A CellFS Implementation for the x86 Architecture

Emiel Mols
a.h.mols@student.utwente.nl

ABSTRACT

Programming paradigms that abstract underlying multi-core properties of computer architectures have existed for several decennia now; however, for the recently introduction Cell Broadband Architecture, well-known for its application in the Playstation 3, such paradigms had to be adapted to fully utilize all power of the parallel-focused Cell processor; CellFS is one of the first libraries that tries to implement such a paradigm explicitly for Cell Architecture [INM07].

Because many Cell development is actually done on x86 hardware, we propose a port of the Cell intrinsics of CellFS to x86 hardware. Subsequently, we evaluate the performance of our port.

Keywords: CellFS, Cell Broadband Engine Architecture

1. INTRODUCTION

Due to the ever-growing demand for more processor power and the fact that physical limits prevent limitless increase of processor speeds, computer scientists resort to the application of parallelism in future computing [Sut05].

STI's recently introduced Cell Broadband Engine (Cell BE or Cell for short) Architecture heavily embraces the concept of parallelism: a Cell processor combines a modest-performance RISC processor with a minimum of 6 coprocessing elements designed for efficient multimedia and vector processing [EOO+05]. Due to these coprocessing capabilities, applications such as folding@home (or other software that mainly uses floating point operations) or high resolution multimedia decoding can be executed much faster on the Cell architecture than on price-comparable alternatives. [KBL+08]

From a programmer's point of view, the application of parallelism in general imposes two different challenges:

- software should be designed in a way tasks (threads) can be distributed among the available processor elements in a significant efficient manner, and
- software should take care of synchronizing data between different threads.

A variety of approaches have been developed that allow a programmer to define these different tasks in an abstract way while at the same providing mechanisms for guarded data exchange where possible. On the x86 architecture, this has resulted in frameworks such as pthreads, Cocoa Threading and Java Threading. The CellFS library implements some of these features on the Cell architecture [INM07].

The remainder of this section will give more background information on the Cell BE Architecture and the CellFS project. In section 2, we will outline our research, while section 3 and 4 contain and explain our findings.

10th Twente Student Conference on IT, Enschede 23rd January, 2009

1.1 Cell Broadband Engine Architecture

The Cell processor is based around a generic, Power Architecture-compliant RISC processor (called Power Processor Element or PPE in Cell terminology) that features 64KB L1 cache, 512 L2 cache and supports simultaneous multithreading. The PPE on itself has poor performance when compared to a full fledged modern x86 processor. Cell's core feature, however, is the availability of multiple Synergistic Processing Elements (SPEs) on a single Cell processor. These coprocessors have a relatively simple SIMD instruction set and are highly optimized for vector calculations and multimedia processing. The PPE and SPEs are connected via a bus called the Element Interconnect Bus (EIB) that is also connected to the main memory. Each SPE has 256KB memory (called local store) and DMA logic for bus I/O.

A graphical overview of the Cell Architecture is shown in figure 1.

1.1.1 SDK

IBM has published a software development kit, called libspe2, for the Cell BE Architecture. Besides compiler tools that allow code generation for the SPE ABI, library functions implement 3 relevant mechanisms for communication over the EIB between the PPE and the SPEs:

- DMA transfers for chunks of data
- Mailboxes for small pieces of data
- SPE signals

Communication using mailboxes is relatively cheap, but data is limited to 32 bits per message. Mailboxes can operate in both polling and interrupting mode on the PPE. On SPEs, only polling mode is supported. To interrupt SPE code, the PPE can write to signal registers on a SPE.

In practice, mailboxes and signals are used to synchronize programs running on the different elements, while the DMA feature is used to actually transfer data from or into the SPE local store.

