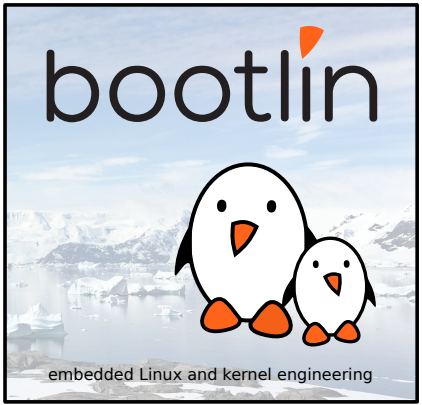
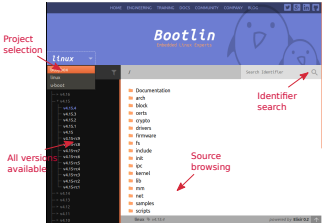
LLVM tools for the Linux kernel

Michael Opdenacker
michael.opdenacker@bootlin.com



Michael Opdenacker

- Founder and Embedded Linux engineer at Bootlin:
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
- Free Software contributor:
  - Current maintainer of the Elixir Cross Referencer, making it easier to study the sources of big C projects like the Linux kernel. See https://elixir.bootlin.com
  - Current documentation maintainer for the Yocto Project
  - Co-author of Bootlin's freely available embedded Linux and kernel training materials (https://bootlin.com/docs/)



Disclaimer

- I'm neither a Clang/LLVM expert, nor involved in the project to build the Linux kernel with Clang.
- I'm just interested in the topic, and sharing my own findings with you.
- This is also why this is a short presentation!

Compiling the Linux kernel with Clang

Definitions and conventions in this document

- Clang: compiler front-end for C, C++, Objective-C, OpenCL, CUDA...
- `clang`: the command provided by the Clang project
- LLVM: compiler back-end, and name of the project Clang is part of.
- GCC: GNU Compiler Collection.
- Gcc or `gcc`: the C compiler in GCC

Why compiling the Linux kernel with Clang?

- Google reason: have only one toolchain and build all of Android with Clang.
- Good to support two different compilers, to shake out code that relies on undefined behavior in the compiler.
- Get different warnings from Clang.
- Access further optimizations such as *Link Time Optimization (LTO)*.
- Interest in LLVM static analysis tools.
- Linux is a big and complex project: it can also allow to find Clang bugs.

Differences between clang and gcc

- `gcc` needs to be compiled for each architecture you want to support. Therefore, many different `gcc` cross-compilers are available.
- `clang` supports all target architectures at the same time. The same executable can generate code for all the architectures it supports.

Linux kernel: Clang support status

Architecture	Level of support	`make` command
arm	Supported	`LLVM=1`
arm64	Supported	`LLVM=1`
mips	Maintained	`CC=clang`
powerpc	Maintained	`CC=clang`
riscv	Maintained	`CC=clang`
s390	Maintained	`CC=clang`
x86	Supported	`LLVM=1`

Source: https://www.kernel.org/doc/html/latest/kbuild/llvm.html

Environment setup for Clang (Ubuntu 22.04)

Common packages:

```
sudo apt install build-essential flex bison libssl-dev
```

Clang packages:

```
sudo apt install clang llvm lld
```

Environment for compiling the kernel:

```
export ARCH=arm
export LLVM=1
```

Environment setup for GCC (Ubuntu 22.04)

Gcc packages:

```
sudo apt install gcc-12-arm-linux-gnueabihf
sudo update-alternatives --install /usr/bin/arm-linux-gnueabihf-gcc \
     arm-linux-gnueabihf-gcc /usr/bin/arm-linux-gnueabihf-gcc-12 12
```

Environment for compiling the kernel:

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabihf-
unset LLVM
```

Compile time and size results

With Clang 14

