Clinging To Clang

KHEM RAJ
Comcast
Agenda

❖ Introduction to Clang
❖ Project Goals
❖ Clang based Cross toolchain
❖ Compiling Embedded Linux applications, kernel with Clang
❖ Using Clang in Yocto for system compile
❖ Using Yocto to generate Clang based cross compiler SDK
❖ Additional Clang tools
❖ Using Clang runtime in Embedded Linux Applications
Introduction To Clang

❖ Native compiler FrontEnd to LLVM Infrastructure
❖ Supports C/C++ and Objective-C
❖ The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. - llvm.org
❖ First release in 2003
❖ Latest Release 3.9.0 (Sep 2016)
❖ Pronounced as /klaNG/
Clang Goals

- GCC compatibility
- All extensions are recognized and marked as extension diagnostics
- IDE integration
- Uses LLVM BSD license
  - Considering change to Apache-2
- Language conformance, ISO C, C++
Clang Goals

- Newer codebase designed using C++, supports API based architecture
- Focuses on making it light and fast
- User friendly diagnostics (http://clang.llvm.org/diagnostics.html)
- Offers fix-it hints, highlights

```bash
kraj@haswell ~ % aarch64-poky-linux-musl-clang --sysroot=/opt/poky/2.0+snapshot/sysroots/aarch64-poky-linux-musl -Ofast test.c -c
```
```
test.c:9:21: warning: implicit declaration of function 'canonicalize_file_name' is invalid in C99 [-Wimplicit-function-declaration]
  resolved_path = canonicalize_file_name(path);
  ^
test.c:9:19: warning: incompatible integer to pointer conversion assigning to 'char *' from 'int' [-Wint-conversion]
  resolved_path = canonicalize_file_name(path);
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
```
Speedy Compile with Clang

❖ Compile time
❖ Core goal of project fast compilation and low memory usage
❖ Webkit clang (2297.93 seconds) gcc (2838.10 seconds)
❖ Linking Time
❖ Use split dwarf (-gsplit-dwarf)
❖ Can reduce the link time by 3x
Who is using Clang

- Debian experimental
  - Optional compiler ~90% packages can compile
- LLVMLinux
  - Compile Linux Kernel with Clang
- The ELLCC Embedded Compiler Collection
- FreeBSD
- OpenMandriva
- OpenEmbedded/Yocto Project
- ....
Compiler for Embedded Linux - Current Norm

- Embedded Linux is primarily cross-compiled
- GCC is primary system compiler
  - Supports many Architectures/machines
  - Full list of GCC backends [https://gcc.gnu.org/backends.html](https://gcc.gnu.org/backends.html)
    - arc, arm, aarch64, mips, mips64, powerpc, powerpc64, x86, x86_64, tile, nios2, microblaze, and many more …..
GCC based toolchains

- GNU toolchain basic ingredients
  - Binutils
    - Provides linker, assembler and post processing tools
  - C/C++/Java/ADA/Fortran/golang gcc compiler
    - Cross Compilers
      - C/C++ runtime (libgcc, libstdc++, libfortan …)
  - Standard System C Library
    - glibc/uclibc/musl
- Debugger
  - gdb
Toolchain Build Sequence - GCC

Prerequisites

- cross gdb

- cross binutils

- bootstrap cross gcc

- linux kernel headers

  - libc headers/
  - startupfiles/dummy
  - libc.so

- gcc runtime

- Full libc

- full cross gcc stage 2
Prerequisites

- cross binutils
- bootstrap cross gcc
- linux kernel headers

Full libc

- gcc runtime
- full cross gcc stage 2

Libc headers/
startupfiles/dummy libc.so
Toolchain Build Sequence - Clang

**Prerequisites**
- cross gdb

**cross binutils**

**Clang with Universal driver**

**linux kernel headers**
- libc headers/
- startupfiles/dummy libc.so

**compiler runtime**

**Full libc**
Cross-Compile Embedded Linux Apps

- install clang on your host distribution (Debian, Arch ..)
- Download prebuilt toolchain from Yocto Project
  - http://autobuilder.yoctoproject.org/pub/nightly/CURRENT/toolchain/x86_64/
- Linaro toolchain releases for arm
- Install and add the cross toolchain to PATH
  ```bash
  /usr/bin/clang --target=aarch64 -ccc-gcc-name aarch64-poky-linux-gcc hello.cpp --sysroot=/opt/poky/2.0+snapshot/sysroots/aarch64-poky-linux
  ```
Cross-Compile Embedded Linux Apps

- This would _only_ compile the given application with clang
- Rest of system is still precompiled
- GNU binutils will be used for linking and assembling
- Same setup can be leveraged for building Linux kernel
- Export the CROSS_COMPILE, CC variables and its ilk correctly.
Cross-Compile Embedded Linux Platform

- Clang can not _yet_ build every bit of Embedded Linux Platform
- Linux Kernel effort
- [http://llvm.linuxfoundation.org/index.php/Main_Page](http://llvm.linuxfoundation.org/index.php/Main_Page)
- System C library e.g. glibc does not compile with clang
  - [https://sourceware.org/glibc/wiki/GlibcMeetsClang](https://sourceware.org/glibc/wiki/GlibcMeetsClang)
Cross-Compile Embedded Linux Platform

- Hybrid approach is needed (both GCC and Clang)
- Chromium OS
  - has overlays for clang
- OpenEmbedded
  - provides a layer meta-clang
Cross-Compile Embedded Linux Platform

❖ OpenEmbedded approach
❖ Managed in a layer of its own
   ❖ meta-clang layer(https://github.com/kraj/meta-clang)
❖ meta-clang - when added switches default system compiler to clang
❖ Defines TOOLCHAIN variable (one of “gcc”, “clang”)
❖ “gcc” - Enable gcc as default compiler for the package
❖ “clang” - Enable clang as default compiler for package
Cross-Compile Embedded Linux using Yocto

```bash
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git clone git://github.com/kraj/meta-clang
$ ./oe-init-build-env
$ bitbake-layers add-layer ../meta-clang
```
Cross-Compile Embedded Linux using Yocto

- Non-clangable recipes
  - Use specific gcc extensions not implemented in clang
    - Nested functions
  - Has been fixed but patches not accepted upstream
  - Has been fixed but not updated in OE yet
  - Has valid diagnostics
  - Laziness..
Cross-SDK for Embedded Linux using Yocto

- build images
  - core-image-sato - X based Graphical image
  - core-image-minimal - Small console image
- Generates SDK for application development
  - bitbake -cpopulate_sdk core-image-minimal
    - Self installing SDK is
      - tmp/deploy/sdk/poky-glibc-x86_64-core-image-minimal-aarch64-toolchain-2.0+snapshot.sh
- Installing SDK
  - ./tmp/deploy/sdk/poky-glibc-x86_64-core-image-minimal-aarch64-toolchain-2.0+snapshot.sh
Using SDK

- Setup Environment
- 
  `. /opt/poky/2.0+snapshot/environment-setup-aarch64-poky-linux`
- SDK contains both clang and gcc cross compilers
  - `CC,CXX,CPP` variables for gcc based cross compilers
  - `CLANGCC, CLANGCXX,CLANGCPP` for clang based c/c++ compiler
Using SDK - Applications

❖ Building GNU hello world

```
$ tar xf hello-2.10.tar.gz
$ cd hello-2.10
$ . /opt/poky/2.0+snapshot/environment-setup-aarch64-poky-linux-musl
$ CC=${CLANGCC} ./configure --host=aarch64-poky-linux
$ make V=1
$ make install DESTDIR=/tmp/hello
$ scp /tmp/hello/usr/local/bin/hello <target>
```
Using SDK - Kernel

- Building llvmlinux kernel
  
  ```
  $ git clone git://git.linuxfoundation.org/llvmlinux/kernel.git llvmlinux
  
  $ cd llvmlinux
  
  $ make ARCH=arm64 CC=${CLANGCC} LDFLAGS="" defconfig
  
  $ make ARCH=arm64 CC=${CLANGCC} LDFLAGS="" -j8 vmlinux
  ```

- It ends in compiler errors :(

- What have you been waiting for - Fix it!!
Clang - More tools

- Clang Static Analyzer http://clang-analyzer.llvm.org/
- Static analysis of musl (C library)
- Configure
  ```
  /a/builder/home/kraj/work/oe/musl/configure --enable-debug --target=arm CC=/a/build/tmp/sysroots/x86_64-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-clang CFLAGS="--sysroot=/a/build/tmp/sysroots/raspberrypi2" LDFLAGS="-lgcc_s"
  ```
- Compile
  ```
  scan-build --use-analyzer /a/build/tmp/sysroots/x86_64-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-clang --use-cc /a/build/tmp/sysroots/x86_64-linux/usr/bin/arm-poky-linux-gnueabi/arm-poky-linux-gnueabi-clang make -j
  ```
- Results e.g. https://busybox.net/~kraj/scan-build-2016-03-02-225259-30448-1/
musl scan-build runs found some issues which resulted in improvements

Clang - More tools

- Clang-check - A syntax checker
  - Selective runs with diagnostics for subset of files
  - Helps integrate with IDEs
  - Use it in fix-it mode
- clang-format
  - Reformat C++ source files
  - Useful for IDE integration
  - Commit policy
- clang-tidy
  - Lint tool
Using Clang Compiler Runtime - libc++

- libc++ is C++ runtime implementation
- STL - libc++
- ABI - libc++abi
- EH support
- libunwind
- llvm-libunwind
- Control with -stdlib option

```
kraj@haswell ~ % clang++ -std=c++11 -stdlib=libc++ -lc++abi ~/hello.cpp
kraj@haswell ~ % ./a.out
1: Hello dude!
2: Hello dude!
3: Hello dude!
kraj@haswell ~ % readelf -d ./a.out
```

Dynamic section at offset 0x1c18 contains 28 entries:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Name/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc++abi.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc++.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libm.so.6]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libgcc_s.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc.so.6]</td>
<td></td>
</tr>
</tbody>
</table>
Using Clang Compiler Runtime - compiler-rt

❖ Compiler-RT provides
  ❖ compiler built-ins
    ❖ Full support for libgcc interfaces
❖ Sanitizer runtimes
  ❖ Support libraries sanitizer instrumented code
❖ Profile
  ❖ Coverage collection
Using Clang Compiler Runtime - sanitizers

- AddressSanitizer -fsanitize=address
  - memory error detection e.g. out of bound accesses
    - Compiler instrumentation and runtime code
- ThreadSanitizer (64bit arches only) -fsanitize=thread
  - Detect Data Races
- MemorySanitizer -fsanitize=memory
  - Detects uninitialized reads
- LeakSanitizer -fsanitize=address ( only x86_64 )
  - Run-time memory leak detector ( WIP x86_64)
- DataFlowSanitizer - Provides Data flow analysis
Using Clang Compiler Runtime - libunwind

- implements system unwinder
  - High level APIs
    - implement _Unwind_* functions needed by libcxxabi
  - low level APIs
    - unw_* functions
    - HP libunwind compatible APIs
Clang Runtime in Action

❖ Use libunwind & libc++ runtimes

❖ before

kraj01@eos ~ % aarch64-poky-linux-clang++ --sysroot=/opt/poky/2.6+snapshot/sysroots/aarch64-poky-linux hello.cpp
kraj01@eos ~ % aarch64-poky-linux-readelf -d ./a.out

Dynamic section at offset 0xdd8 contains 27 entries:
<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Name/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libstdc++.so.6]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libm.so.6]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libgcc_s.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc.so.6]</td>
<td></td>
</tr>
</tbody>
</table>

❖ After

kraj01@eos ~ % aarch64-poky-linux-clang++ --sysroot=/opt/poky/2.6+snapshot/sysroots/aarch64-poky-linux -stdlib=libc++ -nodefaultlibs -lc++ -lc++abi -lc -llunwind hello.cpp
kraj01@eos ~ % aarch64-poky-linux-readelf -d ./a.out

Dynamic section at offset 0x1dd8 contains 27 entries:
<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Name/Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc++.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libcppabi.so.1]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libc.so.6]</td>
<td></td>
</tr>
<tr>
<td>0x0000000000000001 (NEEDED)</td>
<td>Shared library: [libunwind.so.1]</td>
<td></td>
</tr>
</tbody>
</table>
Limitations

- Not all packages can be compiled with clang yet
- See https://github.com/kraj/meta-clang/blob/master/conf/nonclangable.conf
- Integrate cross SDKs into IDEs e.g. eclipse, develop etc.
- Upstream kernel doesn’t yet compile
Thank you