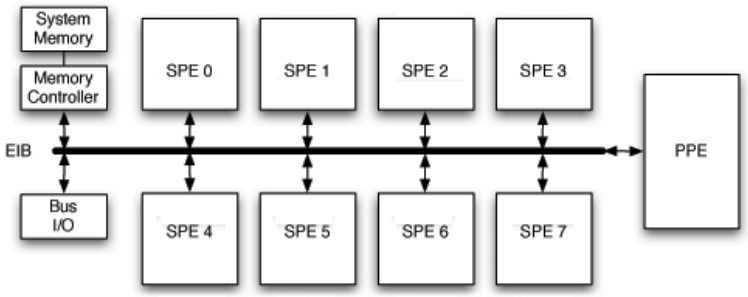

Figure 1: Cell BE Architecture

1.1.2 Operating System Support

Almost all operating systems used in non-Playstation-3 Cell hardware are based on the Linux 2.6 kernel, that natively supports the Cell BA platform. The kernel runs on the PPE, and, through the libspe2 library, tasks may be offloaded to the SPEs.

1.2 CellFS

CellFS' main goal is to provide a generic way of exchanging data from and to the SPEs on a Cell processor. The native communication interface is abstracted in a wrapper that exposes POSIX-like file descriptors. Access to I/O devices that are not directly accessible on the EIB, such as a network interface or a hard drive, is provided by implementing a proxy mechanism on the PPE. This allows for a generic programming approach for programming routines that are to be run on the SPEs, since data does not have to be preloaded or manually supplied by a program that runs on the PPE.

An example program that uses the I/O features of CellFS is shown in figure 2. Relevant functions implemented in the libspu library are spc_open, spc_close, spc_read, spc_write, spc_seek, etc., which operate similar to their posix equivalents.

1.2.1 Plan 9 Filesystem Protocol

CellFS encapsulates the Plan 9 Filesystem Protocol (9P for short) in the CBEA mailbox channels to communicate over the EIB. 9P is a lightweight transaction-based remote filesystem protocol in which the client issues simple requests to be answered by the server within a transaction.

The proxy logic on the PPE implements a 9P server, while the SPEs are all loaded with a simple 9P client library. CellFS' current root filesystem implementation supports 6 different subfilesystems (cited from [INM07]):

- #r File server allowing operation on files existing in ramdisk on the main memory of the Cell.
- #U File server allowing operation on files existing on the UNIX file system accessible by the PPE. Files served by #U are mmap()-ed to main memory to increase I/O bandwidth.
- #R Similar to #U, but changes to the files are not propagated to the disk. This is equivalent to a read-only file system, however it allows the SPEs to communicate data between each other as the computation progresses.

```
1  #include <stdio.h>
2  #include "libspu.h"
3
4  u8 buf1[8192] __attribute__((aligned(128)));
5  u8 buf2[8192] __attribute__((aligned(128)));
6  char stack1[Stksize] __attribute__((aligned
       (128)));
7  char stack2[Stksize] __attribute__((aligned
       (128)));
8
9  void readcor(void *a)
10 {
11   int fdin = spc_open("#p/pip", Oread);
12
13   int n = spc_read(fdin, buf2, sizeof(buf2));
14   buf2[n] = '\0';
15   spc_log("%d: Got: %s", corid(), buf2);
16 }
17
18 void writecor(void *a)
19 {
20   int fdout = spc_create("#p/pip", 0666,
         Owrite);
21   sprintf(buf1, "hello world\n");
22
23   for (int i = 0; i < 20; i++) {
24     spc_write(fdout, buf1, strlen(buf1));
25   }
26 }
27
28 void cormain(unsigned long long spuid,
       unsigned long long argv, unsigned long
       long env)
29 {
30   mkcor(readcor, NULL, &stack1, sizeof(stack1
         ));
31   mkcor(writecor, NULL, &stack2, sizeof(
         stack2));
32 }
```

Figure 2: Example SPU code using CellFS - this example creates two coroutines that communicate over a named pipe.

#p Clients can use this file system to create a named pipe which can be used to communicate between clients running on different SPEs.

#l Log file system used by lightweight library routines replacing printf().

CellFS implements 9P encapsulation in the following manner:

1. When the SPE needs to send a command, it sends a pointer to the command structure in the local SPE store to the interrupting mailbox on the PPE.
2. The PPE reads the memory structure from the SPE local store through DMA, and deserializes the command.
3. When a response is ready, it is written back, through DMA, into the original structure in the SPE local store.
4. When writing is complete, the SPE is notified by a signal from the PPE.

Furthermore, CellFS implements an optimization in the #U- and #R-filesystems: when opening a file, it is mapped into main memory, and instead of reading/writing data through 9P commands, a pointer structure to the location in main memory is provided to the SPE, which consequently accesses the file using direct DMA operations in main memory.

