Linux on Cell Broadband Engine status update

Arnd Bergmann
IBM Linux Technology Center
arnd.bergmann@de.ibm.com

Abstract

With Linux for the Sony PS3, the IBM QS2x blades and the Toshiba Celleb platform having hit mainstream Linux distributions, programming for the Cell BE is becoming increasingly interesting for developers of performance computing. This talk is about the concepts of the architecture and how to develop applications for it.

Most importantly, there will be an overview of new feature additions and latest developments, including:

- Preemptive scheduling on SPUs (finally!): While it has been possible to run concurrent SPU programs for some time, there was only a very limited version of the scheduler implemented. Now we have a full time-slicing scheduler with normal and real-time priorities, SPU affinity and gang scheduling.
- Using SPUs for offloading kernel tasks: There are a few compute intensive tasks like RAID-6 or IPsec processing that can benefit from running partially on an SPU. Interesting aspects of the implementation are how to balance kernel SPU threads against user processing, how to efficiently communicate with the SPU from the kernel and measurements to see if it is actually worthwhile.
- Overlay programming: One significant limitation of the SPU is the size of the local memory that is used for both its code and data. Recent compilers support overlays of code segments, a technique widely known in the previous century but mostly forgotten in Linux programming nowadays.

1 Background

The architecture of the Cell Broadband Engine (Cell/B.E.) is unique in many ways. It combines a general purpose PowerPC processor with eight highly optimized vector processing cores called the Synergistic Processing Elements (SPEs) on a single chip. Despite implementing two distinct instruction sets, they share the design of their memory management units and can access virtual memory in a cache-coherent way.

The Linux operating system runs on the PowerPC Processing Element (PPE) only, not on the SPEs, but the kernel and associated libraries allow users to run special-purpose applications on the SPE as well, which can interact with other applications running on the PPE. This approach makes it possible to take advantage of the wide range of applications available for Linux, while at the same time utilize the performance gain provided by the SPE design, which could not be achieved by just recompiling regular applications for a new architecture.

One key aspect of the SPE design is the way that memory access works. Instead of a cache memory that speeds up memory accesses in most current designs, data is always transferred explicitly between the local on-chip SRAM and the virtually addressed system memory. An SPE program resides in the local 256KiB of memory, together with the data it is working on. Every time it wants to work on some other data, the SPE tells its Memory Flow Controller (MFC) to asynchronously copy between the local memory and the virtual address space.

The advantage of this approach is that a well-written application practically never needs to wait for a memory access but can do all of these in the background. The disadvantages include the limitation to 256KiB of directly addressable memory that limit the set of applications that can be ported to the architecture, and the relatively long time required for a context switch, which needs to save and restore all of the local memory and the state of ongoing memory transfers instead of just the CPU registers.

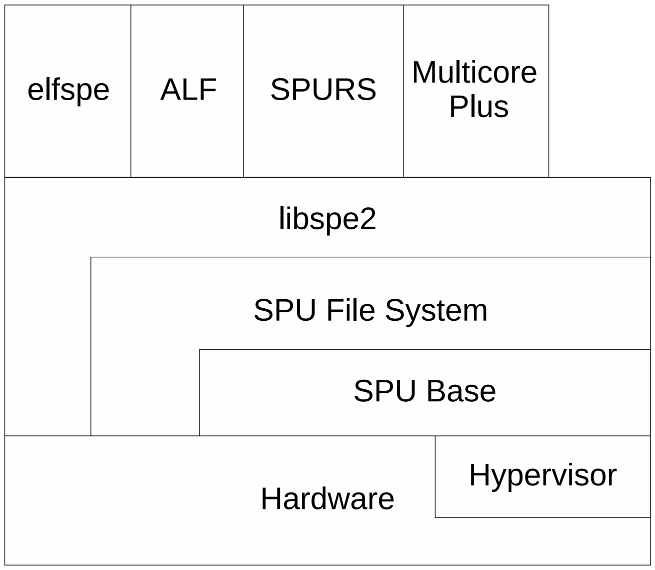

Figure 1: Stack of APIs for accessing SPEs

1.1 Linux port

Linux on PowerPC has a concept of platform types that the kernel gets compiled for, there are for example separate platforms for IBM System p and the Apple Power Macintosh series. Each platform has its own hardware specific code, but it is possible to enable combinations of platforms simultaneously. For the Cell/B.E., we initially added a platform named "cell" to the kernel, which has the drivers for running on the bare metal, i.e. without a hypervisor. Later, the code for both the Toshiba Celleb platform and Sony's PlayStation 3 platform were added, because each of them have their own hypervisor abstractions that are incompatible with each other and with the hypervisor implementations from IBM. Most of the code that operates on SPEs however is shared and provides a common interface to user processes.

