Generalizing the SPUFS concept – a case study towards a common accelerator interface

Andreas Heinig[1], René Oertel[1,2], Jochen Strunk[1], Wolfgang Rehm[1], Heiko Joerg Schick[2]

[1]Chemnitz University of Technology, Germany
{heandr,oere,sjoc,rehm}@cs.tu-chemnitz.de

[2]IBM Deutschland Entwicklung GmbH, Germany
{oertelr,schickhj}@de.ibm.com

This research is supported by the Center for Advanced Studies (CAS) of the IBM Böblingen Laboratory, Germany in collaboration with the NICOLL Project.

Abstract

The development of specialized application accelerators is happening today. However, they do not share a common attach point, and have no common architecture or programing model. A framework that economically and efficiently enables specialized acceleration is highly desirable.

In this work we propose a generic interface concept called "ACCFS" for integrating application accelerators into Linux-based platforms. The idea is to extend the programing model chosen by the Linux for Cell/B.E. team. On the Cell/B.E. multiple independent vector processors called Synergistic Processing Units (SPUs) are built around a 64-bit PowerPC core (PPE). The programing model is to create a virtual file system (VFS) to export the functionality of the SPUs to the user space via a file interface, they called it "SPUFS". Against other solutions such as using character devices or introducing a new process space the VFS interface uses common file system calls and provides an economic and efficient access to the accelerator units, the SPUs. Together with this concept and the experiences from an intermediate step called "RSPUFS" we introduce the first approaches towards a common accelerator file system (ACCFS).

1 Introduction

Utilizing accelerators in the field of research, the financial sector, the industry or the embedded market is a common technique to overcome limitations of standard processors (CPUs). The requirements can vary from overall floating point peak performance, run time, power consumption, power efficiency, form factor to real-time behavior. Application specific accelerators such as GPUs, FPGAs, DSPs and specialized streaming engines for encoding and decoding are used to satisfy these demands. Almost all of these accelerators are part of a bigger computing system, i.e. they are attached into computing nodes using pluggable add-in cards, socket add-in modules or other internal or external connections to processor and peripheral buses.

What these systems have in common with accelerators is the need of integrating the hardware into the operating system and providing an API for application software. Today each vendor provides its proprietary accelerator specific device driver and optimized application library.

A common, hardware independent framework and interface tightly integrating distinct accelerators into operating systems is missing so far. A hardware independent solution offerers the chance to simplify the use of different accelerators at the same time, switching from one accelerator to another and ease the development of accelerating libraries.

2 Related Work

With the growing field of accelerator appliances, different kinds of problems appear when using accelerators for offloading computational intensive calculations. Several well-known companies and organizations take approaches to fulfill requirements both of application and operating systems programmer. These approaches deal with different layers of the abstraction of the bare metal. We distinguish between the programming model, the libraries usable by the application programmer, the management layer in the operating system and the extensions of the hardware.

The representatives for the programming model are for example the Mitrion-C [8] and the StreamIT [12] programming languages. Their task is to provide an accelerator aware programming environment which deals with the translation of application code to hardware dependent code without the need of the programmer to have a deeper knowledge of the underlying hardware. These environments focus on the usage of the accelerators, e.g. FPGAs, but not on their management in particular.