1.2.2 Coroutines

To prevent SPE idle time when waiting on data, CellFS implements a scheduler in the SPE library that can switch context between different so-called coroutines; as a result, multiple coroutines are mapped on a single SPU.

Coroutines are subtly different from threads in that they can yield execution only at given points in the program, thereby preventing many (hard to debug) locking issues that may arise when context switching occurs arbitrarily. This form of cooperative multitasking is implemented in CellFS, and coroutine context switching can occur only when operating on the CellFS file handles.

Similar to generic threading libraries, coroutines in CellFS are created dynamically by supplying an entrypoint function. Every coroutine defines its own in-memory stack, thereby allowing different stack sizes per coroutine. Example code that uses the CellFS library to create two coroutines with an 8K stack each is shown in figure 2.

The following functions are provided by the coroutine scheduler:

- **mkcor(entrypoint_func, argument, stackbase, stacksize)**
  Prepares a new coroutine. Note that the coroutine is not immediately started; instead, it may be scheduled the next time the declaring coroutine is switched out of.
- **yield()**
  Yields current coroutine operation while staying in a ready state.
- **terminate()**
  Terminates the current coroutine.
- **sched()**
  Yields coroutine operation and switches to other coroutine. Used by terminate(), yield() and apc_* functions when waiting for a DMA request to be finished.

These functions are implemented using native SPU assembly helper functions. An SPU element contains 128 general-purpose registers, of which R80 through R127 are non-volatile and therefore saved and restored between context switches. R0 is the Link Register that contains the return address of a function execution, while R1 contains a stack pointer structure. The following helper functions are used:

- **fakelabel**, only used when constructing a coroutine, contains a mechanism to initially point the R0 register to a coroutine's entrypoint and R1 to the coroutine's custom stack
- when switching out of a coroutine, **setlabel** has functionality to save R0, R1 and the non-volatile registers into a separate memory structure
- **getlabel** restores data from this memory structure into the general-purpose registers when switching into a coroutine.

Note that with this mechanism, context switching is implemented on a function call level, and not on a processor instruction level, as one might expect.

As seen in the example code, the user application entrypoint is defined using the cormain() function. The actual main() function, implemented in the CellFS library, registers a default coroutine for cormain(). As a result, code inside cormain() can also make use of coroutine scheduling logic.

1.2.3 Bootloading

CellFS is compatible with any Linux distribution that runs on Cell's PowerPC-compatible PPE. It includes a bootloader program to load binaries into the respective SPUs using functions of IBM's default Cell library. The binaries for the SPUs are separately compiled for the SPU instruction set and link against both IBM's libspe2.h and CellFS' libspu.h: the first contains standard-library implementations for the SPU ABI as well as functionality to access the EIB, while the latter contains the 9P client and coroutine functionality. Loading the SPE binaries is done by invoking the bootloader with the SPE binaries as arguments. After bootloading is complete, the bootloader launches the 9P server to run within the default process space on the PPE.

1.2.4 CellFS Layout

CellFS is logically structured into three separate tiers:

- **libspfs** is a generic 9P server library extended with a cbeconn module to communicate over the CBE; for the latter, it depends on functionality provided by libspe2.
- the **cellfs** tier compiles into the main cellfs binary, containing bootcode logic and the different 9P filesystem implementations; it depends on both libspfs to run the 9P server and on libspe2 to load code into the SPEs.
- **libspu** provides functionality for the SPE binaries, and contains the coroutine scheduler and 9P client logic; it depends on libspe2.h.

2. RESEARCH PROPOSAL

Although the Cell Architecture has promising features, the availability of Cell hardware is relatively scarce, and many Cell development is done on common x86 hardware, for instance using IBM's emulator application. Although this will generally suffice, CellFS' abstraction layer allows us to research a 'more native' solution for applications that use CellFS: an x86 implementation of the Cell intrinsics of CellFS.

In our research, we investigate different methods for transparently translating CellFS' behaviour to x86. We research several models per feature and determine the most suitable by means of literature research and performance testing. We conclude by investigating the limitations of using our implementation for testing Cell applications.