2 Programming interfaces

There is a variety of APIs available for using SPEs, I'll try to give an overview of what we have and what they are used for. For historic reasons, the kernel and toolchain refer to SPUs (Synergistic Processing Units) instead of SPEs, of which they are strictly speaking a subset. For practical purposes, these two terms can be considered equivalent.

2.1 Kernel SPU base

There is a common interface for simple users of an SPE in the kernel, the main purpose is to make it possible to implement the SPU file system (spufs). The SPU base takes care of probing for available SPEs in the system and mapping their registers into the kernel address space. The interface is provided by the `include/asm-powerpc/spu.h` file. Some of the registers are only accessible through hypervisor calls on platforms where Linux runs virtualized, so accesses to these registers get abstracted by indirect function calls in the base.

A module that wants to use the SPU base needs to request a handle to a physical SPU and provide interrupt handler callbacks that will be called in case of events like page faults, stop events or error conditions.

The SPU file system is currently the only user of the SPU base in the kernel, but some people have implemented experimental other users, e.g. for acceleration of device drivers with SPUs inside of the kernel. Doing this is an easy way for prototyping kernel code, but we are recommending the use of spufs even from inside the kernel for code that you intend to have merged upstream. Note that as in-kernel interfaces, the API of the SPU base is not stable and can change at any time. All of its symbols are exported only to GPL-licensed users.

2.2 The SPU file system

The SPU file system provides the user interface for accessing SPUs from the kernel. Similar to procfs and sysfs, it is a purely virtual file system and has no block device as its backing. By convention, it gets mounted world-writable to the `/spu` directory in the root file system.

Directories in spufs represent SPU contexts, whose properties are shown as regular files in them. Any interaction with these contexts is done through file operation like read, write or mmap. At time of this writing, there are 30 files that are present in the directory of an SPU context, I will describe some of them as an example later.

Two system calls have been introduced for use exclusively together with spufs, spu_create and spu_run. The spu_create system call creates an SPU context in the kernel and returns an open file descriptor for the directory associated with it. The open file descriptor is significant, because it is used as a measure to determine the life time of the context, which is destroyed when the file descriptor is closed.

Note the explicit difference between an SPU context and a physical SPU. An SPU context has all the properties of an actual SPU, but it may not be associated with one and only exists in kernel memory. Similar to task switching, SPU contexts get loaded into SPUs and removed from them again by the kernel, and the number of SPU contexts can be larger than the number of available SPUs.

The second system call, spu_run, acts as a switch for a Linux thread to transfer the flow of control from the PPE to the SPE. As seen by the PPE, a thread calling spu_run blocks in that system call for an indefinite amount of time, during which the SPU context is loaded into an SPU and executed there. An equivalent to spu_run on the SPU itself is the stop-and-signal instruction, which transfers control back to the PPE. Since an SPE does not run signal handlers itself, any action on the SPE that triggers a signal or others sending a signal to the thread also cause it to stop on the SPE and resume running on the PPE.

Files in a context include

mem The mem file represents the local memory of an SPU context. It can be accessed as a linear file using read/write/seek or mmap operation. It is fully transparent to the user whether the context is loaded into an SPU or saved to kernel memory, and the memory map gets redirected to the right location on a context switch. The most important use of this file is for an object file to get loaded into an SPU before it is run, but mem is also used frequently by applications themselves.

regs The general purpose registers of an SPU can not normally be accessed directly, but they can be in a saved context in kernel memory. This file contains a binary representation of the registers as an array of 128-bit vector variables. While it is possible to use read/write operations on the regs file in order to set up a newly loaded program or for debugging purposes, every access to it means that the context gets saved into a kernel save area, which is an expensive operation.

