Class Objectives – Things you will learn

- Get to know the toolchain for the Cell BE
- Learn how gdb can be used on the Cell BE architecture to debug applications
- Credits
  - Ulrich Weigand
  - Sidney Manning

**Trademarks** - Cell Broadband Engine ™ is a trademark of Sony Computer Entertainment, Inc.
Class Agenda

- Linux on Cell BE
- GNU tool chain for Cell BE
- GDB – The GNU Debugger
- Using GDB on Cell
  - Kernel Debugging
  - Debugging SPE threads
  - Debugging Cell BE applications
  - Debugging Capability of Systemsim
- Graphical Debugger Front-End DDD
Linux on Cell BE
SPE exploitation – kernel level API

- Indirect programming model
- Virtual file system namespace (“spufs”)
  - Directories represent virtual SPE contexts
  - Files provide access to SPE resources
  - Memory, registers, mailboxes, signals, ...
- SPE execution control
  - Provided by spufs_run system call
  - Synchronous execution on behalf of PPE thread
  - PPE address space used for SPE DMA
  - Virtual SPE contexts scheduled onto physical SPEs
SPE exploitation – PPE programming interfaces

- Asynchronous SPE thread API ("libspe")
  - `spe_create_thread`, `spe_wait`, `spe_kill`, ...
  - Intended to be portable across operating systems
  - On Linux, implemented on top of spufs kernel API

- Implementation of `spe_create_thread`
  - Allocate virtual SPE context in spufs
  - Load SPE application code into context
  - Start PPE thread using `pthread_create`
  - Within new thread, commence SPE execution (spufs_run)
SPE exploitation – SPE programming interfaces

- Application start-up and termination
  - Start-up code sets up environment
  - Entry point called with arguments from spe_create_thread
  - “Stop-and-signal” instruction signals termination

- Language and library features
  - C support, (limited) C++ support available
  - Vector data types & intrinsics, SPE assembler intrinsics
  - Limited library facilities available
  - “System calls” to offload processing to PPE
SPE execution control

- Application
- Kernel
- SPE

spufs_run
set Runnable
stop and signal
return
SPE execution control – signals

Similar flow of control for PPE assisted SPE system calls

Application

Kernel

SPE

signal handler invocation

“trap”

spufs_run system call restart

set RUNNABLE
GNU Toolchain for Cell BE
GNU tool chain

- PPE support
  - Just another PowerPC variant ...

- SPE support
  - Just another embedded processor ...

- Cell BE support
  - More than just PPE + SPE !
Object file format

- **PPE**: regular ppc/ppc64 ELF binaries
- **SPE**: new ELF flavour EM_SPU
  - 32-bit big-endian
  - No shared library support
  - Manipulated via cross-binutils tools
- **Cell**: combined object files
  - SPE executable embedded as a whole by embedspu tool
  - Contained in .rodata.spuf elf section in PPE object
  - CESOF: support SPE->PPE symbol references
GCC support

- **PPE**: handled by rs6000 back end
  - Processor-specific tuning, pipeline description
- **SPE**: new spu back end
  - Built as cross-compiler
  - Handles vector data types, intrinsics
  - Middle-end support: branch hints, aggressive if-conversion
  - Future: GCC 4.x port exploiting auto-vectorization, SMS
- **Cell**: no special support today
  - Future: single source mixed-architecture compiler
SPE architecture features

- **Instruction set architecture**
  - 128-bit SIMD execution unit (integer + float/double)
  - Large unified register file (128 general-purpose registers)
  - Branch penalty avoidance – conditional move, branch hint
  - “User mode” only – channel instructions for I/O

- **Memory architecture**
  - Software-managed memory (256 KB local storage + DMA)
  - No address translation or page protection
  - Load/store architecture - 16-byte alignment enforced
  - *Local storage access cannot trap!*
SPE Application Binary Interface

- **Register usage**
  - R0: link register, R1: stack pointer, R2: volatile
  - R3-R79: function arguments & return value, volatile
  - R80-R127: local variables, non-volatile

- **Stack frame**

![Diagram of stack frame](image)
GDB – The GNU Debugger
GDB – The GNU Debugger

- Core features
  - Run, stop, and continue target applications
    - Breakpoints, watchpoints, and catchpoints
    - Alter execution, e.g. perform inferior function calls
  - Examine and modify target application state
    - Register and memory contents, stack backtraces
    - Symbolic debugging, high-level language support
  - Modes of operation
    - Native: run application on same system as debugger
    - Remote: attach to remote target system via stub
    - Post-mortem: analyze core files after crash
GDB implementation – structure

- Basic structure
  - User interface (line mode, text mode, GUI)
  - Symbol side ("everything you can do without live process")
  - Target side (access to/manipulation of process state)

- Target side back-end interfaces
  - Architecture vector
    - Instruction set, registers, ABI, stack unwinding, ...
  - Target vector
    - Access to memory & registers, execution control, ...
GDB implementation – architecture vector

- Instruction set
  - Disassembler, breakpoints, software single-step

- Registers
  - Names, sizes, contents of special registers

- ABI
  - High-level language data types
  - Function calling convention (for inferior calls)

- Stack unwinding
  - Prologue analyzers, stack frame detection
GDB implementation – target vector

- Process control
  - Start/stop processes
  - Attach to/detach from running processes

- State manipulation
  - Access inferior registers & memory
  - Manage threads of inferior process

- Execution control
  - Resume execution, single-step, wait for events
  - Manage HW-assisted breakpoints, watchpoints, ...
GDB implementation – Linux target

- Target vector
  - Generally based on the ptrace system call
    - PT_ATTACH/PT_DETACH for process control
    - PT_CONT/PT_SINGLESTEP for execution control
    - PT_PEEKDATA/PT_POKEDATA to access memory
    - PT_PEEKUSR/PT_POKEUSR to access registers
  - Analyze target libpthread data structures for thread control

- Architecture vector
  - Depends on architecture Linux is running is on
  - More than 20 architectures supported
Using GDB on Cell
Using GDB on Cell - Overview

- **Kernel debugging**
  - Downloading and cross compiling GDB for PPC
  - Kernel configure options
  - Attaching and debugging the Linux kernel using System Sim’s debug port

- **Debug Native executable running on the Simulator**
  - Debugging options
  - Use of command scripts to track variables

- **Debug a standalone SPU application**
  - It’s just as easy debugging a native application

- **Debug a BE application**
  - Environment variables required
  - Limitations

- **Simulation Features**
  - Tracking memory corruption

- **gdb documentation** [http://www.gnu.org/software/gdb/documentation]
Kernel Debugging
Kernel Debugging - Overview

- Cell sdk allows to debug a cell Linux kernel running inside the simulator
- gdb version supporting PowerPC64 is required
  - on PowerPC systems → already installed
  - on Intel systems → download and compile
- Kernel needs to be compiled with debug option
Download and build gdb for 64bit PowerPC Architecture

# Commands to download and build gdb for ppc64.

    mkdir -p base
    mkdir -p obj
    tar jxvf base/gdb-6.3.tar.bz2
    pushd obj
    ../gdb-6.3/configure --target=powerpc64-linux
    make all
    make install
    popd obj
Preparing the Linux Kernel for debug on x86 systems

- **Change working directory to objsim directory**
  
  ```
  cd <SDK_ROOT>/cbe_linux/src/kernel/objsim
  ```

- **Configure the Kernel**
  
  ```
  ARCH=powerpc PLATFORM=cell \\
  CROSS_COMPILER=/opt/sce/toolchain-3.2/ppu/bin/ppu \\
  make xconfig
  ```

  - Go to Kernel Hacking / Kernel Debugging
  - Check box “Compile the kernel with debug info”

- **Disable Optimization for selected files**
  
  ```
  CFLAGS_spu_ptrace.c = -O0 -g
  ```

- **Rebuild the kernel**
  
  ```
  cd <sdkroot>/cbe_linux/src/kernel directory
  make
  ```

- **this will install the kernel into <sdkroot>/systemsim-cell-release/images/cell**
Starting a Kernel Debug Session

- **Start simulator with new kernel**
  - path to kernel loaded is
    - current directory or `<sdkroot>/systemsim-cell-release/images/cell`

- **Attach to the simulator’s debug stub**
  - click “Service GDB” in the simulator’s main window
  - `cd <sdkroot>/cbe_linux/src/kernel/objsim`
  - Start GDB using graphical front-end `ddd`
    - `ddd -debugger ./gdb \ cbe_linux/src/kernel/objsim/vmlinux`
  - Set breakpoint and attach `gdb` to the simulator
    - `(gdb) break start_kernel`
    - `(gdb) target remote :2345`
    - `(gdb) continue`
GDB with Breakpoint in Kernel
Native Debugging

- debugging an executable running in the simulator
- debugging on Cell BE
Debugging SPE threads

- **Current implementation: spu-gdb**
  - Available as part of the Cell BE SDK

- **Supported operating modes**
  - Debug stand-alone SPE binary
    - Kernel support via binfmt_misc required
    - Allows to execute the full GDB test suite
  - Attach to single SPE thread of running Cell application
    - Use simplified by debug assists in libspe runtime: print Linux PID of SPE thread on startup and wait for GDB attach
  - Remote debugging via gdbserver
SPE debugger – process state access

Access PPE state via ptrace, and analyze it

Result: PPE thread is blocked inside spufs_run system call on spufs directory D

Access SPE state of context in spufs directory D via spufs file operations
SPE debugger – execution control

Debugger

Application

Kernel

SPE

return from ptrace call

ptrace (PT_CONT)

“breakpoint”

spufs_run system call re-issued

set RUNNABLE

Handling identical to PPE thread execution control
SPE debugger – executable file mapping

- Problem: Which executable to load symbols from?
  - PPE executable: specified on gdb command line
  - PPE shared object: retrieved from ld.so data structures
  - SPE executable: retrieved from libspe data structures

- How does it work?
  - Directory name in spufs encodes libspe thread ID
  - Thread ID points to data structure in PPE memory
  - Data contains spe_program_handle_t used to start SPE
  - Handle refers to SPE executable present in PPE memory
SPE debugger – runtime library debug assists

- **SPU_INFO=1**
  - Implemented within libspe runtime library
  - When loading SPE ELF executable, prints message
    Loading SPE program : NNN
    SPU LS Entry Addr   : NNN
  - Before starting up new SPE thread, prints message
    Starting SPE thread 0x..., to attach debugger use:
    spu-gdb -p NNN

- **SPU_DEBUG_START=1**
  - Includes everything done by **SPU_INFO=1**
  - Waits until debugger is attached (or signal received)
Debugging an Executable Running in the Simulator

- Debugging can be done locally or remote
- `gdb` for sdk 1.0.1
  - `rpm spu-gdb-2.3-bsc2.1.ppc.rpm`
  - is installed on sysroot disk (gets mounted in simulator)
- If you downloaded `gdb` to debug the kernel you can configure `gdbserver` to debug remotely.
  
  ```
  CC=/opt/sce/toolchain-2.3/ppu/bin/ppu32-gcc
  ../../../gdb-6.3/gdb/gdbserver/configure --target powerpc-linux
  ```
  - Use the same debugger you used to debug the kernel to drive the remote session.
  - Remember that bogus net must be configured or remote debug sessions will not work.
Standalone SPU debugging

- very similar to debugging a PowerPC application
- the Cell SDK 1.0.1 ships with spu-gdb preloaded
  `spu-gdb <program_name>`
- easy way to get a handle on spu programs that have no external dependencies
- good way to learn what some of the vector intrinsic operations do to your data
Debugging SPU binaries
Debugging Cell BE Applications
BE application Debugging

- Must use either native gdb or spu-gdb depending on what part of the application you are debugging.
  - Use standard gdb to debug the PowerPC threads.
  - Use spu-gdb to debug the SPU threads.
  - `SPU_DEBUG_START` – Tells the libspe to start your spu threads in the stopped state.
    