There are already some libraries, which provide man-

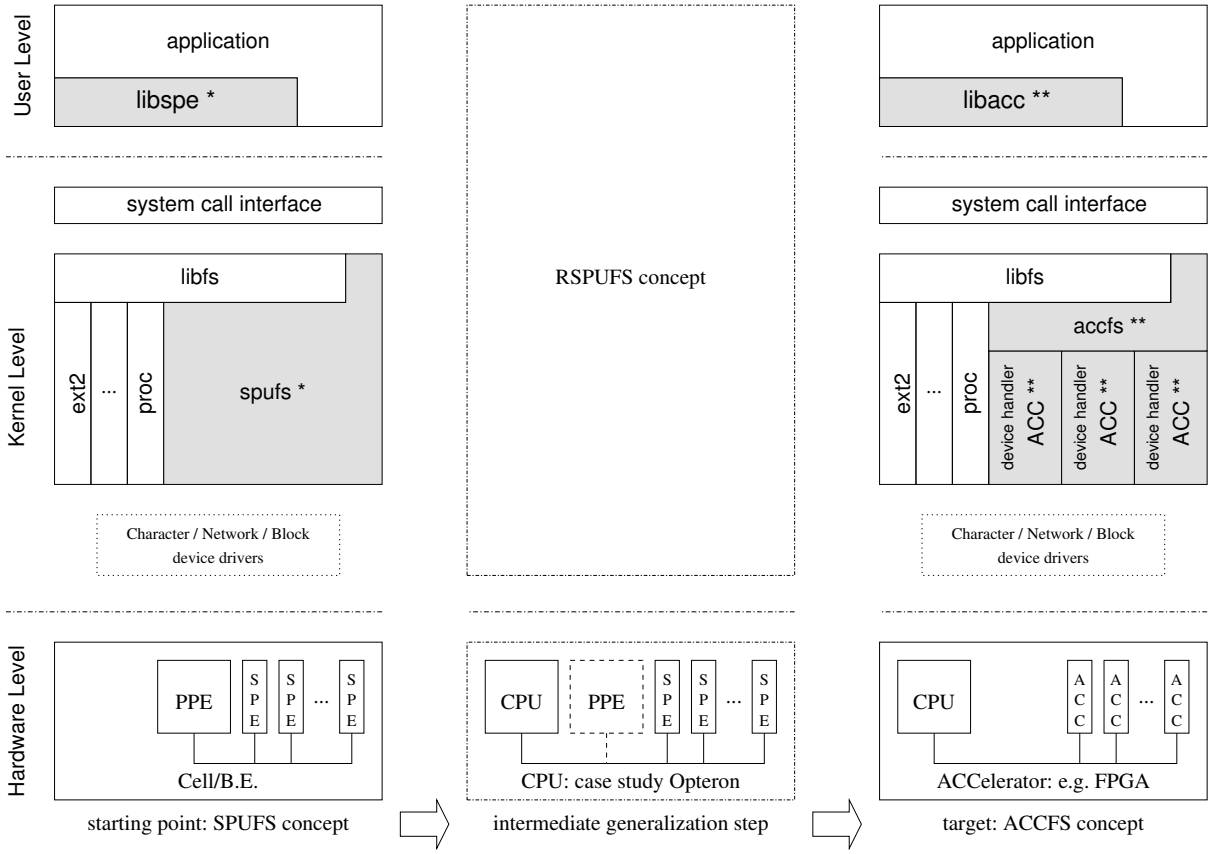

Figure 1: Extending the SPUFS * concept to ACCFS **

agement and control facilities of accelerator entities. The IBM SPE Runtime Management Library [6], the Intel QuickAssist Technology Accelerator Abstraction Layer (AAL) [4] and the proposal of the OpenFPGA GenAPI [2] are some examples of such libraries. Their objective is to provide a hardware independent and consistent layer for application programmer, which saves the user application development investments and eases the porting of already available code. The NVIDIA CUDA™ [9] development environment is a special case of an accelerator framework, because it coalesces a GPU specific compiler, application libraries and a runtime driver. Thus it provides a multilayer solution for NVIDIA GPU accelerators.

At the operating system layer there are well established and shortly introduced concepts for managing similar hardware entities. Typical examples are the management of block devices or WLAN host adapters (MAC80211, [11]) in the Linux operating system, but there is no generic accelerator framework, yet. The approach is to define a common set of functions needed for a uniform access which is provided by the low-level device drivers and registered by them in the framework.

Accelerator specific hardware extensions [7] which focus on the tightly coupling of such computational offload engines are for example the Intel Geneseo Technology [5], the AMD Torrenza initiative [1] and the PCI-SIG PCI Express I/O Virtualization [10] drafts. The challenge on these extensions is to provide low-level enhancements, which are requested of higher layers of our abstraction model, e.g. the support of virtualization and atomic operations.