2.1 Relevance

As Sutter [Sut05] stated, the performance of single core computing is bound to come to a halt when certain physical thresholds in processor development are exceeded. Due to the need for alternatives, parallelism is very actively researched in the current field of computer science and all research in this area can be considered of general importance to future computing.

Furthermore, current solutions to test Cell software on x86 systems (depending on the application) run up to 1/20th of its real-time speed, due to the slow nature of SPU instruction emulation. Depending of the performance of our proposed port, using CellFS for Cell application development may significantly ease development and behavior testing.

2.2 Scope

In our research, we focus on 'getting the architecture right', and limit our implementation to a functional one, which may not be fully optimized for speed or resource use. Furthermore, our implementation will only target the Linux platform, which, in our experience, will generally allow for porting to other UNIX compliant operating systems with not too much of an effort. Also, our implementation will obviously not be able to run CellFS applications that still make use of some native libspe2 library calls.

3. ARCHITECTURAL DIFFERENCES

3.1 Cell Hardware Characteristics

When researching possibilities to encompass the hardware properties of the Cell Architecture (figure 1) into a software paradigm on the Linux operating system, we observe the following:

- Normal PowerPC binaries (such as the 9P server program) that run as process in the operating system on the PPE, can, very similarly, be executed in our x86 environment. We only require that the binary is recompiled for the x86 instruction set, which is relatively trivial. Also, most dependency libraries will be identical in both x86 Linux and Linux compiled for the Cell Architecture.
- The easiest and most intuitive way to run SPE code in the normal Linux process space is to recompile the code for the native x86 architecture and spawn a process for every SPE. Alternatively, we could use threads instead of processes for this separation, yet this resembles the original Cell processor layout far less, since SPEs are separate entities that share no resources.

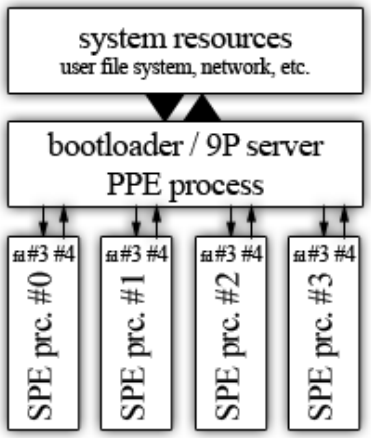

Figure 3: Communication using unix pipes

- In CellFS, data to an SPE always flows through the PPE, even when data comes from main memory (with the exception of memory-mapped files, but this optimization can be removed). To communicate between the 9P server process and the SPE processes, a two-way communication channel between every SPE process and the server process therefor suffices.
- Since there is no problem accessing resources from a process running in standard Linux user space, in theory, we could drop the 9P server process and make a complete implementation for the filesystems in the adapted libspu library. This, however, will effectively prevent the reuse of the filesystem implementation code, and we also expect it to drastically change the coroutine switching behavior compared to the Cell Architecture implementation. Since we are trying to have our port resemble native CellFS where possible, we prefer to keep the 9P client/server architecture intact.

3.2 Porting the EIB

With this in mind, we propose a communication mechanism using UNIX pipes. The pipe-construction is done in the ported bootloader binary, that also launches the SPE binaries as child processes, while, after bootloading is complete, the 9P server is launched in the parent process. This also nicely follows the paradigm of the PPE server process regulating the SPE processes. Furthermore, the 9P server logic is adapted to manage and communicate over the pipes instead of over the EIB. Input and output to an SPE binary will use file descriptor 3 and 4, respectively. To keep our implementation as simple as possible, we will remove the optimization that uses mmap-ed I/O when accessing files from disc, although, in theory, this could be used similarly on the x86 architecture.

A visual overview of the communication mechanism is shown in figure 3.

Alternatively, we considered the possibility of using a network protocol such as TCP. This has the added advantage, that, theoretically, one could spawn the SPE binaries on different networked systems; the downside, however, is a larger overhead and a more complex implementation.