wbox The wbox file represents one of three mail box files that can be used for unidirectional communication between a PPE thread and a thread running on the SPE. Similar to a FIFO, you can not seek in this file, but only write data to it, which can be read using a special blocking instruction on the SPE.

phys-id The phys-id does not represent a feature of a physical SPU but rather presents an interface to get auxiliary information from the kernel, in this case the number of the SPU that a context is loaded into, or -1 if it happens not to be loaded at all at the point it is read. We will probably add more files with statistical information similar to this one, to give users better analytical functions, e.g. with an implementation of top that knows about SPU utilization.

2.3 System call vs. direct register access

Many functions of spufs can be accessed through two different ways. As described above, there are files representing the registers of a physical SPU for each context in spufs. Some of these files also allow the mmap() operation that puts a register area into the address space of a process.

Accessing the registers from user space through mmap can significantly reduce the system call overhead for frequent accesses, but it carries a number of disadvantages that users need to worry about:

- When a thread attempts to read or write a register of an SPU context running in another thread, a page fault may need to be handled by the kernel. If that context has been moved to the context save area, e.g. as the result of preemptive scheduling, the faulting thread will not make any progress until the SPU context becomes running again. In this case, direct access is significantly slower than indirect access through file operations that are able to modify the saved state.

- When a thread tries to access its own registers while it gets unloaded, it may block indefinitely and need to be killed from the outside.

- Not all of the files that can get mapped on one kernel version can be on another one. When using 64k pages, some files can not be mapped due to hardware restrictions, and some hypervisor implementations put different limitation on what can be mapped. This makes it very hard to write portable applications using direct mapping.

- In concurrent access to the registers, e.g. two threads writing simultaneously to the mailbox, the user application needs to provide its own locking mechanisms, as the kernel can not guarantee atomic accesses.

In general, application writers should use a library like libspe2 to do the abstraction. This library contains functions to access the registers with correct locking and provides a flag that can be set to attempt using the direct mapping or fall back to using the safe file system access.

2.4 elfspe

For users that want to worry as little as possible about the low-level interfaces of spufs, the elfspe helper is the easiest solution. Elfspe is a program that takes an SPU ELF executable and loads it into a newly created SPU context in spufs. It is able to handle standard callbacks from a C library on the SPU, which are needed e.g. to implement printf on the SPU by running some of code on the PPE.

By installing elfspe with the miscellaneous binary format kernel support, the kernel execve() implementation will know about SPU executables and use `/sbin/elfspe` as the interpreter for them, just like it calls interpreters for scripts that start with the well-known "#!" sequence.

Many programs that use only the subset of library functions provided by newlib, which is a C runtime library for embedded systems, and fit into the limited local memory of an SPE are instantly portable using elfspe. Important functionalities that does not work with this approach include:

shared libraries Any library that the executable needs also has to be compiled for the SPE and its size adds up to what needs to fit into the local memory. All libraries are statically linked.

threads An application using elfspe is inherently single-threaded. It can neither use multiple SPEs nor multiple threads on one SPE.

IPC Inter-process communication is significantly limited by what is provided through newlib. Use of system calls directly from an SPE is not easily available with the current version of elfspe, and any interface that requires shared memory requires special adaptation to the SPU environment in order to do explicit DMA.

2.5 libspe2

Libspe2 is an implementation of the operating-system-independent "SPE Runtime Management Library" specification.[1] This is what most applications are supposed to be written for in order to get the best degree of portability. There was an earlier libspe 1.x, that is not actively maintained anymore since the release of version 2.1.

Unlike elfspe, libspe2 requires users to maintain SPU contexts in their own code, but it provides an abstraction from the low-level spufs details like file operations, system calls and register access.

Typically, users want to have access to more than one SPE from one application, which is typically done through multithreading the program: each SPU context gets its own thread that calls the spu_run system call through libspe2. Often, there are additional threads that do other work on the PPE, like communicating with the running SPE threads or providing a GUI. In a program where the PPE hands out tasks to the SPEs, libspe2 provides event handles that the user can call blocking functions like `epoll_wait()` on to wait for SPEs requesting new data.

