cell (gray line). The measured time in micro seconds was printed on the axis of ordinates.

In the first figure the parameters are transmitted with the mailbox functionality. That means two sending operations (a and b) and one receiving operation (z).

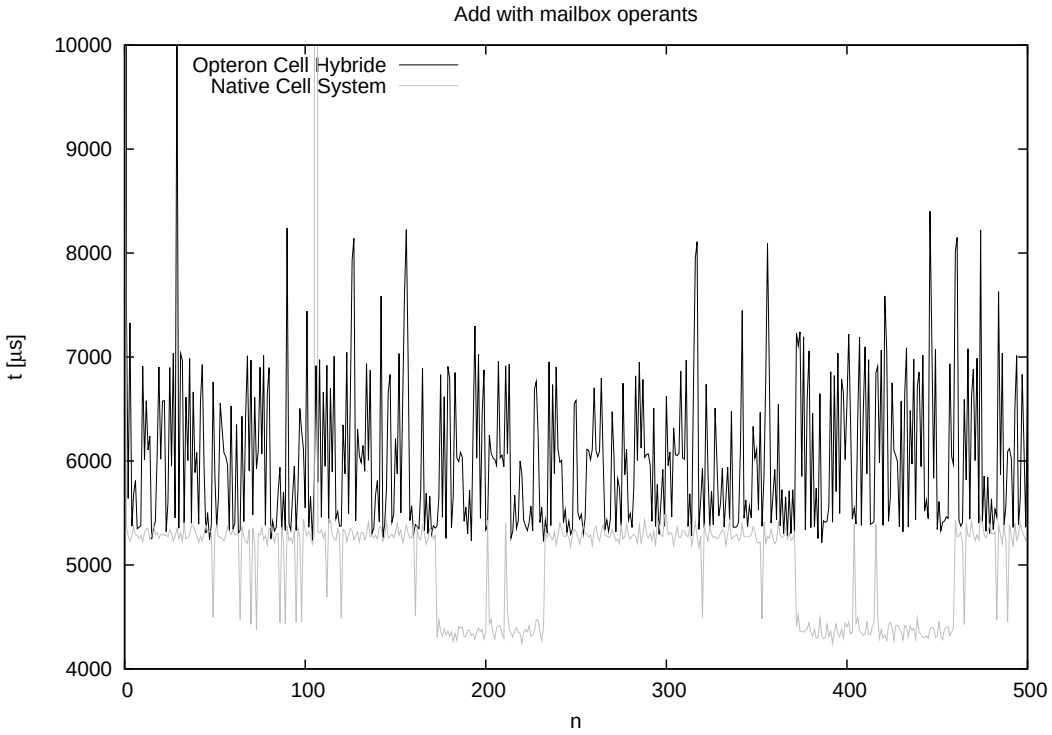

Figure 5.2: Adding two numbers with mailbox parameter transmission

On average the Hybrid needs $6.1ms$ and the Cell $4.4ms$ for loading the program code, creating the context, transmitting the operands and performing the addition operation on the SPE. The timer is set to 1000 HZ on both systems. Thus a peak of one milli second is equal to one schedule.

In the second figure (5.3) the operands are transfered through the XDR main memory.

