Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Porting the GNU Tool Chain to the Cell Architecture

Ulrich Weigand
IBM Deutschland Entwicklung GmbH
uweigand@de.ibm.com

Abstract

The Cell processor architecture, jointly developed by Sony, Toshiba, and IBM, represents a new direction in processor design. The Cell processor features in addition to a PowerPC-compatible Power Processor Element (PPE) an array of eight Synergistic Processor Elements (SPEs) supporting a new instruction set. SPEs are optimized for media workloads, providing a set of 128 vector registers of width 128 bits, and capable of sustaining 4 multiply-and-add operations per cycle. Applications making optimal use of the Cell processor will comprise both PPE and SPE components, requiring tool chain support for working with two different instruction set architectures and ABIs in an integrated fashion.

Work is currently underway to port both the Linux kernel and the GNU tool chain to run on a system based on the Cell processor. We will give an overview of the features of the Cell architecture relevant to the tool chain and introduce the proposed programming model for Cell software. We will present the current status of the GNU tool chain porting effort, and discuss in particular those areas where architecture-independent common code needs to be changed in order to support the Cell processor. Special focus will be placed on the question how to enable GDB to debug a combined PPE/SPE program.

1 Introduction

The Cell processor [1] is a multi-core microprocessor, containing a Power Processor Element (PPE) and an array of eight Synergistic Processor Elements (SPEs). The PPE is a general-purpose 64-bit PowerPC-compatible processor providing the VMX multimedia extension. In the current Cell processor, the PPE uses an in-order pipeline and supports two-way simultaneous multi-threading. It runs the operating system and performs system management and control tasks; it can also run existing PowerPC-compatible application programs.

Each SPE [2] consists of a Synergistic Processor Unit (SPU) and a memory-flow controller (MFC). They are intended to provide computational performance, in particular for game, media, and broadband workloads. The SPE implements a new instruction set architecture featuring a register file of 128 SIMD vector registers, each 128 bits wide. All general-purpose SPE instructions are SIMD vector instructions and operate on that register file; the contents of a register can be interpreted as a vector of either sixteen 8-bit integers, eight 16-bit integers, four 32-bit integers or single-precision floating-point values, or two double-precision floating-point values. Operations on scalar values have to be implemented by simply using vector operations on vectors holding the scalar as one “preferred” element; the scalar result of the operation is again found in the preferred element

slot, while the other result vector elements can be simply ignored.

Load and store instructions allow to access a 256KB local-storage memory (local store) private to the SPE. All memory accessed directly by SPE load and store instructions, as well as the instruction text itself, must reside within that local store. However, the SPE is able to use DMA operations to copy data between local store and main storage, using facilities provided by the MFC. Local store accesses are not subject to address translation; however when the SPE performs a DMA operation, the main storage effective addresses involved in that operation will be translated to physical addresses using the page tables set up by the operating system. Local store accesses are also not subject to protection checks, and addresses beyond the local store limit will simply wrap around; thus SPE load and store instructions can never cause an exception to be generated.

The SPE supports only a single mode of operation, there is no distiction between problem state and privileged state. When running Linux on the Cell processor, all privileged operating system tasks are performed by PPE code.

2 Linux on Cell

Linux and the GNU tool chain have been ported to the Cell architecture early during the design phase of the processor, and were extensively used in simulation and during bringup. With a team at the IBM Böblingen development lab, in cooperation with the STI Design Center in Austin, we are currently working on providing support to run Linux on a hardware system based on the Cell processor. In this section of the paper we will describe the core features and interfaces available to application programs running under Linux on that system; the implications on the GNU tool chain and the GDB debugger will be examined later on.

2.1 Kernel support

At the time of this writing, we are currently working on Linux kernel support for the Cell architecture; initial patches have just been published on the kernel mailing list. This consists of a new subarchitecure of `ppc64`, paralleling the existing pSeries and Power Mac subarchitectures. It provides all necessary support to operate the Cell processor, including interrupt controller and IOMMU functionality. SMP and multi-threaded environments will work as well. For PPE applications running under this kernel, the environment will appear very similar to other `ppc64` machines.

In order to allow user space applications to run code on the Synergistic Processor Elements of the Cell processor, the Linux kernel provides a name space via a file system called `spufs`, similar to existing file systems providing name spaces for posix shared memory or message queues. Given appropriate permissions, users can create directories in the `spufs` root; each directory represents a logical SPE context.