3 Basic Idea

We have had the idea to check out whether the virtual file system approach of SPUFS could be adopted successfully to a more generic coupling of a CPU and accelerators. Therefore we have chosen a step-by-step porting. In a first step we replaced the PPE by a commodity main stream CPU – in that case AMD Opteron. A further step will be the substitution of the SPUs by other specialized accelerators such as FPGAs and the like.

Figure 1 illustrates the stepwise generalization starting with the SPUFS concept followed by an intermediate step ("RSPUFS") that finally leads to the ACCFS concept.

4 Cell Broadband Engine Architecture

The Cell Broadband Engine (Cell/B.E. or Cell) Architecture was developed in corporation of Sony, Toshiba and IBM (”STI”). The goal was to develop an architecture for the next generation of entertainment devices, especially the PlayStation 3 from Sony. In order to extend the applicability of the Cell and to start a developing community, an open source development environment based on GNU/Linux has been released.

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 shows a complete picture about the Cell. The components are explained on the following pages.

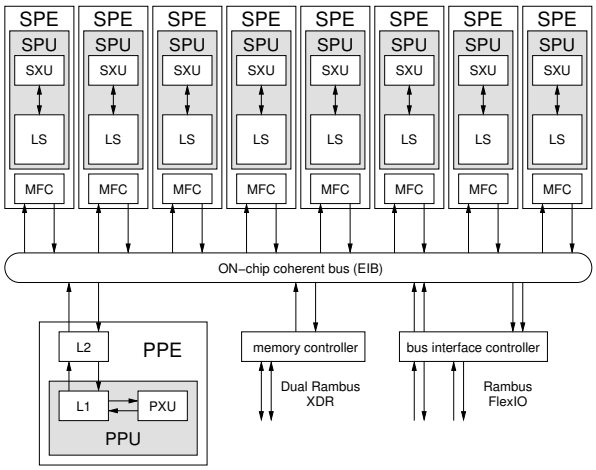

Figure 2: Cell/B.E. Block Diagram

Power Processor Element

The 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. The instruction set architecture (ISA) is nearly the same as on other 64-Bit PowerPC based processors. Thus the GNU/Linux operating system can be executed unchanged on the PPE. Only modifications on the device drivers were necessary to support the components of the main board.

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. The program code and all (local) data must fit into the LS. To access addresses outside the LS the DMA engine, represented by the Memory Flow Controller (MFS), has to be used. The MFC processes the DMA asynchronously to the SPU-program execution.

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 SPE is the acceleration unit of the Cell system. The most parts of the SPE are exported via the memory. Thus it is possible to map these into the application.

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 SXU, 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. The MFS is mapped into the Cell address space, too.

Memory and Interface Controller

On the die a Memory Controller and a Interface Controller is located, too. The Interface Controller provides a high bandwidth flexible I/O interface (FlexIO) and is splittable into two logical controllers.

Element Interconnect Bus

The twelve elements (1xPPE, 8xSPE, 1xMC, 2xIC) are connected with a 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.

5 SPUFS

The operating system is running on the PPE. To include also the SPE in the Linux environment the Synergistic Processing Unit File System (SPUFS) is used. SPUFS is based on two concepts: **virtualization of the SPE** and **virtual file system context access**.

5.1 Virtualization

Virtualization is necessary to avoid resource conflicts on a multi-user system, when multiple applications are trying to acquire the SPEs.

SPUFS manages contexts of virtual SPEs. A context contains all data for suspending and resuming SPU program execution, like the register file, local store or the MFC status. The context gets bounded on a physical SPE by a scheduler inside SPUFS. For the context access the following methods are conceivable:

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.

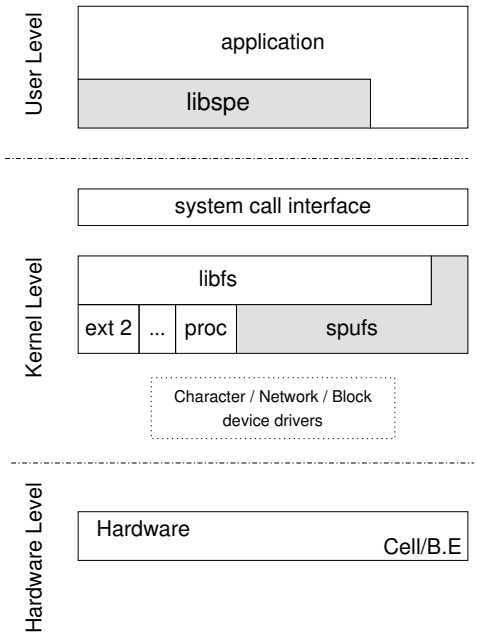

Figure 3: SPUFS Model

For the communication between user- and kernel-space, system calls like `open`, `read`, `write` or `mmap` are used.

