LLDB for your hardware:

Remote Debugging the Hexagon DSP

Colin Riley – Games Technology Director

Outline

- Introductions
- Adapting LLDB for your hardware
  - The Why and How?
  - The 3 steps
- Summary
- Q&A

Introductions

- Hello!
  - I'm Colin Riley
  - Games Technology Director at Codeplay
  - Games? Isn't this the LLDB talk?
  - Background in Games Technology
    - Lately been interested in working with debuggers
    - worked with LLDB last 18 months on customer projects
    - Wrote a specialised PlayStation®3 debugger (non LLDB)

- Heterogeneous compiler experts
- 35 talented engineers
- Based out of Edinburgh, Scotland
- We work on
  - R&D (both self and externally funded)
  - Work for hire, games fire fighting, compiler tech
  - Standards via bodies such as Khronos & HSA

Edinburgh Castle

Introductions – The Hexagon Connection

- We are creating an LLDB-based debugger for Hexagon
- Hexagon an incredible DSP
  - Information available on Qualcomm developer portal
  - "Porting LLVM to a Next Generation DSP"
    - LLVM DevMtg 2011, Taylor Simpson of Qualcomm Innovation Center, Inc

Introductions – The Hexagon Connection

- We are creating an LLDB-based debugger for Hexagon
- Development still ongoing
  - Remote debugger
  - Linux and Windows LLDB hosts
  - Eclipse Integration
- Talk is about adapting LLDB
  - using hexagon only as example

QuIC QUALCOMM INNOVATION CENTER, INC

HEXAGON™

Adapting LLDB for your hardware

Why?

Adapting LLDB for your hardware – Why?

- Debuggers essential part of any SDK
- Fast, advanced debuggers are demanded
- LLDB is the perfect balance of
  - Performance
  - Clean architecture, extendable
  - Leverages much from LLVM
- Lots of other reasons

How?

- The three steps to debugger implementation
  1. Binary and debugging information parsing
  2. Target system state and control
  3. Interpretation of the two previous steps
     - Advanced features require using both sets of data
     - Extensive work here, the difference between a debugger and a *useful* debugging experience

Step 0

(Before we begin)

LLVM/Clang support for your target

Step 0 – LLVM/Clang support for target

- Can hack around this in some ways...
- But disassembler is a must – LLDB uses it
  - Hexagon disasm is in development by Qualcomm Innovation Center, Inc - will be upstreamed
- For expression evaluation, need the frontend enabled for whatever language you are debugging

Step 0 – LLVM/Clang support for target

- Another reason why teams should be staying near tip of LLVM/Clang
- Some work may be needed at an API level to integrate older versions of LLVM with LLDB
- Could result in some nasty issues too
- Hexagon tracks tip, so we are onto a winner

Step 1

Binary and debug information parsing

- What do we need to load?
  - The binary sections/symbols/debug information
  - LLDB already supports ELF & Mach-O out of the box
  - In terms of debugging information, DWARF supported
    - Features being added all the time
  - My experience simply with ELF & DWARF

- However, if you are not using a supported format:
- ObjectFile
  - If you need to add your object file format, need to extend this interface
    - Can refer to ObjectFileELF
- For debuginfo, look at SymbolFileDWARF

Step 1: Binary and debug information parsing

- Hexagon is ELF & DWARF
  - Job done?
- We still need to ensure the architecture lines up
  - Allows LLDB to understand the binary is for Hexagon
  - Uses LLVM target information

Step 1: Binary and debug information parsing

- ELF Architecture definitions
  - Very simple changes
- Really only 3-4 lines in ArchSpec.cpp

```
--- a/source/Core/ArchSpec.cpp
+++ b/source/Core/ArchSpec.cpp
@@ -104,6 +104,11 @@ static const CoreDefinition g_core_definitions[ArchSpec::kNumCores] =     {
eByteOrderLittle, 8, 1, 15, llvm::Triple::x86_64 , ArchSpec::eCore_x86_64_x86_64  , "x86_64"    },
+    { eByteOrderLittle, 4, 4, 4, llvm::Triple::hexagon , ArchSpec::eCore_hexagon_generic,    "hexagon"   },
+    { eByteOrderLittle, 4, 4, 4, llvm::Triple::hexagon , ArchSpec::eCore_hexagon_hexagonv4,  "hexagonv4" },
....
+    { ArchSpec::eCore_hexagon_generic , llvm::ELF::EM_HEXAGON, LLDB_INVALID_CPUTYPE, 0xFFFFFFFFu,
0xFFFFFFFFu }  // HEXAGON
```

- g_core_definitions, g_elf_arch_entries, cores_match(), Thread::GetUnwinder()

- Is that it?
    - We can test
- Create a target with a binary from it

```
(lldb) target create hello_sample
Current executable set to 'hello_sample' (hexagon).
(lldb)
```

Step 1: Binary and debug information parsing

- Inspect the image sections

```
(lldb) image dump sections

Sections for 'hello_sample' (hexagon):
  SectID     Type             File Address                             File Off.  File Size  Flags      Section Name
  ---------- ---------------- ---------------------------------------  ---------- ---------- ---------- ----------------------------
...
  0x00000005 code             [0x0000000000005000-0x000000000000b070)  0x00005000 0x00006070 0x00000006 hello_sample..text
...
  0x0000000b data             [0x000000000000e018-0x000000000000e6c8)  0x0000d018 0x000006b0 0x00000003 hello_sample..data
  0x0000000c zero-fill        [0x000000000000e700-0x000000000000f240)  0x0000d6c8 0x00000000 0x00000003 hello_sample..bss
  0x0000000d dwarf-info                                                0x0000e110 0x00000091 0x00000000 hello_sample..debug_info
  0x0000000e dwarf-abbrev                                              0x0000e1a1 0x0000005d 0x00000000 hello_sample..debug_abbrev
  0x0000000f dwarf-line                                                0x0000e1fe 0x00000045 0x00000000 hello_sample..debug_line
  0x00000010 dwarf-frame                                               0x0000e244 0x00000040 0x00000000 hello_sample..debug_frame
  0x00000011 dwarf-str                                                 0x0000e284 0x0000007a 0x00000030 hello_sample..debug_str
...
  0x00000013 elf-symbol-table                                          0x0000e6c4 0x000023f0 0x00000000 hello_sample..symtab
...
(lldb)
```

Step 1: Binary and debug information parsing

- Inspect the image source line maps

```
(lldb) image dump line-table hello.c
Line table for /home/user/code/hexagon/hexagon-tools/tests/hello.c in `hello_sample
0x000050c0: /home/user/code/hexagon/hexagon-tools/tests/hello.c:4
0x000050c4: /home/user/code/hexagon/hexagon-tools/tests/hello.c:5
0x000050d0: /home/user/code/hexagon/hexagon-tools/tests/hello.c:6
0x000050d8: /home/user/code/hexagon/hexagon-tools/tests/hello.c:9
0x000050dc: /home/user/code/hexagon/hexagon-tools/tests/hello.c:10
0x000050e8: /home/user/code/hexagon/hexagon-tools/tests/hello.c:11
0x000050f0: /home/user/code/hexagon/hexagon-tools/tests/hello.c:12
0x000050f4: /home/user/code/hexagon/hexagon-tools/tests/hello.c:15
0x0000510c: /home/user/code/hexagon/hexagon-tools/tests/hello.c:16
0x0000511c: /home/user/code/hexagon/hexagon-tools/tests/hello.c:17
0x00005128: /home/user/code/hexagon/hexagon-tools/tests/hello.c:18
0x0000513c: /home/user/code/hexagon/hexagon-tools/tests/hello.c:19
0x00005150: /home/user/code/hexagon/hexagon-tools/tests/hello.c:20
0x0000516c: /home/user/code/hexagon/hexagon-tools/tests/hello.c:21
0x00005184: /home/user/code/hexagon/hexagon-tools/tests/hello.c:21

(lldb)
```

- Try setting some breakpoints

```
(lldb) b hello.c:20
Breakpoint 1: where = hello_sample`main + 92 at hello.c:20, address = 0x00005150

(lldb) b main
Breakpoint 2: where = hello_sample`main + 24 at hello.c:16, address = 0x0000510c
```

- Can see these match up with line table and symbols

```
0x00005150: /home/user/code/hexagon/hexagon-tools/tests/hello.c:20

[   428]     445    X Code       0x00000000000050f4      0x0000000000000090 0x00000012 main
```

- Or does it?

Step 1: Binary and debug information parsing

- Try setting some breakpoints

```
(lldb) b main
Breakpoint 1: where = hello_sample`main + 24 at hello.c:16, address = 0x0000510c
```

- Can see these match up with line table and symbols

```
[  428]     445    X Code       0x00000000000050f4       0x0000000000000090 0x00000012 main

(the breakpoint in main is the function prologue end, hence address difference)


Address              Line   Column File   ISA Flags
------------------ ------ ------ ------ --- -------------
0x00000000000050c0      4      0      1   0  is_stmt
0x00000000000050c4      5      0      1   0  is_stmt prologue_end
..
0x00000000000050f4     15      0      1   0  is_stmt
0x000000000000510c     16      0      1   0  is_stmt prologue_end
0x000000000000511c     17      0      1   0  is_stmt
..
0x0000000000005184     21      0      1   0  is_stmt end_sequence
```

Step 1: Binary and debug information parsing

- ‘Tests’ pass
- Enough information to progress
  - Other hardware may have additional sections you may want to give LLDB knowledge about
  - Can add as when required

Step 1

Binary and debug information parsing

Done!

Step 2

Control and state of target system

Step 2: Control and State of target system

- Two routes for this, local and remote

- Local Debugging
  - OS X, Linux and FreeBSD support for this in trunk
  - This is the ‘normal’ debugger architecture
  - We don’t want to run the full debugger on the DSP, or other embedded style systems.
  - Will not be looking at local debugging in this talk

- Remote Debugging
  - Usually over TCP, serial, TCP over USB
  - For Hexagon, remote is ideal
  - LLDB has built-in support for GDBs Remote Serial Protocol (RSP)
  - gdb/gdbserver style, for those familiar with that

- Remote Serial Protocol – Crash Course
    - Simply client/server ASCII communication
    - Packet-based

- Remote Serial Protocol – Crash Course
- Set breakpoint, hit breakpoint, read memory, continue and stop with a segfault

```
CLIENT SENDS -> $Hc0                        //Set Thread
CLIENT SENDS -> $Z1,<address>,4             //Set a hardware breakpoint
SERVER SENDS -> $OK
CLIENT SENDS -> $c                          //Continue execution
SERVER SENDS -> $T<thread status>           //Thread status report
SERVER SENDS -> $S05                        //Signal: Trap
CLIENT SENDS -> $m<address>,<size>          //Read memory
SERVER SENDS -> $<data>
CLIENT SENDS -> $g                          //Get register values
SERVER SENDS -> $<data>
CLIENT SENDS -> $c                          //Continue execution
SERVER SENDS -> $T<thread status>           //Thread status report
SERVER SENDS -> $S0b                        //Signal: Segfault
```

- Remote Debugging
  - Stub runs on the target, communicates to LLDB via RSP over whichever medium is available
  - Read/Write memory
  - Read/Write registers
  - Thread states
  - Breakpoint setting/unset

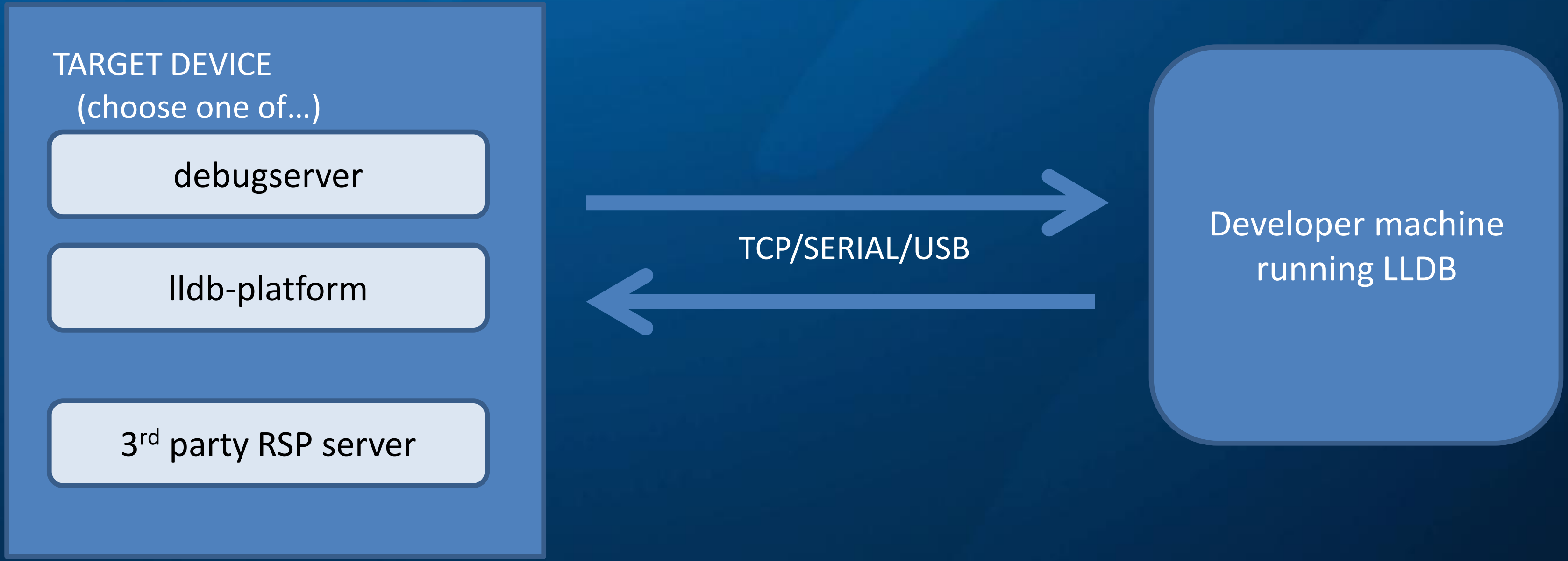
- Remote Debugging
  - Important point – the stub can be dumb, and should be for embedded
    - Why do something that isn't needed?
  - It doesn't have debug info for the running program
  - It is simply target control and state inspection

