Google

LTO, PGO, and AutoFDO

Linux Plumbers Conf 2020 - LLVM MC

Sami Tolvanen, Bill Wendling, Nick Desaulniers

Upstreaming

Google is shipping all of the following in downstream kernels but…

Unless these are *repeatable* processes for upstream, then they are not solutions, they are *technical debt*.

But there are implications for CI; added build durations, stale profile data, ...

What is LTO?

C code is compiled into LLVM IR. LLVM's integrated assembler is used for inline assembly.

Bitcode is compiled and optimized into native code at link time. Objtool etc. must be delayed until after linking.

Build times are tolerable with **ThinLTO**. **Full LTO** produces faster code, but is very resource-intensive.

In addition to better runtime performance, enables Clang's forward-edge **Control-Flow Integrity** (CFI) checking.

Google has shipped LTO+CFI kernels on Pixel phones since 2018.

Patches for x86 and arm64 are available in GitHub. Upstreaming in progress.

Build times with LTO

make -j**72** # 2x Xeon Gold 6154

x86_64 defconfig	**Clang w/o LTO**	**ThinLTO**	**Full LTO**
Time	51s	1min	5min 10s
Memory	240 MiB	1.7 GiB	2.8 GiB

arm64 defconfig	**Clang w/o LTO**	**ThinLTO**	**Full LTO**
Time	2min 6s	2min 14s	10min
Memory	1.6 GiB	3.5 GiB	20.3 GiB

LTO != PLO

LTO is more akin to “whole program optimization.”

It’s run just prior to link of the final executable; program semantics are fully retained.

“Post Link Optimization” (ie. Bolt, Propeller) tries to recreate program semantics from disassembly.

For PLO, what does that mean for inline asm; kernel specific sections?

Adding profiling data into the mix

- The compiler can improve code layout (function or even basic block placement) and spend time optimizing hot code only, if it has a better sense how hot/unlikely sections of code are.
- We can statically provide these via `__builtin_expect()`, and refute stale hints by providing profile data to clang-misexpect.
    - We can also collect information at runtime.
- Exporting this information from the kernel can be a challenge (PGO).
- Measuring what's important can become a double edged sword.

PGO a.k.a. FDO for Clang

Clang has two types of profiling:

IR instrumentation: **`-fprofile-generate`**
Front-end instrumentation: **`-fprofile-instr-generate`**

- Front-end instrumentation is done by the clang front-end, while IR instrumentation is done by an IR pass in LLVM. IR instrumentation is done after some limited early inlining, which makes profiling more context sensitive and instrumented binaries faster to run.

TL;DR: Use **`-fprofile-generate`**.

PGO Example Workflow

```
# Instrument the kernel
$ clang -fexperimental-new-pass-manager -fprofile-generate ...

# Profile data collection
$ echo 1 > /sys/kernel/debug/pgo/reset
 ... run load tests ...
$ cp /sys/kernel/debug/pgo/profraw vmlinux.profraw
$ llvm-profdata merge --output vmlinux.profdata vmlinux.profraw

# Using PGO
$ clang -fexperimental-new-pass-manager -fprofile-use=vmlinux.profdata ...

# Using PGO with LTO
$ ld.lld -lto-cs-profile-file=vmlinux.profdata ...
```

PGO by Numbers

Netperf: TCP STREAM	Throughput with PGO 10^6 bits/sec	Throughput without PGO 10^6 bits/sec
-m 4096 -s 128K -S 128K	15994.01	15223.10
-m 8192 -s 128K -S 128K	16058.02	15713.99
-m 32768 -s 128K -S 128K	16246.86	16189.35
-m 4096 -s 57344 -S 57344	8937.68	8781.61
-m 8192 -s 57344 -S 57344	8773.22	8666.66
-m 32768 -s 57344 -S 57344	8823.13	8743.46

AutoFDO

By using sampling rather than instrumentation, we can feed sample based profiles back into compilation.

Pros: no separate profile/release builds, better layout == improved performances.

Cons: not all architectures have hardware counters, the samples after inlining has been performed are hard to match up.

```
KBUILD_CFLAGS += -fdebug-info-for-profiling -fprofile-sample-use=
```

https://clang.llvm.org/docs/UsersManual.html#using-sampling-profilers

AutoFDO

On systems we deployed AutoFDO built kernels,
we observed up to a **12% reduction in cycles spent in the kernel**. But:

- Different micro architectures gave differing results. (None were worse, just not as good).
- Profiles from ETM on ARM weren't as profitable as using LBR on x86 (potential problem with data? Still looking).
- It's possible to use profiles from x86 when targeting ARM. Miss out on optimizing arch specifics.

Tradeoffs between AutoFDO and PGO

Separate Training vs Release builds (PGO).

Different levels of precision.

Upstream profile data

Profile data has to be processed before fed back into compiler.

Would we check in preprocessed profiles, or post processed?

Profile data needs to be periodically refreshed.

(Rough analogy to transfer learning).