2.6 Middleware

There are multiple projects targeted at providing a layer on top of libspe2 to add application-side scheduling of jobs inside of an SPU context. These include the SPU Runtime System (SPURS) from Sony, the Accelerator Library Framework (ALF) from IBM and the MultiCore Plus SDK from Mercury Computer Systems.

All these projects have in common that there is no public documentation or source code available at this time, but that will probably change in the time until the Linux Symposium.

[1] `http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/1DFEF31B3211112587257242007883F3/$file/cplibspe.pdf`

3 SPU scheduling

While spufs has had the concept of abstracting SPU contexts from physical SPUs from the start, there has not been any proper scheduling for a long time. An initial implementation of a preemptive scheduler was first merged in early 2006, but then disabled again as there were too many problems with it.

After a lot of discussion, a new implementation of the SPU scheduler from Christoph Hellwig has been merged in the 2.6.20 kernel, initially only supporting only SCHED_RR and SCHED_FIFO real-time priority tasks to preempt other tasks, but later work was done to add time slicing as well for regular SCHED_OTHER threads.

Since SPU contexts do not directly correspond to Linux threads, the scheduler is independent of the Linux process scheduler. The most important difference is that a context switch is performed by the kernel, running on the PPE, not by the SPE, which the context is running on.

The biggest complication when adding the scheduler is that a number of interfaces expect a context to be in a specific state. Accessing the general purpose registers from GDB requires the context to be saved, while accessing the signal notification registers through mmap requires the context to be running. The new scheduler implementation is conceptually simpler than the first attempt in that no longer attempts to schedule in a context when it gets accessed by someone else, but rather waits for the context to be run by means of another thread calling spu_run.

Accessing one SPE from another one shows effects of non-uniform memory access (NUMA) and application writers typically want to keep a high locality between threads running on different SPEs and the memory they are accessing. The SPU code therefore has been able for some time to honor node affinity settings done through the NUMA API. When a thread is bound to a given CPU while executing on the PPE, spufs will implicitly bind the thread to an SPE on the same physical socket, to the degree that relationship is described by the firmware.

This behavior has been kept with the new scheduler, but has been extended by another aspect, affinity between SPE cores on the same socket. Unlike the NUMA interfaces, we don't bind to a specific core here, but describe the relationship between SPU contexts. The spu_create system call now gets an optional argument that lets the user pass the file descriptor of an existing context. The spufs scheduler will then attempt to move these contexts to physical SPEs that are close on the chip and can communicate with lower overhead than distant ones.

Another related interface is the temporal affinity between threads. If the two threads that you want to communicate with each other don't run at the same time, the special affinity is pointless. A concept called gang scheduling is applied here, with a gang being a container of SPU contexts that are all loaded simultaneously. A gang is created in spufs by passing a special flag to spu_create, which then returns a descriptor to an empty gang directory. All SPU contexts created inside of that gang are guaranteed to be loaded at the same time.

In order to limit the number of expensive operations of context switching an entire gang, we apply lazy context switching to the contexts in a gang. This means we don't load any contexts into SPUs until all contexts in the gang are waiting in spu_run to become running. Similarly, when one of the threads stops, e.g. because of a page fault, we don't immediately unload the contexts but wait until the end of the time slice. Also, like normal (non-gang) contexts, the gang will not be removed from the SPUs unless there is actually another thread waiting for them to become available, independent of whether or not any of the threads in the gang execute code at the end of the time slice.

4 Using SPEs from the kernel

As mentioned earlier, the SPU base code in the kernel allows any code to get access to SPE resources. However, that interface has the disadvantage to remove the SPE from the scheduling, so valuable processing power remains unused while the kernel is not using the SPE. That should be most of the time, since compute-intensive tasks should not be done in kernel space if possible.

For tasks like IPsec, RAID6 or dmcrypt processing offload, we usually want the SPE to be only blocked while the disk or network is actually being accessed, otherwise it should be available to user space.

Sebastian Siewior is working on code to make it possible to use the spufs scheduler from the kernel, with the concrete goal of providing cryptoapi offload functions for common algorithms.

For this, the in-kernel equivalent of libspe is created, with functions that directly do low-level accesses instead of going through the file system layer. Still, the SPU contexts are visible to user space applications, so they can get statistic information about the kernel space SPUs.