```
cd linux-5.18-rc6
make omap2plus_defconfig
time make -j8 zImage

real    25m59,392s
user    84m4,292s
sys     13m27,629s

du -s arch/arm/boot/zImage
4912    arch/arm/boot/zImage
```

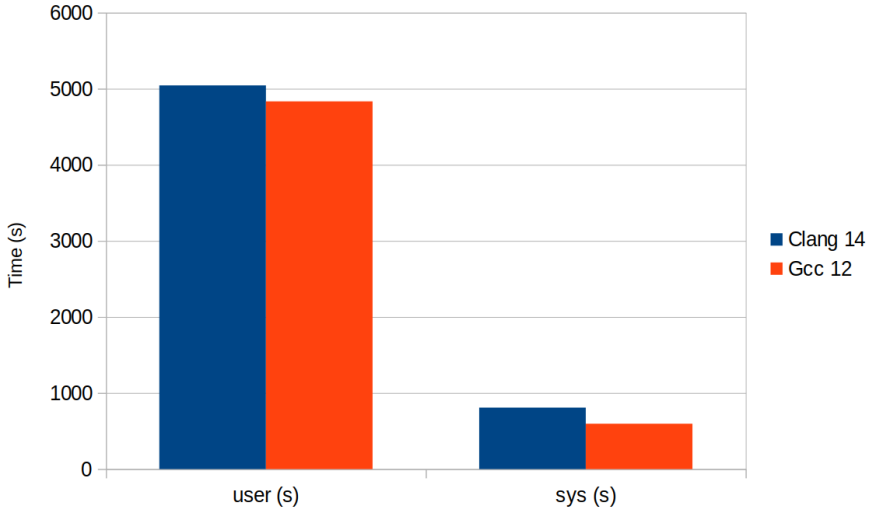
With GCC 12

```
cd linux-5.18-rc6
make omap2plus_defconfig
time make -j8 zImage

real    24m11,143s
user    80m34,624s
sys     9m55,833s

du -s arch/arm/boot/zImage
4908    arch/arm/boot/zImage
```

Compile time compared



Boot time comparisons

On BeagleBone Black, booting Linux 5.18-rc6 on a small initramfs with Busybox, time between `U-Boot SPL` and `Please press Enter`:

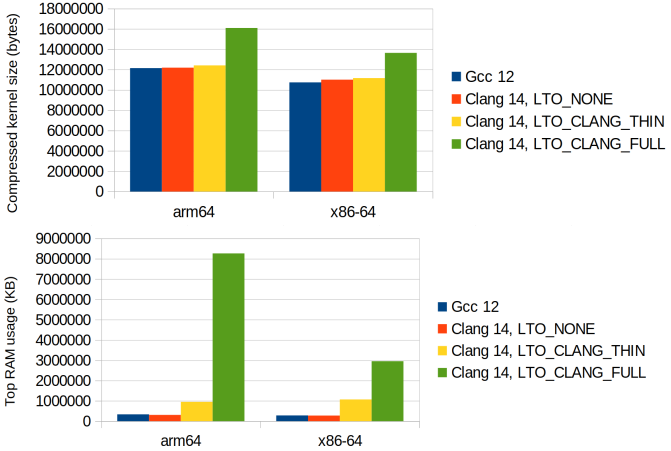
- Kernel built with Clang 14:
  Average boot time: 6.427 s
- Kernel built with gcc 12:
  Average boot time: 6.422 s (-5 ms)

Conclusion: the boot time difference is negligible.

Link Time Optimization (LTO)

- Clang allows to implement global optimizations at link time
  See https://www.llvm.org/docs/LinkTimeOptimization.html for details.
- LTO is supposed to be able to identify and delete dead code.
- Three LTO options in the Linux kernel:
  - CONFIG_LTO_NONE: no LTO (by default)
  - CONFIG_LTO_CLANG_FULL: full LTO but heavy CPU and RAM usage at link time.
    Example: needs 7.9 GB of RAM to link an arm64 kernel (defconfig configuration).
  - CONFIG_LTO_CLANG_THIN: much lighter than full LTO on RAM usage and CPU time. See https://clang.llvm.org/docs/ThinLTO.html.
- Gcc also has LTO support but is not supported for building the Linux kernel.

LTO tests



- `arm64` build (`Image.gz`): Linux 5.18-rc7, `defconfig` configuration
- `x86` build (`bzImage`): Linux 5.18-rc7, `x86_64_defconfig` configuration

Notes on LTO results

- Note that LTO (full and thin) is currently not enabled on `arm` (32 bit).
- The full LTO kernels are way bigger than non LTO ones
- This is most probably due to extra inlining, good for performance, but not for boot time (a bigger kernel takes more time to load and decompress). Topic discussed on https://github.com/ClangBuiltLinux/linux/issues/1643.
- Lacked time to run performance benchmarks on `x86` or `arm64`

clang-tidy

"clang-tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis."

- Set up your environment for compiling your kernel with Clang
- Configure your kernel
- You could even run `make allyesconfig` to cover the whole code
- Run `make clang-tidy` or better `make -j8 clang-tidy`
- Output: no issue reported on Linux 5.18-rc6 (`omap2plus_defconfig`)

https://clang.llvm.org/extra/clang-tidy/

clang-analyzer

"The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs."

- Also set up your environment for compiling your kernel with Clang and configure your kernel (possibly with `make allyesconfig`).
- Run `make clang-analyzer` or better `make -j8 clang-analyzer`
- As the volume of output is huge, suggestion to duplicate it to a file:
  `make -j8 clang-analyzer 2>&1 | tee /tmp/clang-analyzer.log`
- Note: static analysis takes much more time than compiling.

https://clang-analyzer.llvm.org/

clang-analyzer: many false positives