    ```
    SPU_DEBUG_START=1 <be_program> &
    ```
  - start you application in the background to be able to attach to the spu thread

- Limitations:
  - Can only attach to one thread at a time.
Example: Bus Error due to DMA

```
Example: Bus Error due to DMA

Program received signal SIGBUS, Bus error.
0x00000000 in get_work (buffer=0x00000000, ef_addr=0xffffd000)
```

```
if (!buffer[i]) {
  perror("memory low");
  return -1;
}
memset (buffer[i], 0, DMA_SIZE);

speid[i] = //spe_create_thread (speid[i], spu_dramo, buff[i], NULL, 0, 0);
printf ("Failed spe_create_thread(speid[i], spu_dramo, buff[i], NULL, 0, 0)\n");
```
Native Debugging on CBEA

- Debugging can be done locally or remote
- The same mechanisms are available as for using gdb on Systemsim
  - `spu-gdb <spu_binary>`
  - `SPU_DEBUG_START=1 <be_program> &`

- Debuggers
  - `/usr/bin/gdb` gdb for PPU
  - `/usr/bin/spu-gdb` gdb for SPU

- rpm for spu-gdb
  - `spu-gdb-3.2-2.ppc.rpm`
Using the graphical Front-End DDD
Graphical Debugger Front-End DDD

- **Prerequisites general Linux system**
  - Fedora Core 5 including X11 (install via yum)

- **Prerequisites Cell BE**
  - the Cell SDK 1.0.1
    - (http://www.bsc.es/projects/deepcomputing/linuxoncell/)
    - sdk 1.0.1 includes the debugger
  - gcc for PPU and SPU
  - gdb for PPU and SPU
  - glibc
  - ddd

- **Documentation**: http://www.gnu.org/manual/ddd/

For rpm package numbers see next page
### PRMs tested with FC5

#### Gcc

```
[root@localhost yum.repos.d]# rpm -qa | grep gcc
gcc-4.1.0-3
libgcc-4.1.0-3
libgcc-4.1.0-3
ppu-gcc-3.2-2
spu-gcc-3.2-2
ppu-gcc-c++-3.2-2
spu-gcc-c++-3.2-2
```

#### Glibc

```
[root@localhost yum.repos.d]# rpm -qa | grep glibc
glibc-devel-2.4-4
glibc-common-2.4-4
glibc-2.4-4
glibc-headers-2.4-4
glibc-devel-2.4-4
glibc-2.4-4
glibc-kernheaders-3.0-5.2
```

#### Gdb

```
[root@localhost yum.repos.d]# rpm -qa | grep gdb
spu-gdb-3.2-2
gdb-6.3.0.0-1.122
gdb-6.3.0.0-1.122
```

#### Ddd

```
[root@localhost ~]# rpm -qa | grep -i DDD
ddd-3.3.11-5.2
```
Check that SPU support is active

- **spufs must be mounted**
  - not necessarily by default for the latest kernel versions
  - Check with `mount`
    - will show all mounted file systems
    - Should include “/spu on /spu type spufs (rw)”

- **if spufs is not mounted**
  - check if /spu exists, if not `mkdir /spu` as root
  - mount `-t spufs /spu /spu`
  - or add to `/etc/fstab`
Debug PPU and SPU Applications

- **PPU application**
  - debug as usual

- **SPU application**
  - Start application with debug support for SPU
    - SPU_DEBUG_START=1 <PPU-Program> &
  - change into the spu source file directory
  - start the debug session
    - Assuming that spu-gdb is installed in /usr/bin
    - ddd --debugger /usr/bin/spu-gdb simple_spup -p 24701
Example: Starting Cell BE Application for Debug

![Command to start Cell BE application]

- SPE thread started, command for debugging
Example: Starting DDD

Starting the `ddd` with `spu` support
Example: DDD Window

- Source file, included in sdk
- breakpoint
- ddd command window
- ddd source file window
- spu-gdb command window