- Remote Debugging options with LLDB
  - LLDB has gdb-remote support
  - There are three server options for the target

1. Debugserver
2. lldb-platform
3. 3rd party RSP server

Step 2: Control and State of target system

- # Remote Debugging options with LLDB



- Debugserver
  - LLDB feature page states OS X debugging only
  - A manual process, run debugserver with debugee executable as argument
  - Could be ported
    - Not ideal
    - However, focus not on debugserver any more...

- lldb-platform
  - Designed as a daemon, services remote actions
  - Should be able to list processes, attach, transfer files, start debugging sessions
  - Development gaining momentum
    - But it is still very early in development, needs work
  - If you were to port anything, port this
    - You will be happy you did in the longer term

Step 2: Control and State of target system

- lldb-platform
  - And if you were to port...
  - It’s just one source file lldb-platform.cpp in tools/lldb-platform
  - Uses GDBRemoteCommunicationServer.cpp
  - Would need to implement a Host interface
    - See lldbHostCommon
    - Host.cpp

- 3rd party RSP server
  - Your architecture may already have a remote debug server integrated
  - This is the case for the Hexagon simulator
  - Want to leverage this as much as possible
  - Need to watch out for divergence from the ‘standard’ protocol
    - Extensions easy to seep into system
    - Will need to ensure LLDBs GDBRemote system is updated

- LLDBs RSP support
  - Has been extended adding features
  - Traditionally version mismatches between gdbserver/gdb has been very nasty
  - New extensions trying to aid this
  - Extensions are documented pretty well
    - http://llvm.org/svn/llvm-project/lldb/trunk/docs/lldb-gdb-remote.txt

- LLDBs RSP support
  - You need to define the target register set if packet extensions not supported
    - Previously this has been hard coded
  - Can be done via a python script

```
(lldb) settings set plugin.process.gdb-remote.target-definition-file
                                                         hexagon_target_definition.py
(lldb) gdb-remote test:1234
```

- Example script available (x86_64_target_definition.py)

- LLDBs RSP support
  - Would be nice to have a plugin system to extend supported packets
    - From a client perspective, as to aid with 3rd party servers
  - At the moment you need to add to a large switch statement
  - Ideally, have a default fallthrough path to a series of handler plugins

- LLDBs RSP support – packet extensions
  - We will probably look into this with Hexagon
  - Will aid debugger developers, especially if some RSP packets are optional/internal only
    - Easily able to separate handlers out for upstream/internalonly
  - Will keep the 'base' RSP implementation in LLDB clean

- Where are we?
- Step 2
  - Control and state of the target system
- We can see the status of the target, read/write memory
  - Is that enough?

- It could be!
  - But we want a useful debugger.
- Can we pull files from the target?
  - This is incredibly useful
- Also need to tie up the remote-debugging aspects to the architectures we support too
- We add this with the Platform plugin

- Platform Plugin
  - Methods for performing actions on the platform we’re adding
  - What architectures are supported?
  - How to launch and attach to processes – if supported.
  - Downloading and uploading files
  - See PlatformRemoteGDBServer.cpp for example

- Process Plugin
  - Directs the various parts we need to do in debugging a process to the GDBRemote system
  - Resuming processes, writing memory, etc
  - Does the waiting for responses from the remote server
  - ProcessGDBRemote.cpp should be enough already for general debugging requirements

Step 2: Control and State of target system

- Remote debugging
    - Control the target system
    - Query its state

Step 2

Control and State of target system

Done!