In the current implementation, each logical SPE context is mapped directly to a physical SPE present in the system. In future we plan to allow the kernel to virtualize SPEs by switching between multiple SPE contexts running on the same physical SPE. However, note that this type of kernel-provided context switch will be significantly more expensive than traditional PPE context switches, due to the large amount of context information that needs to be saved and restored. Thus, the facility should be employed judiciously.

Each SPE context directory in `spufs` contains a predefined set of files used for manipulating

the state of the logical SPE. At the time of this writing, the exact set of supported files has not been fully defined yet. However, these files of primary interest here:

The `mem` file represents the contents of the local store memory of the SPE. It can be accessed like a shared memory segment; read, write and mmap access is supported. For SPE contexts currently scheduled onto a physical SPE, the physical local store can be directly accessed through that file; for SPE contexts not scheduled the file can be used to access the saved context information instead.

The `regs` file represents the contents of the SPE register set, i.e. a set of 128 16-byte registers; reading and writing that file will cause the kernel to fetch or modify the register contents of the SPE. This is possible only at a time when the SPE is currently not running. (Note that this file should only be used for debugging purposes.)

The `run` file allows to actually execute code on the SPE. The only supported access to this file is via ioctl. Currently only one ioctl method, `spufs_run`, is supported. The ioctl needs to be called with a pointer to a structure of this type:

```
struct spufs_run_arg {
       u32 npc;
       u32 status;
};
```

On entry to the ioctl, `npc` specifies a Next Program Counter value; the kernel will start execution on the SPE at this address. While SPE code executes, the PPE thread will block on the ioctl system call. Once SPE execution stops due to execution of certain instructions on the SPE, the kernel will update `npc` with the address of the next instruction to be executed on the SPE, and `status` will be updated to hold the status code signalled by the last SPE instruction executed, if applicable.

If a PPE thread currently blocked on a `spufs_run` ioctl receives a signal that needs to cause the system call to return to user space, the kernel will first ensure the SPE is stopped. As above, the `npc` value is updated; this allows the PPE thread to continue executing of interrupted SPE processing by just re-issuing the same ioctl operation. In fact, this will happen by default in the case where the signal was marked to automatically restart interrupted system calls using the `SA_RESTART` flag.

To summarize, in this model all SPE execution takes place only on behalf of a specific PPE thread which is currently blocked in an ioctl system call. This allows SPE threads of execution to be identified by the task ID of that PPE thread; for example it is possible to kill a currently running SPE thread by sending a regular kill signal to the corresponding PPE task ID.

2.2 PPE programming interfaces

The `spufs` kernel interface described in the previous section is not intended to be the primary interface used by application programs containing SPE execution components. Similarly to how the Linux user space thread library provides the standard pthread interface to application programs, implemented on top of a Linux-specific kernel interface, we will provide a `libspu` user space library that exposes an application interface to handle SPE execution threads. On Linux, this library will be implemented using the `spufs` kernel interface. However, the `libspu` interface may be made available on other operating systems for the Cell platform as well; applications written to that interface will thus be more easily portable.

At the time of this writing, the `libspu` interface specification was not yet fully standard-

ized. We will provide an overview of some aspects of the current implementation here; be aware that details may still be subject to change at this point.

The central concept of the `libspu` interface is the SPE thread, represented by an `spuid_t` identifier. SPE threads can be created using the `spu_create_thread` routine. The program to be executed within the new SPE thread is identified by passing a pointer to the in-memory copy of an SPE executable image to the `spu_create_thread` routine. The caller can establish such a pointer either by memory-mapping a stand-alone SPE executable file using the `spu_open_image` routine, or else by referring to an SPE executable image embedded into the PPE executable file (see below). Using an SPE image loader, which is part of the `libspu` library, the `spu_create_thread` routine will load the SPE executable into the local store associated with the new SPE context, and then start SPE execution at the corresponding entry point.

In addition, the `libspu` interface provides a variety of other routines related to SPE threads, like `spu_kill` to send signals to an SPE thread or `spu_wait` to wait until an SPE thread has terminated.

2.3 SPE programming interfaces

The SPE loader initializes local store and register contents as defined by the SPE ABI prior to starting execution on the SPE. The initial stack pointer is set up to point to a minimal stack frame at the top of the stack, where the stack frame backchain is terminated by a NULL backchain pointer. In addition, some parameters passed by the PPE application to the `spu_create_thread` are made available to the SPE start-up code, which then calls the high-level language entry point of the SPE application using those parameters. For a C application, that entry point will be conventionally declared as:

```
extern int
main (unsigned int spu_id,
      unsigned long long param,
      unsigned long long env);
```

where `spu_id` is the unique SPE task identifier, `param` is a system memory address pointing to application parameters, and `env` is a system memory address pointing to runtime environment information.

To finish processing, the SPE program executes a Stop and Signal instruction, which halts SPE execution and presents an interrupt to the operating system, providing a signal code. Certain signal codes are reserved by the ABI to denote normal and abnormal termination; one signal code is reserved to denote a debugger breakpoint. Other signal codes may be freely used to communicate status between the SPE and PPE components of an application.

SPE applications can be written either in assembly language or a high-level language; currently support for C and limited support for C++ exist. However, to accommodate the special circumstances encountered in the SPE environment, some extensions and modifications to the C and C++ language standards apply for SPE code. To allow applications to exploit the SIMD capability of the processor, vector data types and intrinsics are defined, similar to the types and intrinsics allowing to exploit the VMX instruction set on PowerPC. In addition, most SPE assembler instructions are directly represented as intrinsic functions, allowing to write fully optimized code for performance-critical sections without resorting to GCC's inline assembler facility.

Due to the restricted local store size, it is not easily possible to make the full set of library

functions defined by the C and C++ standard available to SPE code. Work is currently underway to define a reasonable subset meeting the requirements of typical SPE code. For less frequently used library calls, the code implementing the functionality can be offloaded to the PPE by executing a "system call" on the SPE. Although not an ABI requirement, system calls are implemented by Stop and Signal instructions with special signal codes which are intercepted and handled by PPE operating system and/or runtime support code.

3 Compiler and tool chain

A team at Sony Computer Entertainment is working on providing GCC and GNU binutils support for the Cell processor. At the time of this writing, work on a tool chain based on GCC 3.4.1 and binutils 2.15 is ongoing; initial patches should be made public soon.

3.1 Object file format

As object file format, the Executable and Linking Format (ELF) is used. SPE binaries are implemented as 32-bit big-endian ELF files with the ELF header `e_machine` field set to the `EM_SPU` processor identification; this identifier has been registered with a numerical value of 23. ELF executable files must be statically linked; the SPE loader does not perform any relocation processing. A limited form of shared libraries called SPE plugins is supported, however. Like regular SPE executables, plugins require no relocations and provide just a single entry point; however, they contain position-independent code, allowing them to be loaded at different local store addresses as required. The current implementation of SPE plugins requires some binutils common code changes.

A new binutils target `spu` supporting this variant of ELF allows a set of cross-tools hosted on the PPE and targetting the SPE object file format to be built; the tools do not run natively on the SPE. To process PPE object files, the existing `ppc` and `ppc64` targets can be used, possibly slightly extended to handle some new features of the Cell processor.

To facilitate applications with both PPE and SPE components, it is possible to embed one or more SPE binaries into a PPE executable. To do so, a section called `.spuelf.`*`filename`* is added to the PPE executable, whose contents are an identical copy of the SPE executable *filename*. The section should be 128-byte aligned to facilitate DMA transfer into SPE local store. A new utility `embedspu` takes an SPE executable and embeds it in this fashion into a PPE object file, and defines a global variable pointing to the start of the new section. This object can subsequently be linked into a PPE executable that uses this global pointer to load the embedded SPE executable onto a SPE. (The `embedspu` utility could in fact be implemented as a variant of the `--add-section` command of `objcopy`.)

3.2 GCC support

For GCC, it is sufficient to handle the PPE and SPE components of an application completely separate, similarly as described for binutils. As the PPE is PowerPC compatible, it is already supported by the existing `rs6000` GCC back end. However, processor-specific tuning information and a scheduler pipeline description for the Cell processor have been added, allowing generation of PowerPC code optimized for the PPE.

Code generation for the SPE is implemented in a new GCC back end for the `spu` target. Like mentioned above for binutils, this is typically

used to build a cross-compiler generating SPE object files, hosted on the PPE. The SPE GCC back end currently supports C and C++ code generation, and implements support for vector data types and intrinsics. Instruction scheduling for the SPE processor's in-order dual-issue pipeline is also provided.

One feature of the SPE required GCC common code changes in order to achieve reasonable code quality. The processor features a deep pipeline, causing very expensive stalls on mispredicted branches. Furthermore, the SPE does not have any hardware branch prediction logic. Instead, the instruction set contains explicit branch-hint instructions. These instructions allow to specify a branch instruction address and the target address, and will cause the SPE pipeline to assume that the given branch is likely to branch to the specified target address. However, the branch hint needs to be executed several cycles *ahead* of the actual branch to allow it to take effect.

In some cases, the compiler can automatically generate branch hints, for example to mark the back-edge branch of a loop as likely taken. Profile-directed feedback data, if available, is also an excellent source to generate branch hints; the programmer can also manually specify such information by using the `__builtin_expect` function. To allow for even greater flexibility, this function was extended for the SPE GCC to accept a non-constant second argument. Just like for the regular version of `__builtin_expect`, the intended semantics is simply that the compiler should assume the value of the first argument is likely to be equal to the second argument. The builtin will be expanded to code that evaluates the second argument to compute the target address the branch controlled by `__builtin_expect` will jump to if the expectation holds indeed true, together with a branch hint instruction specifying that target address for this branch.

The SPE provides instructions that perform conditional assignments, commonly referred to as conditional moves. These can be used to avoid branch instructions in the first place in many cases. This type of optimization is currently performed by the if-conversion pass; however this pass only succeeds in a limited number of scenarios. The SPE GCC provides a more aggressive if-conversion pass that eliminates branches that have possibly several assignments within the branch-taken and/or fall-through blocks. Because of the high branch mis-prediction penalty, this pass has proven to significantly enhance SPE performance in certain applications.

The next step will be to port the SPE and PPE toolchains to a current version of GCC. In the case of SPE, this will make recent GCC features available that are likely to prove very beneficial on that platform; the autovectorization support introduced with GCC 4.0 is of course very important on the vector-centric SPE processor, and due to the large register set the Swing Modulo Scheduling software pipelining algorithm has the potential to significantly improve performance.

3.3 Future directions

As the SPE is able to access PPE memory, it could be useful to extend the object format to support accessing PPE symbols by name from an embedded SPE executable image. This would simplify development of applications employing closely interacting PPE and SPE components. A further step into that direction would be to support automatic generation of the DMA code sequences needed to actually access PPE variables by the SPE GCC back end. This would allow SPE code to directly use such variables.

Currently, PPE and SPE code generation require two different compile steps applied to two different source files. It may be desirable to investigate ways that would allow for even more seamless integration, e.g. by providing a compiler that would generate both PPE and SPE object code from a single source file, using annotations to decide which parts of the code to run on the PPE and which on the SPE. It might even be conceivable to have the compiler automatically select hot spots that would benefit most from running on the SPE, thus allowing applications to exploit SPEs by a simple recompile. All this would definitely require significant GCC common code changes.

4 Debugging options

To make application development for the Cell architecture feasible, good debugging tools are an essential requirement. In this section, we discuss a number of options how to extend GDB to provide debugging support on the Cell platform. As GDB currently does not support the notion of a platform with two distinct instruction sets, some GDB common code modifications will likely be required.

4.1 SPE-only debugging

The core requirement for GDB on the Cell architecture is to debug both PPE and SPE code. At a minimum, we need one GDB port able to debug PPE code and another to debug SPE code. While the existing GDB Linux on PowerPC target can be used to handle the PPE, a new target to support the SPE is required.

GDB provides two distinct interfaces to allow its behaviour to be adapted to a particular platform. The *architecture vector* contains routines that define the machine instruction set and ABI used by the program being debugged. The *target vector* contains routines that define how to access the debuggee's state information (e.g. register and memory contents) from whatever host platform GDB currently runs on. This distinction is made even in the common case where GDB is used to debug programs running on the same system as itself.

In order to allow GDB running on the Cell platform (i.e. as a PPE executable running under Linux on Cell) to debug an SPE thread, we need to implement architecture vector routines to handle the SPE instruction set and ABI as well as target vector routines to allow accessing SPE state.

At the time of this writing, we have patches against GDB 6.0 that implement these two interfaces. Work is currently underway to port them to current GDB mainline, and adapt them to changes in the kernel interface introduced by `spufs`. We hope to be able to release initial patches soon.

The SPE architecture vector routines are mostly straightforward. As with many targets, stack unwinding is currently implemented by analyzing the function prolog to detect the size of the allocated stack frame and the location of saved registers. It might prove beneficial to switch to using DWARF-2 Call Frame Information records instead to provide that information; however currently the tool chain does not yet provide that data.

While the SPE does provide a hardware single-step interface, this is currently not used by GDB. The main reason for this is that the hardware single-step does not always execute only one instruction; instead it will stop execution after one "issue group," which may consist of one or two instructions. (In addition, the current kernel interface does not actually provide a

means for GDB to activate the hardware single-step feature.) Instead, the current GDB implementation uses a software single-step mechanism that temporarily inserts a break point after the next instruction. If the next instruction is a branch, the code needs to be analyzed to determine the potential branch target addresses, and the single-step break point is then inserted at all those locations.

The SPE-specific target vector routines are the most interesting part. They need to implement access to SPE register and local store contents. As described previously, at the kernel level SPE threads are represented by the task ID of the PPE thread issuing the `spufs_run` ioctl; GDB uses `ptrace` to attach to that thread. Note that attaching to the PPE thread causes not only it, but also the associated SPE thread to stop. Continuing the inferior task using the `PTRACE_CONT` operation will cause the `spufs_run` ioctl to be restarted, and the SPE thread will continue. Thus, GDB run control works as usual. However, the `ptrace` interface does not provide direct access to SPE state information.

To access SPE state, GDB uses the `spufs` kernel interface directly. Using `ptrace`, GDB determines the parameters used by the PPE thread to invoke the `spufs_run` ioctl, and in particular the file handle it is performed on. By looking up the `/proc/`*`pid`*`/fd/`*`handle`* symbolic link, GDB can determine the directory name associated with this file handle, and open the `mem` and `regs` files in that directory to access SPE local store and general-purpose register contents. To get and set the current program counter value, GDB will use the `npc` member of the `spufs_run_args` data structure used as argument to the `spufs_run` ioctl. This can be implemented by overriding the target vector `target_fetch_registers` and `target_store_registers` routines to access registers, and the `target_xfer_memory` routine to access local store memory.

4.2 GDB child-session support

The SPE-only debugger described in the previous section does allow the user to debug a single SPE thread. However, this is a very unsatisfactory way of debugging a real Cell application that consists of PPE and SPE components closely interacting. One means of improving that user experience, while still keeping SPE and PPE two separate targets from a GDB implementation perspective, could be to integrate multiple SPE-gdb and PPE-gdb debugger processes attached to the various component threads of a Cell process into a single overall user interface.

Similar as for the SPE-only debugger, we currently have an implementation of that idea, and are working to adapt it to recent kernel and GDB changes in preparation for release. To use the current implementation, an instance of GDB configured for the PPE target is invoked on the main PPE executable. This allows PPE debugging as usual. In addition, this GDB instance is modified to intercept generation of SPE threads by the process being debugged. This could be implemented with the help of kernel support, using a `ptrace` event similar to the `PTRACE_EVENT_FORK` family of events supported by recent Linux releases. If the kernel does not provide such a facility, the debugger can instead install break points on known library functions like `spu_create_thread` similar to how shared library event are handled. This assumes the debugged process only uses the standard library to create SPE threads, of course.

Once it has intercepted a SPE thread creation event, the main debugger can then launch an instance of the SPE-only debugger that will attach to the new SPE thread. At this point the

current implementation offers two choices how to allow the user to interact with the newly started debugger. In "launch mode," some external program is used to host the SPE debugging session; this can be either an `xterm` terminal session, or a GDB-wrapping GUI like DDD. The user can configure via the `LAUNCH_PLATFORM` environment variable or a command line option which host program is to be used. From then on, the user interacts with the SPE debugger via that host program.

In "passthru mode," the PPE debugger itself acts as host to the SPE debugging session, by intercepting the standard input and output streams of the SPE debugger. User input is forwarded to the currently active SPE child debugger, and its output is presented back to the user. New GDB commands allow the the user to choose one of possibly multiple SPE child debugging sessions to work with at any moment; it is also possible to bypass the command forwarding and apply commands back to the PPE debugging session. Note that GDB will not attempt to automatically coordinate the various debugging sessions; for example, starting or stopping a single child debugging session will not automatically start or stop the other sessions as well. Passthru mode requires significant changes to GDB common code to coordinate the flow of commands between the various child debugging sessions. The current implementation, while useful for initial debugging support in a simulator environment, is likely not suited for upstream integration; we will probably drop this mode in the future.

4.3 GDB multi-architecture target

The best solution to debug Cell applications, however, would appear to be an integrated GDB target that by itself supports both PPE and SPE instruction sets and ABIs. Using such a target, the user should be able to debug a full process, including multiple threads any of which might currently be executing code in a logical SPE context or executing PPE signal handler code that happened to interrupt SPE execution, from within a single GDB session. Stack backtraces should show PPE and SPE frames properly intermixed. Disassembly of both PPE and SPE code should work transparently. Symbol and line-number resolution should work correctly for all PPE and SPE code. The complete GDB user interface should ideally be absolutely unchanged, allowing all GDB-wrapping debuggers (like DDD) to work without changes.

At the time of this writing, we do not yet have implemented this solution. The remainder of this section presents a proposal how a target along these lines could be implemented within the current GDB framework, and which interfaces to common code might need to be modified in order to make this approach possible.

Accessing process state

The debugger requires access to a variety of process-related information in order to operate, collectively called "process state" here. For a regular Linux PPE process, the process state includes its memory image, register file contents (one set for each thread of the process), and some other kernel meta-data, like the hardware watchpoint or other control registers.

For a Cell process using one or more logical SPE contexts provided by `spufs`, however, the process state includes in addition information about each SPE context the process is currently using (i.e. has an open file handle to). This information includes the local store memory image, register file contents, and other kernel meta-data, like run status information or executable-file mapping data.

The debugger needs to be able to retrieve, and in some cases manipulate, the state of the process being debugged. GDB currently supports three major ways of doing this: native debugging, remote debugging, and core-file debugging. As we want to support all three modes of operation to debug Cell processes, all three methods have to be enhanced to support accessing SPE state in addition to the traditional process state.

Native debugging The debugger runs as a process on the same system where the debuggee runs. GDB will use operating system interfaces provided by the kernel (e.g. the `ptrace` system call and/or the `/proc` file system) to access process state of the debuggee. To allow for native debugging on Cell, the kernel provides interfaces to allow accessing SPE state for a logical SPE context in use by the inferior process. The SPE context is identified using the `spufs` file handle used by the inferior to access it. GDB can directly access the associated `spufs` files to manipulate process state, just as the previously described SPE-only GDB back end does.

Remote debugging The debugger runs on a host system and is used to debug a process running on another target system. GDB will interface, typically over the network or a serial link, with a remote debugging stub provided by the target system. The stub can either be implemented by a user space application running on the target system (e.g. the gdbserver application provided with GDB), or else be implemented as part of the operating system, boot monitor, or other system component. The remote debugging interface provides operations to access process state. To allow for remote debugging on Cell, that remote protocol needs to be extended by new operations that allow to access SPE state. The gdbserver application would use same interfaces GDB uses in native mode to implement the new operations.

Core-file debugging After abnormal termination of a process, the kernel writes the full process state to a core file. GDB reads the core file to retrieve the process state. The format of the core file is a variant of ELF; the memory image is represented by LOAD segments, and register set information is added in the form of NOTE segments. To allow for core-file debugging on Cell, the kernel needs to be enhanced to add SPE state to core files it generates. The core file will contain additional NOTE segments for each SPE context used by the process, containing their local store images and register sets (and potentially meta-data).

Determining executable file mappings

One important part of SPE state meta-data is the path to the SPE ELF executable image that was initially loaded into the logical SPE context. This is required by the debugger to resolve symbolic information for SPE local store code and data. The same information may also be required within the kernel, for example to allow tools like `oprofile` to collect symbol information for profile samples collected during SPE execution.

In the SPE-only debugger case, the debugger is invoked with the path to executable image already specified on the command line, so no further information is required. However, in the multi-architecture scenario, the main executable specified thus would be the PPE executable; therefore it will be necessary to retrieve the file mapping information for all logical SPE contexts by some other means.

We have not yet finally decided on how that information will be passed. One option would

be to have the SPE loader maintain a list of all loaded SPE executables in main storage, similar to how the shared library loader `ld.so` maintains a list of all loaded shared libraries. That list could be traversed by the debugger. One disadvantage of that solution is that it will not cover the case where one process attaches to a `spufs` context initialized by another process.

The second option, which would also handle that more general case, would be to store the executable file mapping information within the `spufs` directory itself. Similar to how the `/proc` directory associated with a process contains a symbolic link `exe` pointing back to the process' main executable, the `spufs` directory associated with a SPE context could contain such a link. However, as the kernel is not directly involved with loading SPE executables, that link would likely have to be created by the user space SPE loader itself. Note that to allow for remote debugging, the gdbserver protocol would require an extension to retrieve that path; and to enable core-file debugging, the kernel would need to store the path into the core as well (presumably into yet another NOTE segment per SPE context).

GDB implementation issues

The single-target GDB implementation supporting both PPE and SPE is likely to require some changes to GDB common code. We will describe in the remainder of this section how such a target could be integrated into the current GDB structure, and where we would need some changes.

Address space GDB fundamentally assumes an internal representation of target addresses that consists of a single flat address space per process. However, it allows the target back end to provide mappings between this GDB internal format and a more general target-specific situation, using the `POINTER_TO_ADDRESS` and `ADDRESS_TO_POINTER` macros. (The AVR is an example of a target currently supported by GDB that uses multiple address spaces.)

It should be possible to employ this feature to define an internal address representation able to encompass both the PPE address space and SPE local stores. That internal representation would encode addresses in the PPE address space by a marker bit of zero followed by the target address. An address in a logical SPE context's local store would be encoded by a marker bit of one followed by the SPE identifier (file handle) and the local store address. This encoding assumes a 64-bit GDB internal address space.

```
|0|       PPE address     |
--------------------------
|1|  SPE id  | LS address |
--------------------------
| |  31-bit  |   32-bit   |
```

Register set The GDB internal register set for the Cell back end would be defined as the union of the PPE register set and the SPE register set, plus a pseudo register representing the current execution state. For each thread, the contents of the PPE subset of the register set would have their usual meaning; in particular, if the thread is currently executing on an SPE, the PPE register set would represent the state of the PPE at the time it executed the `spufs_run` ioctl. The contents of the SPE subset of the register set would hold the current SPE register contents if the thread is currently executing on an SPE, and be in an undefined state otherwise. The extra pseudo register would hold the identifier (file handle) of the logical SPE context that is currently executing, or a NULL value if the thread is currently executing on the

PPE. The function of the extra pseudo register can most likely be subsumed into the virtual PC register—the PC would hold a GDB internal address that by itself identifies the current execution state.

Symbol resolution To allow for symbol and line-number resolution for the SPE executable loaded into a SPE context, the most natural way would appear to be to treat them as a type of "shared libraries" to the main process. We would implement a new `target_so_ops` class that retrieves the executable-mapping meta-data for each SPE context (see above), and provides the information to GDB's symbol table routines, with a relocation offset appropriate to map the file at its proper GDB internal address representing its location in SPE local store.

Disassembly listing To enable proper multi-architecture disassembly, the back end can register a disassembler routine `print_insn` that would check the top bit of the GDB internal memory address passed to it and dispatch the call to either the PPE or the SPE disassembler accordingly.

Execution control Due to way `spufs` execution control works, the debugger should be able to attach to and detach from a SPE thread using the normal `ptrace` interfaces applied to the corresponding PPE thread; stopping and starting SPE threads is handled likewise. Should we want to support hardware single-step on the SPE, the most natural implementation would be for the kernel to treat a `PTRACE_SINGLESTEP` into a `spufs_run` ioctl as a request to single-step the SPE. Thus, the debugger could just continue the use the `ptrace` interface for this action as well.

Stack unwinding We would need to provide stack frame sniffers and unwinders for both PPE and SPE architectures. By checking the top bit of the GDB internal address of the virtual stack pointer register we would ensure that only the appropriate sniffer for the architecture would trigger on any given frame. If we want to support mixed backtraces representing the transitions between PPE and SPE code by means of the `spufs_run` ioctl and signal handler invokations, this can be handled similiarly to the way signal trampoline frames are currently recognized: For the PPE to SPE transition, the signal trampoline sniffer would recognize that the `spufs_run` ioctl was interrupted by the signal, and unwind to the frame described by the SPE register set for the SPE file handle used with the ioctl, in addition to performing the normal unwind for the PPE register set. For the SPE to PPE transition, a special sniffer would recognize the top of the SPE stack and unwind to the frame described by the PPE register set.

Inferior function calls A call to a PPE function while currently executing PPE code works just as today. A call to an SPE function while currenly executing SPE code can be implemented in the straightforward way like the current SPE-gdb does. A call to a PPE function while currently executing SPE code should also work; the dummy frame on the PPE stack will act as PPE/SPE boundary like a signal trampoline frame. A call to an SPE function while currently executing PPE code would require construction of a dummy `spufs_run` ioctl invocation.

Native target function overrides At the process stratum level, the functions `target_fetch_registers` and `target_store_registers` would need to be implemented as follows: First the PPE registers are fetched or stored as usual. Then the PPE register contents

are checked to determine whether the thread is currently executing an `spufs_run` ioctl, as described for the SPE-only case. If so, the SPE registers of the respective logical SPE context are fetched or stored. The `target_xfer_memory` function would need to be implemented as follows: If the address to be transferred has its top bit clear, perform a memory transfer to or from PPE main storage. If the address has its top bit set, transfer to or from SPE local store on the SPE identified within the address. Note that functions from the other target strata should not require any overrides. In particular all handling of threads should just work using the regular Linux thread target functions.

Core-file target function overrides To enable handling of combined PPE/SPE core files as described above, the core stratum target functions will have to be adapted similarly to the native target functions. In particular, they will have to be able the SPE register file and SPE local store NOTEs provided by the core file. It appears that some GDB common code changes will be required to implement this.

Architecture vector overrides Several of the functions implemented by a GDB target by providing an architecture vector callback routine have already been described above. For the other architecture vector callbacks, it should be possible to either just share them between PPE and SPE code, or else allow them to determine from looking at an address passed as argument whether to apply PPE or SPE architecture methods. However, it may turn out that some architecture methods cannot be correctly implemented in the mixed-architecture scenario; in this case some GDB common code changes may be required.

Test suite To provide a high-quality GDB port it is essential to be able to run the extensive GDB test suite. Running the test suite natively will ensure the PPE part of the Cell GDB still works. Running the test suite using an SPE compiler is necessary to ensure the SPE part of GDB works as well; this comprises e.g. stack frame unwinding and inferior call support. To make this happen requires that GDB is able to start SPE-only binaries. This could be achieved either by kernel support allowing to directly execute an SPE executable, or else by exploiting the remote-target features of the test suite harness to interpose a SPE loader binary. To test mixed-mode debugging, new test cases will certainly be required as well.

5 Conclusion

The Cell processor requires a programming model that differs in significant aspects from those used with previous microprocessors; corresponding changes in the GNU tool chain are needed to enable a high-quality software development environment for Cell. We are convinced that both Linux and the GNU tool chain will be of great relevance on this new platform, and are looking forward to working with the GNU developer communities to integrate Cell architecture support into mainline source trees soon.

Acknowledgements

The design of the Cell microprocessor as well as the programming models for Cell that form the basis of what I have presented in this paper are the work of a large number of people based at the STI Design Center in Austin and other Sony, Toshiba, or IBM locations around

the world. I am grateful to Russell Olsen (Sony Computer Entertainment), Sidney Manning (IBM Austin) and Arnd Bergmann (IBM Böblingen) for reviewing this paper.

References

[1] D. Pham et al., "The Design and Implementation of a First-Generation CELL Processor," *ISSCC Dig. Tech. Papers*, Paper 10.2, pp. 184–185, Feb. 2005.

[2] B. Flachs et al., "A Streaming Processor Unit for a CELL Processor," *ISSCC Dig. Tech. Papers*, Paper 7.4, pp. 134–135, Feb. 2005.