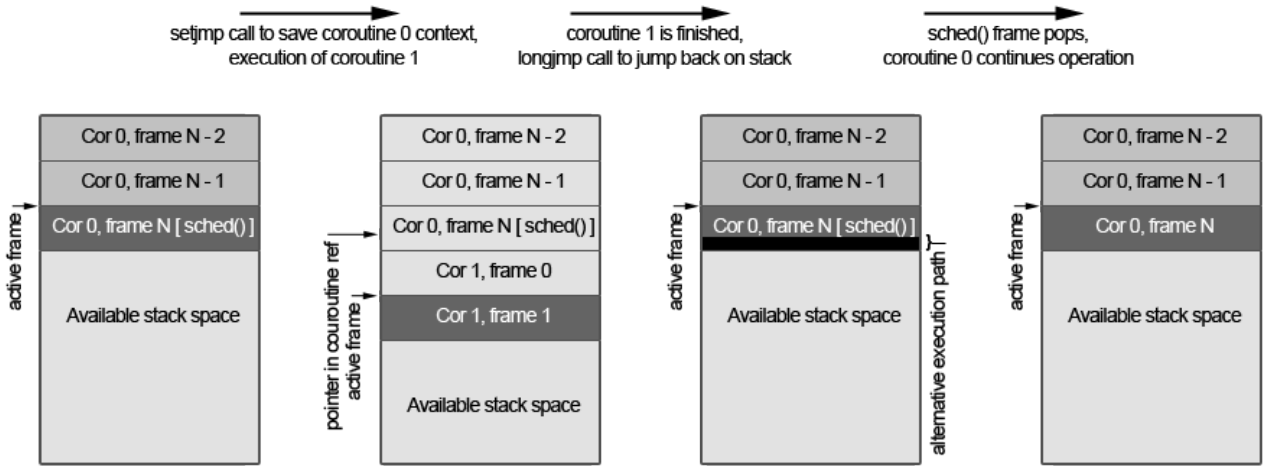

Figure 4: setjmp operates on a single stack.

3.3 Coroutine mapping

Having decided on how to port the Cell hardware characteristics, we are left with porting the coroutine switching that is implemented in SPU assembly in libspu. We evaluated several approaches to map coroutines to a x86 environment:

1. Spawn coroutines into individual child processes. This would allow coroutines to run simultaneously in a very efficient manner. Problems arise, however, since coroutines actually depend on not running concurrently. Since child process scheduling cannot be controlled manually, and locking all shared resources is not feasible either, there seems no way to prevent race conditions.

2. Implement context switching in x86-assembly. Theoretically, the aforementioned functions fakelabel, gotolabel and setlabel could be by adapting to their x86 equivalents. The adaptation, on the other hand, will not be trivial, since assembly execution will not be deterministic due to the native context-switching mechanisms present in Linux.

3. Use the context switching functions setjmp and longjmp. Although these functions initially may seem to be able to do effective context switching, they lack functionality to swap out complete stacks. setjmp can mark the current stack before a second context starts to run; when the second context has finished executing, longjmp can be used to restore the stack for the first context. This operation is visualized in figure 4. Although usable in some cases, this approach will not allow to switch coroutines pseudo-arbitrarily, as is required for CellFS SPU contexts.

4. Spawn coroutines into individual threads. Execution of one coroutine at the time can be guaranteed by managing a single lock for all the coroutines on an SPE. We suspect than such an implementation may not be trivial due to locking requirements in the shared code and the implicit requirement of a custom stack per thread.

5. Use the context switching functions makecontext, setcontext and getcontext, available through the ucontext.h kernel routines. These functions are more advanced than setjmp, allowing the creation of contexts with their own stack space. The major downside is that these functions are quite exotic, and, although specified in the Single UNIX Specification, not all UNIX-like operating systems may provide them.

Although option 4 is a feasible one, we chose approach 5, as it will allow a relatively easy, yet complete implementation. In our implementation, calls to fakelabel, getlabel and setlabel can be replaced with makecontext, setcontext and getcontext, respectively.

Our proposed communication mechanism does not allow signaling the SPE processes when data is ready. This meansthat our coroutine scheduling implementation will have no meansof knowing when a coroutine should switch back from waitingto ready state. Although a signaling feature would probably re-sult in a better resemblance of program execution on Cell hard-ware and our Linux environment, we expect little performancegain since reading data through file pointers costs processor time,as opposed to DMA transfers on the Cell architecture. To makethe libspu implementation functional without signaling, we au-tomatically mark a coroutine as ready when no other coroutinesare available. In the worst case, this will result in the coroutineblocking on reading from the file descriptor.

3.4 Integration in CellFS Layout

We integrate our features into the CellFS repository by adding a new tier 'libspu-x86' that will contain our x86 implementation of libspu; furthermore, we update the cellfs tier to be able to build a cellfs-x86 binary, which is to be used on x86 systems.

A key requirement of our port is that code can be run on x86 systems without requiring source alterations. We therefore adjust the Makefile framework in the repository to allow building for both the x86 architecture and Cell BEA from the same source.

4. TESTING

To test our implementation, we have built a simple set of test scripts. Unfortunately, we do not have any Cell hardware available, which could have been used to compare performance. We

```
#include <stdio.h>
#include "libspu.h"

char stack[10][16384] __attribute__((aligned
    (128)));

int fd;

void
corworker(void *a)
{
  char buf[128];
  int l = sprintf(buf, "corworker::%d\n",
      corid());
  int fd = spc_open("#u/tmp/sout", Owrite);
  spc_write(fd, buf, l);
  spc_close(fd);
}

void
cormain(unsigned long long spuid, unsigned
    long long argv, unsigned long long env)
{
  fd = spc_create("#u/tmp/sout", 0666, Owrite
      );
  spc_close(fd);
  int i;
  for (i = 0; i < 5; i++) {
    mkcor(corworker, i, stack[i], Stksize);
  }
  return;
}
```

Figure 5: Testing of I/O and coroutine scheduling.

```
corworker::1
corworker::2
corworker::3
corworker::4
corworker::5
```

Figure 6: Result of test script.

expect the performance to be very dependent on type of code that is to be run on the SPEs: in the case of operations that can effectively use the SPE-specific instruction set, running this code on x86 hardware will have incomparable performance.

One of our test scripts is shown in figure 5 and the result in figure 6.

5. CONCLUSION

We have presented a port to the x86 architecture of CellFS that allows the compilation and execution of programs that depend (solely) on the CellFS communication mechanism to offload execution logic to SPE-elements.

For our port, the hardware architecture within a Cell architecture is mapped to software paradigms available in the Linux operating system where possible; SPE binaries are launched as separate child processes with the 9P server process as parent, using unix pipes for communication. For the coroutine scheduling logic, we have resorted the the ucontext system library, allowing similar behavior on the Linux platform as the native SPE-assembly implementation. Following the original Cell hardware architecture where possible has made the porting process relatively easy, allowing many code to be reused.

5.1 Future work

Our research lacks concrete performance testing. By researching the categorization of different application types, we believe valuable data concerning our implementation might be attained. Also, our tests were limited to relatively simple (single processor) hardware; we suspect testing and tuning an actual CellFS application for modern SMT systems might result in more usable results. We also think different improvements can be made on our implementation. We suspect the best performance is to be gained by making a complete reimplementation of libspu that does not depend on a 9P server but accesses local resources directly, although the disadvantages named in section 3.1 apply and the coroutine scheduling logic may have to be adjusted significantly. Another improvement would be to port the mmap()-based filesystem.

REFERENCES

[INM07] L. Ionkov, A. Nyrhinen, A. Mirtchovski. CellFS: Taking The "DMA" Out Of Cell Programming, April 2007.

[EOO+05] A. Eichenberger, K. O'Brien, K. O'Brien e.a. Optimizing Compiler for a CELL Processor.

[Ber05] IBM. Spufs: The cell synergistic processing unit as a virtual file system. `http://www-128.ibm.com/developerworks/power/library/pa-cell/`.

[Wel94] B. Welch. A Comparison of three Distributed File System Architectures: Vnode, Sprite, and Plan 9.

[NBF96] B. Nichols, D. Buttlar, J. Proulx Farrell. Pthread Programming: A POSIX Standard for Better Multiprocessing, O'Reilly, 1996.

[Joh06] R. Johnson. POSIX Threads (pthreads) for Win32. `http://sourceware.org/pthreads-win32/`.

[Sut05] H. Sutter The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.

[KBL+08] J. Kurzak, A. Buttari, P. Luszczek, J. Dongarra The PlayStation 3 for High-Performance Scientific Computing.