- At this point
  - Breakpoints
  - Can view memory and registers
  - Source debugging!
  - Other features that could work:
    - Step over single step
    - Some variables can be viewed

- Uhh, I'm trying to debug within a dynamic library
  - **Lots** of things left to implement to make a *good* debugger, let alone great!

Step 3

Interpretation of debuginfo and target state

(more often known as the hard part)

- Dynamic Loaders/Linkers
  - The debugger needs to track shared libraries
  - Whatever OS/dyld you use, should have an API debuggers use to inspect state
  - LLDB uses an additional RSP packet in this case:
    - qShlibInfoAddr
    - Traverses known structures to work out shared libraries
    - Then can pull files, parse for debug info and debug
    - Uses the work from step one

- Dynamic Loaders/Linkers
  - If your target does not have shared libraries and is completely static, you will probably not need this at all!
- Can look at DynamicLoaderPOSIXDYLD.cpp
  - Uses Process->GetImageInfoAddress()
  - With a GDBRemote process, this sends the RSP packet to request this information

- Dynamic Loaders/Linkers
  - Hexagon supports dynamic linking
  - Will be adding this support later
    - Based on System V ABI data structures

Step 3: Interpretation of debuginfo and state

- ABI
  - Argument passing
  - Function returns
  - Register status
    - Volatile, etc
- Without a correct ABI plugin, the debugging experience won't be great

- ABI
  - Have a look at ABIMacOSX_arm.cpp
  - Can use that as a base
    - Certainly for ARM targets!
    - Have tried using it on an arm target running Linux with minor changes, more than enough to start with
- Implementing our own ABIHexagon classes
  - At a very early stage currently

codeplay

- Call Stacks
    - If your debug information is of high quality, and includes call frame information (CFI), great
    - If the ABI always has a frame pointer, great
    - Without the CFI to generate frame addresses of previous frames, arguments/registers may be incorrect
    - Unwinding…

- The Unwinder
  - Stack Unwinding occurs via a Plan list
  - Plans used throughout LLDB
  - General idea
    - Finds frame pointer if it's always defined
    - Utilize the CFI in the debugging information
    - If all else fails, it will try to generate CFI by emulation, if an emulator is available
  - The emulator isn't just for unwinding

- InstructionEmulator
  - Emulation is required within the debugger to…
    - Generate CFI debug information if it does not exist
      - Look where registers are saved, etc
    - Calculate branch target addresses for single steps
      - Hexagon has hardware single step support, so this of less important in this case
  - Does not need to be a full emulator
    - Only the instructions which are used for the above actions

- InstructionEmulator
  - Whilst it does not need to be a full emulator
    - Still should be able to emulate to the point that if required, debugging optimized code is possible

Step 3: Interpretation of debuginfo and state

- Unwinding, InstructionEmulation…
- Could fill a whole other talk
- Main point: interpretation of debug information in tandem with runtime state is where the advanced features of the debugger lie
- Developers now *expect* these features
- Need to devote lots of time to these areas

Step 3

Interpretation of debuginfo and target state

Not even close to being done

Conclusion

- In summary
- Three steps
  - Get the binary loading
  - Adapt/port whichever remote server you choose, making sure to add your platform methods
  - The real meat – DynamicLoader, Unwinder, Emulator
- The last 10% takes 90% of the time…

- In summary
  - LLDB fantastic, had good support for the most popular object format and remote debugging
  - Remote debugging needs work with new packets and extendibility – an RSP packet plugin system would be great
  - The advanced features developers crave mean implementing very complex systems to interpret the debuginfo with runtime state
  - Not even mentioned IDE integration yet... (another talk?)

- In summary
  - Steps 1 & 2: Getting a bare bones debugger up and running is fairly straightforward and can progress quite quickly, weeks to months of work
  - Step 3: Getting a good debugger up and running is another matter!

Thank you!

I’m on twitter @domipheus

Codeplay is on twitter @codeplaysoft

Many thanks to Qualcomm Innovation Center, Inc for

allowing use of Hexagon as an example

Q & A

colin@codeplay.com

I'm on twitter @domipheus

Codeplay is on twitter @codeplaysoft

Q & A

colin@codeplay.com

I'm on twitter @domipheus

Codeplay is on twitter @codeplaysoft