5.2 Virtual File System

Because of the disadvantages of character devices or the new thread space, the Linux Cell/B.E. team has decided to implement a virtual file system. Every context entry is accessible thought a single file with the normal file system calls. The integration into the Linux kernel is shown in Figure 3. SPUFS is managed under ”libfs”, the VFS implementation of Linux, like any other file system. Additionally to the file system calls two new system calls are introduced with SPUFS (the left side of the ”spufs”-box):

1. **sys_spu_create**

   The first step for applications is to create a context for each SPE they require. Creating contexts with the system call `spu_create` is the only operation available on the root of the file system.

   This system call returns the handle to the context directory. When closing this handle, the context gets destroyed.

2. **sys_spu_run**

   After loading the program into the local-store memory the `spu_run` system call executes the SPU code. The execution is synchronous. Thus the calling thread is blocked, while the SPU code is running. Returning can have the reason, that the SPU program is finished or an error is occurred.

Creating the context results in a new directory under the SPUFS root. Operations on the files inside the context will directly modify the physical SPE if the context is bounded, otherwise the context safe area inside the main memory. In Table 1 some files are shown. The

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 queue
ibox	r	-	the second SPU to CPU communication mailbox
ibox_stat	r	-	length of the queue
wbox	-	w	CPU to SPU communication mailbox
wbox_stat	r	-	length of the 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 channel 1
signal1_type	r	w	behavior of the signal1 (replace or "OR")
signal2	r	w	signal channel 2
signal2_type	r	w	behavior of the signal2
srr0	r	w	interrupt return address register

Table 1: The SPUFS context (part)

most important are:

- **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.

- **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`. The mapping has the advantage, that the operating system is not involved by each access.

- **wbox / mbox / ibox**

  The Cell supports a "mailbox"-mechanism where short messages (4 byte) can be transfered between SPE and PPE.

- **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.

- **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

5.3 Runtime Management Library

To provide a further abstraction to the application, a SPE Runtime Management Library called "libspe" is provided. 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.

Libspe hides the VFS interface to the application, but in most cases this library is a wrapper around the SPUFS. The library provides the funtions:

- SPE context management
- CPU information
- SPE program image handling
- SPE run control
- SPE event handling
- MFC proxy command
- MFC multi-source synchronization
- MFC proxy tag-group completion
- SPE mailbox
- SPE signal notification
- Direct SPE access
- PPE-assisted library calls

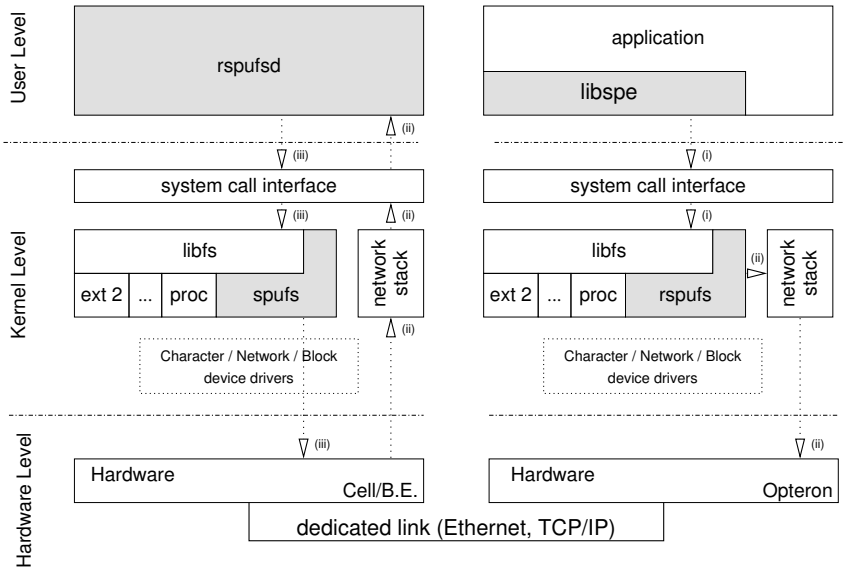

Figure 4: RSPUFS Model

6 RSPUFS

The first step towards a common accelerator file system was to prove that the PPE can be replaced by another processor. In our case we chose an AMD Opteron. As there was no hardware solution available that enables direct coupling of an Opteron processor with SPUs we use an Ethernet-based coupling as intermediate step. In conjunction with our Remote SPUFS (RSPUFS) implementation a direct coupling could be emulated.

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. A complete Linux system is running on the Opteron as well as on the Cell. Both are connected through the 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.

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 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).

6.1 The Cell Part: *rspufsd*

The whole RSPUFS concept works like a proxy. In Figure 4 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.

6.2 The Opteron Part: *rspufs*

The main task of the top half 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.

Another challenge was the unavailable RDMA (Remote Direct Memory Access) capable interconnection. 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.

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

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.

The software emulation results on some changes on the VFS interface:

- **mem**

  The usage is exactly the same as in SPUFS, but the semantics have changed. 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. Second, the synchronization of the memory happens implicitly before each `spu_run` or when accessing a page different the current one.

- **xdr**

  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.

  Unlike the implicit synchronization of the local-store mapping, the application can explicitly synchronize the xdr-mapping.

The runtime management library was also extended to support the new interface.

For a deeper look on the internals of RSPUFS please refer [3].

6.3 Results

We could prove that it is possible to cope with problems like byte ordering (endianes) and direct memory access (even emulated) without modifying the SPUFS concept.

Furthermore the RSPUFS implementation shows a way for integrating accelerators other than SPUs by splitting the functionality into two parts: one abstracting the user interface (rspufs) and one integrating the acceleration hardware (rspufsd). We propose exactly this structure for the ACCFS concept.

7 ACCFS

With ACCFS we try to combine the experiences gathered from SPUFS and RSPUFS. From SPUFS we use the concepts **virtualization** and the **VFS** approach and from RSPUFS the **separation** of the functionalities. Thus ACCFS consits of two parts (confer Figure 5):

1. Top half: **accfs**

   The Linux kernel driver of ACCFS is called "accfs". Its main task is to provide the user interface (VFS) and the vendor interface.

2. Bottom half: **device handler**

   Every vendor can integrate there specific device driver into the ACCFS frame work by writing special device handlers.

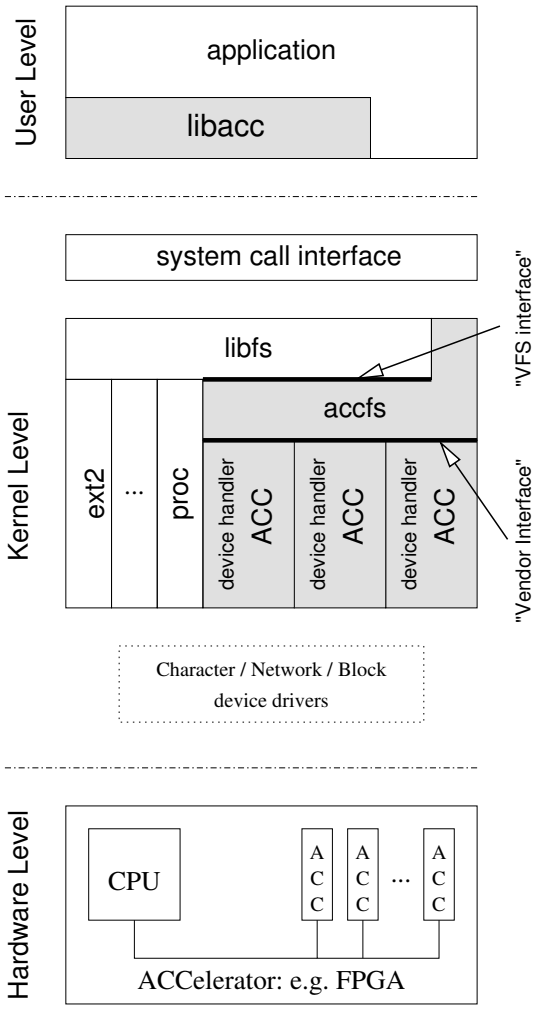

Figure 5: ACCFS Model

7.1 VFS interface

Like SPUFS the user interface consists of the VFS part and the two system calls `sys_acc_create` and `sys_acc_run`. These system calls have the same semantics as the SPUFS ones, except an additional parameter specifying the accelerator type the context has to create for.

In contrast to RSPUFS, where the VFS interface was adopted nearly unchanged, ACCFS uses other file sys-

File	Perm		Description
regs	r	w	register file
message	r	w	message interface between CPU and accelerator
memory/	r	w	folder containing all exported accelerator memory regions
semaphore/	r	w	folder which contains semaphores as basic synchronization primitives

Table 2: The current ACCFS context

tem entries. The current set is shown in Table 2. This set is our first proposal for the pretty basic things. The entries are described detailed in the following points:

- **regs**

  The register file of the accelerator is represented in this entry. Allowed operations are `read` and `write`.

- **messages**

  Short messages could be send (`write`) or received (`read`) with this entry. These operations are generally blocking, except *messages* is opened in non-blocking mode.

- **memory/**

  In contrast to the SPUFS context entries, the ACCFS context has a folder to represent the exported memory. In case of RSPUFS the entries would be ”ls” (for the local-store) and ”xdr” for the XDR memory mapping support. An FPGA maybe exports the connected DRAM or a configuration memory here.

  A memory entry supports `read`, `write` and `mmap`.

- **semaphore/**

  For the synchronization we plan to implement a semaphore mechanism, where the accelerator as well as the main application can access the semaphore with atomic operations. Possible operations within this directory are `create`, `unlink`, `read` and `write`.

Context entries will be only available if the accelerator supports the corresponding feature.

7.2 Vendor Interface

Initially is no accelerator available when ”accfs” is loaded into the Linux kernel. To use an accelerator the vendor specific driver handler has to be loaded (we call it register). For example, the Cell will be supported by the RSPU vendor driver.

The vendor interface consists of the two functions `accfs_register` and `accfs_unregister`.

1. **accfs_register**

   The parameter of this function is a pointer to the `accfs_vendor` structure as shown in Figure 6.

```
struct accfs_vendor
{
        int accelerator_id;
        int (*create)(...);
        int (*destroy)(...);
        int (*run)(...);

        int     (*sem_create)(...);
        int     (*sem_delete)(...);
        int     (*sem_write)(...);
        int     (*sem_read)(...);

        size_t (*message_send)(...);
        size_t (*message_recv)(...);

        int     (*memory_read)(...);
        int     (*memory_write)(...);
        int     (*memory_sync)(...);
};
```

Figure 6: `struct accfs_vendor`

   The accelerator identifier (*accelerator_id*) is used to address the accelerator from the user space. The accelerator type parameter of the `sys_acc_create` is exactly this identifier.

   Next to the accelerator identifier this structure contains the addresses of callback functions for context creation, context destroying, executing the program code, handle semaphores, messages and memory operations. The vendor has to fill in the creation, destroying and run callback. Any other entry can be *NULL* if the accelerator do not support the operation. In this case the VFS interface will not display the appropriate entries.

2. **accfs_unregister**

   If the device handler gets unloaded, it will have to call the unregister function, which blocks until the accelerator is not used anymore.

7.3 Device Handler

Above we have seen the tasks of the ”accfs” component, which handles the VFS stuff, but not the virtualization of the accelerator. This has the reason, that the VFS part is too generic to know all portions of the accelerator needed to safe before loading another context.

Thus the device handler has to provide the virtualization by itself. If the accelerator or the device handler do not support virtualization, only the physical amount of accelerator units will be bound to contexts via the *create* callback (confer Figure 6). Any further request has to be quit with the -EBUSY or -EAGAIN error code.