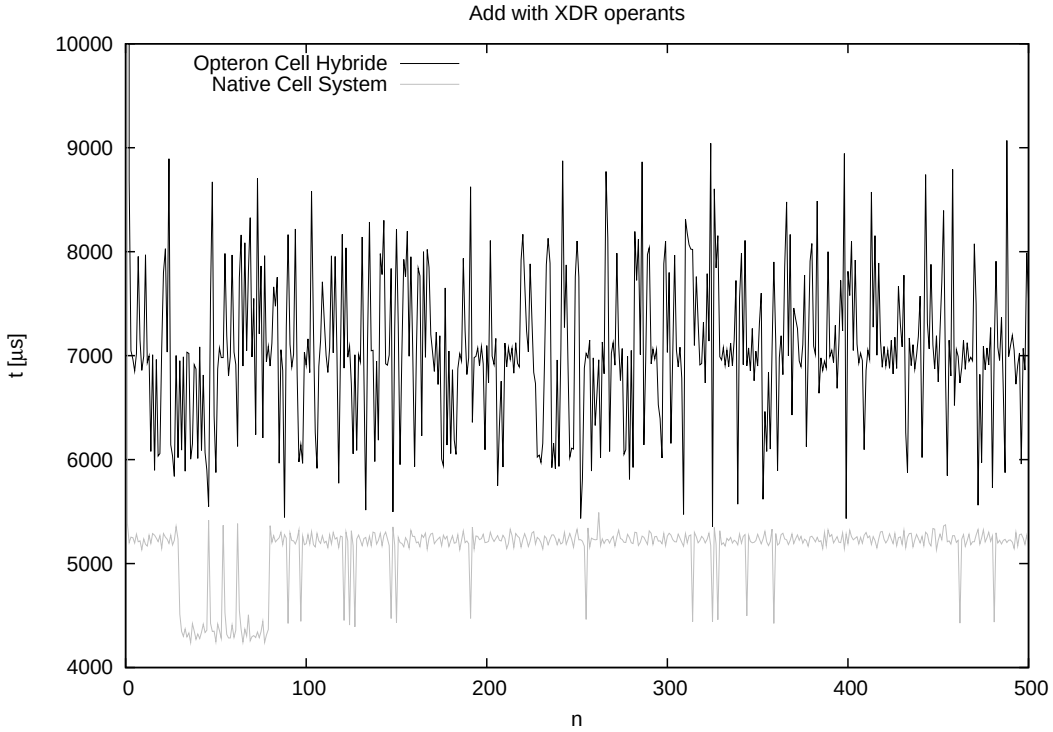

Figure 5.3: Adding two numbers with XDR parameter transmission

The average values are $7.0ms$ for the Hybrid and $4.4ms$ for the Cell. In the XDR case occurs three page faults, but in the mailbox case only one. Thus the mailbox version possesses a better performance when running under RSPUFS.

Measurements have shown that a service request took $0.15ms$ in average. The XDR program version require 12 service requests and the mailbox version 19 service requests, both without all page faults. If the page fault time and the service time is subtracted, the result will be:

mailbox case: $6.1ms - 1 \cdot 0.55 - 19 \cdot 0.15 = 2.70ms$
XDR case $7.0ms - 3 \cdot 0.55 - 16 \cdot 0.15 = 2.95ms$

That shows, the performance is nearly equal. Thus the page fault is the limiting

factor. The remaining time includes the loading of the program, the calculations of libspe and the service processing inside the Opteron kernel.

However, what do these results mean for the PCIe coupling? To answer this question, the following points have to be noticed:

1. With PCIe the "rspufsd" is no more needed, because all parts necessary for managing the SPE can be memory-mapped into the Opteron address space.
2. PCIe has a lower latency than Ethernet.

The latency of the Gigabit Ethernet TCP connection is $0.05ms$[1]. Three quarter of RSPUFS is managing packages. Thus the assumption can be made, that a service needs $0.025ms$ calculation time. Together with the assumption, that PCIe has a latency of less than a half of TCP, a service will take $0.05ms$ ($0.025ms + 0.25ms$).

Back to the addition of the two numbers, the expected PCIe execution time would be:

mailbox case: $6.1ms - 1 \cdot 0.55 - 19 \cdot 0.05 = 4.60ms$
XDR case $7.0ms - 3 \cdot 0.55 - 16 \cdot 0.05 = 4.55ms$

To sum up, with PCIe it should be possible to shrink or eliminate the differences in the execution time between Opteron and Cell completely.

[1] average value measured with NetPIPE (`http://www.scl.ameslab.gov/netpipe/`)

6 Conclusion and Further Work

This thesis shows the way for integrating the Synergistic Processing Elements of the Cell into the Opteron system. The Opteron is the main processing unit, supported by the SPE accelerator components. Ethernet (TCP/IP) was chosen as interconnect network, because it was a simple and available connection method at the beginning of this thesis. One key feature of the used TCP/IP protocol is the similarity to a PCIe link, which replaces the Ethernet in the future.

RSPUFS was implemented along this diploma thesis to realize the software integration of the SPEs into the Opteron Linux environment. RSPUFS is the migration result of SPUFS, the driver integrating the SPEs on the native Cell. The SPUFS concepts, which consist of virtulalization of the SPE and the VFS approach to access the components of the SPE, are preserved by RSPUFS.

The first test results show, that the current performance of RSPUFS is up to 72% of SPUFS for adding two integer numbers. This value includes the whole program execution time consisting of context creation, loading the SPU-code to the Opteron, submitting the arguments and the execution of the addition on the Cell. The performance decreases with the increased size of the operands, because multiple page faults on the Opteron will occur. It is shown, that with the availability of a RDMA capable interconnection (like PCIe) the performance could be 98% or even more.

However, RSPUFS successfully integrates the SPE into the Opteron Ecosystem.

Further Work

The further work around RSPUFS includes:

1. To complete the porting of libspe to the Opteron.

2. Extend the RSPUFS-protocol to support a daemon and libspe running in 64bit-mode.

3. Port the *rspufs* kernel driver to Linux-2.6.24, where the architecture branches of "x86" and "x86_64" are merged.

4. Use direct PCIe coupling between Opteron and Cell.

Within the development of RSPUFS, the idea was born to use the SPUFS concepts to integrate other types of accelerators into the Linux environment. Currently every vendor uses his own proprietary interface. One example is the Tesla[13] GPGPU (General Processing Graphic Processing Unit) from NVIDIA, which is currently only programmable through CUDA[14]. CUDA is a C language development environment for NVIDIA graphic accelerators including a compiler, drivers and a runtime library.

However, if an application requires different accelerators, it will have to use different interfaces and libraries. Thus it is desired to have a common interface for all accelerator types. In a seperate work it has to be prove, that it is possible to use the SPUFS concepts for other accelerators creating a common Accelerator File System (ACCFS).

Bibliography

[1] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[2] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[3] Meet the experts, david krolak on the cell broadband engine eib bus. `http://www-128.ibm.com/developerworks/power/library/pa-expert9/`.

[4] Ibm doubles down on cell blade, press release. `http://www-03.ibm.com/press/us/en/pressrelease/22258.wss`.

[5] Spursengine. `http://en.cellusersgroup.com/modules/spursengine/`.

[6] Roadrunner project. `http://www.lanl.gov/orgs/hpc/roadrunner/index.shtml`.

[7] John A. Turner et al. Roadrunner applications team: Cell and hybrid results to date. Technical report, Los Alamos National Laboratory, October 2007.

[8] Arnd Bergmann. *The Cell Processor Programming Model*. LinuxTag, June 2005.

[9] *SPE Runtime Managment Library Version 2.2*. CBEA JSRE Series. IBM Corporation, September 2007.

[10] *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Version 1.2*. CBEA JSRE Series. IBM Corporation, October 2007.

[11] *Synergistic Processor Unit Instruction Set Architecture Version 1.2*. IBM Corporation, January 2007.

[12] Andrew S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[13] Nvidia high performance computing solutions. `http://www.nvidia.com/object/tesla_computing_solutions.html`.

[14] Nvidia cuda project. `http://www.nvidia.com/object/cuda_home.html`.

A CD-ROM Contents

The diploma thesis includes a CD-ROM with the following contents.

doc/

This directory contains an electronic copy of the diploma thesis in PDF and PostScript format.

testapps/

- addlibspe2.tar.bz2

  Addlibspe2 is the test application, which is used to add two numbers with operants submitting through the mailbox functionality.

- addlibspe2-xdr.tar.bz2

  This is the pendant to "addlibspe2.tar.bz2", but with usage of the XDR-interface to share the operants.

- common.tar.bz2

  All programs using the Makefile-Definitions and the common header of this package.

- xdrcpy.tar.bz2

  This is the application used for the measuring of the page fault time.

- tools.tar.bz2

  Generic tools for measuring execution timings of a whole program are included in this archive.

toolchain/

In this directory the modified Embedder and the migrated libspe can be found.

RSPUFS/

This folder contains the sources of the *rspufsd* and a patched Linux kernel (version 2.6.22) with integrated *rspufs* file system driver.

Thesis Declaration

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography. I declare that it has not been submitted in whole, or in part, for any other degree.

Chemnitz, March 11, 2008

Andreas Heinig