Most likely, there should be one kernel thread per SPU context used by the kernel. It should also be possible to have multiple unrelated functions that are offloaded from the kernel in the same executable, so that when the kernel needs one of them, it calls into the correct location on the SPU. This requires some infrastructure to link the SPU objects correctly into a single binary. Since the kernel does not know about the SPU ELF file format, we also need a new way of initially loading the program into the SPU, e.g. by creating a save context image as part of the kernel build process.

First experiments suggest that an SPE can do an AES encryption about four times faster than a PPE. It will need more work to see if that number can be improved further, and how much of it is lost as communication overhead when the SPE needs to synchronize with the kernel. Another open question is whether it is more efficient for the kernel to synchronously wait for the SPE or if it can do something else at the same time.

5 SPE overlays

One significant limitation of the SPE is the size that is available for object code in the local memory. To overcome that limitation, new binutils support overlay to support overlaying ELF segments into concurrent regions. In the most simple case, you can have two functions that both have their own segment, with the two segments occupying the same region. The size of the region is the maximum of either segment size, since they both need to fit in the same space.

When a function in an overlay is called, the calling function first needs to call a stub that checks if the correct overlay is currently loaded. If not, a DMA transfer is initiated that loads the new overlay segment, overwriting the segment loaded into the overlay region before. This makes it possible to even do function calls in different segments of the same region.

There can be any number of segments per region, and the number of regions is only limited by the size of the local storage. However, the task of choosing the optimal configuration of which functions to go into what segment is up to the application developer. It gets specified through a linker script that contains a list of OVERLAY statements, each of them containing a list of segments that go into an overlay.

It is only possible to overlay code and read-only data, but not data that is written to, because overlay segments only ever get loaded into the SPU, but never written back to main memory.

6 Profiling SPE tasks

Support for profiling SPE tasks with the oprofile tool has been implemented in the latest IBM Software Development Kit for Cell. It is currently in the process of getting merged into the mainline kernel and oprofile user space packages.

It uses the debug facilities provided by the Cell/B.E. hardware to get sample data about what each SPE is doing, and then maps that to currently running SPU contexts. When the oprofile report tool runs, that data can be mapped back to object files and finally to source code lines that a developer can understand. So far, it behaves like oprofile does for any Linux task, but there are a few complications.

The kernel, in this case spufs, has by design no knowledge about what program it is running, the user space program can simply load anything into local storage. In order for oprofile to work, a new "object-id" file was added to spufs, which is used by libspe2 to tell oprofile the location of the executable in the process address space. This file is typically written when an application is first started and does not have any relevance except when profiling.

Oprofile uses the object-id in order to map the local store addresses back to a file on the disk. This can either be a plain SPU executable file, or a PowerPC ELF file that embeds the SPU executable as a blob. This means that every sample from oprofile has three values: The offset in local store, the file it came from, and the offset in that file at which the ELF executable starts.

To make things more complicated, oprofile also needs to deal with overlays, which can have different code at the same location in local storage at different times. In order to get these right, oprofile parses some of the ELF headers of that file in kernel space when it is first loaded, and locates an overlay table in SPE local storage with this to find out which overlay was present for each sample it took.

Another twist is self-modifying code on the SPE, which happens to be used rather frequently, e.g. in order to do system calls. Unfortunately, there is nothing that oprofile can safely do about this.

7 Combined Debugger

One of the problems with earlier version of GDB for SPU was that GDB can only operate on either the PPE or the SPE. This has now been overcome by the work of Ulrich Weigand on a combined PPE/SPE debugger.

A single GDB binary now understands both instruction sets and knows how switch between the two. When GDB looks at the state of a thread, it now checks if it is in the process of executing the spu_run system call. If not, it shows the state of the thread on the PPE side using ptrace, otherwise it looks at the SPE registers through spufs.

This can work because the SIGSTOP signal is handled similarly in both cases. When gdb sends this signal to a task running on the SPE, it returns from the spu_run system call and suspends itself in the kernel. GDB can then do anything to the context and when it sends a SIGCONT, spu_run will be restarted with updated arguments.

8 Legal Statement

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada