LLVM ARM Toolchain

LPC 2014, Düsseldorf, Germany

Renato Golin
LLVM Tech-Lead
Linaro

Agenda

- Building a toolchain from the ground up
  - Correctness, performance, ABI compatibility
  - Tools, libraries, system integration
- Keeping the toolchain stable
  - Validation and continuous integration
  - Release tests and benchmarking
- Push forward
  - Increase compatibility with other compilers, systems
  - Improve performance, target specific behaviour

Building a toolchain from the ground up

- What constitutes a toolchain?
  - Compiler: front-ends, optimisations, back-ends
  - Tools: assembler, linker, object dumps
  - Compiler libraries: libgcc, compiler-rt
  - Libraries: C library, STL, Boost, etc.
  - Standard headers, including target specific (arm_neon.h)
  - System behaviour: compiler driver
- How to validate toolchains?
  - Conformance and performance testing (front/back-end)
  - System integration (driver)

Short history of ARM LLVM

- First, make sure the code generated is correct
  - 2010: Connected EDG front-end to LLVM back-end
- Next, make sure the ABI is followed and code is sane
  - 2011/2012: Extensive ABI tests, performance improvements
- Validation and CI
  - 2013: Basic buildbots (check, self-host, test-suite)
- Integrated assembler & exception handling
  - 2013/2014: extensive support, now on by default
- Libraries
  - 2014: Compiler-RT + libc++ (STL) testing

Current Work

- Compiler Library
  - LLVM used to rely on *libgcc* for ARM
  - But a compiler library has to work on its own
  - Compiler-RT building on ARM and AArch64
  - But still using *libgcc_eh* (instead of *libunwind*)
- C library
  - Using glibc, and that's good enough
- STL Library
  - Libc++ building well on ARM/AArch64, but needs more testing

Current Work

- Linker
  - Bfd and gold work well with LLVM, but would be good to have a linker with compatible license
  - LLd is promising, but still too green
  - MCLinker is more mature, but too specific

Keeping the toolchain stable

- Validation
  - Release testing (self-hosting, test-suite)
  - Release benchmarking (SPEC, EEMBC)
  - Minor release validation, too (3.4.x)
- Continuous integration
  - Buildbots on various stages
    - Build+check-all
    - Self-host+check-all
    - Test-suite (+benchmark)
    - Compiler-RT tests (including sanitizers)

Keeping the toolchain stable

- Further continuous integration
  - Adding more stages of compatibility
    - Libc++ / libc++abi buildbot
    - Run test-suite with RT+libc++
    - Build and use lld on standard bots
    - Bootstrap lldb buildbots
  - System integration
    - Build on different platforms (Debian, Arch, Fedora)
    - Chromium/Firefox build & tests

Pushing forward

- Linker
  - Probably lld (already getting a lot or ARM/AArch64 logic)
  - Maybe MCLinker, too (make it more target agnostic)
  - LTO support everywhere!
- Multiarch / IFUNC
  - Assembler behaviour (.fpu/.arch)
  - Driver environment discovery (header/lib paths)
- Inline assembly
  - GNU magic register definitions (“Q” vs. “Qo”)
  - GNU changing clobber definitions (memory → sp)

Pushing forward

- Sanitizers
  - Make sure all memory sanitizers (msan, lsan, asan) work as intended on ARM architectures (ie. add RT support)
  - Undefined behaviour sanitizer needs investigation
  - Thread sanitizers need 64-architecture (pointer magic)
- Improve integrated assembler support
  - Build large projects (Chromium, Firefox)
  - Build the kernel!
- Stress libc++'s compatibility with EHABI

Far future...

- MCJIT
  - Usage in CPU can be driven by:
    - GPGPU languages, as development / debug platforms, fall-back, load balancing
    - Debugger, as failure-safe execution
    - On-demand computing: scripting (JS, flash, etc)
- VMKit
  - Can we use virtualisation extensions?
- Thread-sanitizer
  - Can we port the thread sanitizer to 32-bit platforms?

Bottom Line

- Creating toolchains is hard work
- The work that needs doing is either boring or annoying
- The amount of politics needed is beyond sanity levels
- **But it has to be done!**

The End

Questions!