Of cause, the other task of the device handler is to manage the supported accelerators. This may include establish the interconnection or configure memory mappings.

7.4 Further Work

Currently we are implementing and extending the previously mentioned interfaces: The Vendor interface is nearly completed and the VFS supports the first basic operations like creating contexts and writing through the register file.

On the device handler side we have implemented a dummy device handler for the demonstration of the ACCFS interfaces and as reference implementation for other handlers.

The next steps are:

- finish the interface implementation of ACCFS
- porting RSPUFS to an ACCFS device handler for SPEs
- implementing device handlers for the first accelerators other than Cell

8 Conclusion

We propose a hardware independent framework, which supports multiple types of accelerators obsoleting vendor specific interfaces. It is based on the well-know operating system functionality like the VFS infrastructure of the Linux kernel, which enables concurrent access and enforces the compliance with access rights.

We have analyzed several frameworks, concepts and environments, which are available at the moment, for the integration of functional offload engines e.g. GPGPUs, FPGAs and the Cell/B.E. SPUs.

SPUFS was chosen as the basis for our concept. We have analyzed its special integration in the Linux operating system in detail to understand the requirements, which led to the approach of the integration via the VFS.

The further step was the extension to modularity. This provides the separation of hardware specific parts (i.e. initialization and register access) and the management component (i.e. administration of different accelerators from a single point).

Afterwards, the concepts of the (R)SPUFS were extrapolated to a more generic and common interface, which is able to handle the different types of accelerators regardless of their unique architecture or functionality.

Finally, our ACCFS framework provides a superset of the current established interfaces. On the one hand ACCFS ease the development for hardware vendors and application library programmers, and on the other it does not restrict special properties of a single type of accelerator. Vendor device driver must only provide a base for the virtual file system by implementing predefined interfaces and need not worry about the manageability of their hardware. They only need to make available the lowest level of abstraction upwards to the operating system. Application library programmer are now able to use a generalized hardware interface, which enables the usage of different kinds of accelerators. They need not pay attention to restrictions or peculiarities of a particular accelerator.

References

[1] Advanced Micro Devices, Inc. AMD Torrenza Initiative. http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx.

[2] Eric Stahlberg, Daryl Popig, Debbi Ryle, Thomas Steinke, Michael Babst, Mohamed Taher, Kelly Anderson. Molecular Simulations with Hardware Accelerators: A Portable Interface Definition for FPGA Supported Acceleration. In *Third Annual Reconfigurable Systems Summer Institute (RSSI'07)*, 1205 W. Clark St., Urbana, Illinois, July 2007. National Center for Supercomputing Applications.

[3] Andreas Heinig. Execution of SPE code in an Opteron-Cell/B.E. hybrid system. Diploma thesis, Chemnitz University of Technology, 2008.

[4] Ian McCallum, Intel Corporation. Intel® QuickAssist Technology Accelerator Abstraction Layer (AAL), 2007. http://download.intel.com/technology/platforms/quickassist/quickassist_aal_whitepaper.pdf.

[5] Intel Corporation. Intel® Geneseo Technology (PCI Express Technology Advancement), 2007. http://www.intel.com/technology/pciexpress/devnet/docs/Intel_Geneseo_White_Paper_Final.pdf.

[6] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *SPE Runtime Management Library v2.2*, Oct 2007. Cell Broadband Engine Architecture Joint Software Reference Environment Series.

[7] Mark Hummel, Mike Krause, Douglas O'Flaherty. Protocol Enhancements for Tightly Coupled Accelerators, 2007. http://enterprise.amd.com/Downloads/Technology/AMD-HP-tightly-coupled-acc.pdf.

[8] Mitrionics. Mitrion Product Brief, 2007.

[9] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Technical report, NVIDIA Corporation, Santa Clara, CA, Nov. 2007. Version 1.1.

[10] PCI-SIG. I/O Virtualization Specification, 2007. http://www.pcisig.com/specifications/iov/.

[11] Robert W. Smith. New WLAN stack for Linux 2.6.22. *Heise Zeitschriften Verlag online*, May 2007. http://www.heise.de/english/newsticker/news/print/89365.

[12] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.