```
...
/home/tux/linux/clang/linux-5.18-rc6/kernel/reboot.c:831:9: note: Call to function 'sprintf' is insecure as it
      does not provide security checks introduced in the C11 standard. Replace with analogous functions that
      support length arguments or provides boundary checks such as 'sprintf_s' in case of C11
        return sprintf(buf, "%d\n", reboot_cpu);
               ^~~~~~
Suppressed 40 warnings (40 in non-user code).
Use -header-filter=.* to display errors from all non-system headers. Use -system-headers to display errors
      from system headers as well.
15 warnings generated.
/home/tux/linux/clang/linux-5.18-rc6/kernel/async.c:125:2: warning: Value stored to 'calltime' is never read [
      clang-analyzer-deadcode.DeadStores]
        calltime = ktime_get();
        ^          ~~~~~~~~~~~
/home/tux/linux/clang/linux-5.18-rc6/kernel/async.c:125:2: note: Value stored to 'calltime' is never read
        calltime = ktime_get();
        ^          ~~~~~~~~~~~
...
```

However:

- The `sprintf_s()` function doesn't exist in the kernel code
- `calltime` in `kernel/async.c` can be accessed if the configuration enables `pr_debug()`.
- There are countless examples like this

Improvements to the kernel code

However, the Clang warnings have allowed to implement many improvements to the Linux kernel code:

- Nathan Chancellor's patches: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=author&q=chancellor
- Nick Desaulniers' patches: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=author&q=desaulniers

Further limitations and issues

- Difficult to run `clang-analyzer` on a single file
  (see workaround on the next page)
- How to tweak `clang-analyzer` to keep only the warnings relevant to Linux kernel code?

How to run clang-analyzer on a single file

- Make a copy of the `compile_commands.json` generated file, which just describes one file. Let's call it `compile_commands-1file.json`
- Then run:
  ```
  python3 ./scripts/clang-tools/run-clang-tools.py clang-analyzer compile_commands-1file.json
  ```

Conclusions

- For the most popular CPU architectures, building the kernel with Clang instead of Gcc is already possible and mature.
- With Clang, you don't need a cross-compiler any more!
- The Clang warnings have already helped to improve the kernel code
- However, we haven't reaped all the benefits of using Clang yet:
  - No size benefits of LTO yet: LTO kernels much bigger
  - Many `clang-analyzer` warnings still irrelevant

Useful resources

- ClangBuiltLinux project: https://clangbuiltlinux.github.io/
  Build status, bug reports, documentation, meetings.
  The place to join to get involved.
- Kernel documentation: *Building Linux with Clang/LLVM*
  https://www.kernel.org/doc/html/latest/kbuild/llvm.html
- LWN.net: *Building the kernel with Clang*
  https://lwn.net/Articles/734071/

Thanks to Nathan Chancellor and Nick Desaulniers (Clang/LLVM Build Support kernel maintainers) for answering my questions!

Questions? Suggestions? Comments?

Michael Opdenacker
michael.opdenacker@bootlin.com

https://bootlin.com/pub/conferences/2022/lee/