Optimizing beyond the optimizer: LTO and FDO

Nicolò Valigi

June 15, 2019

HOST

PATRON

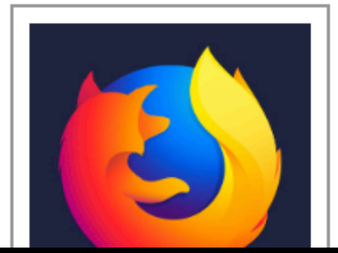
Community Crumbs

SPONSORS

Goals

Firefox Is Now Built With Clang+LTO Everywhere, Sizable Performance Wins For Linux

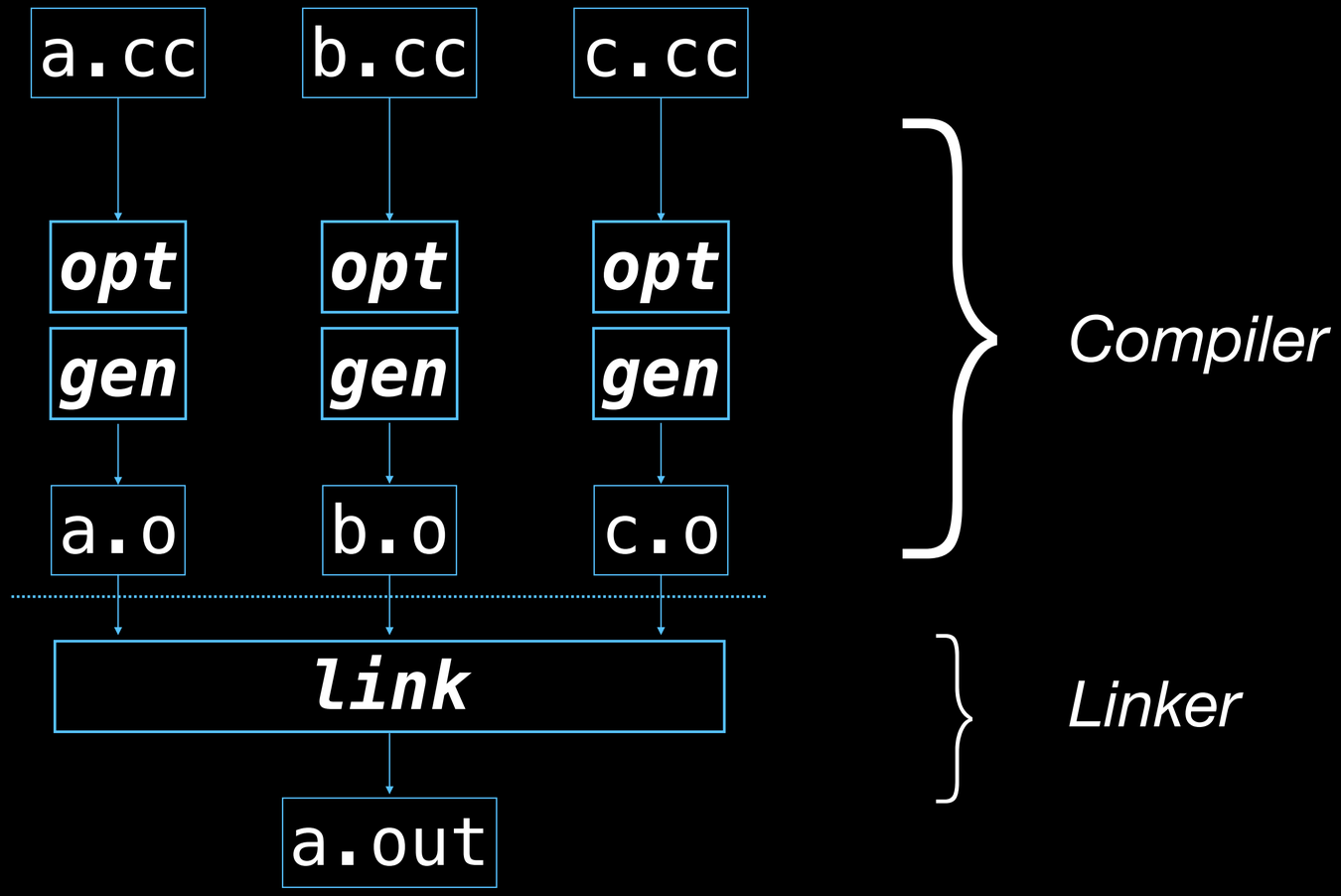
Written by Michael Larabel in Mozilla on 12 September 2018 at 05:33 AM EDT. 50 Comments



Firefox nightly builds are now built with the LLVM Clang compiler on all major platforms and the Linux build in particular is also now utilizing PGO optimizations too. Faster Firefox is coming thanks to this compiler work.

- Make programs run faster with *little* (?) effort
- Practical tips, but also learn about compilers

The C/C++ compilation model



Why separate compilation is great

+ **parallel**: each Translation Unit (roughly *.cpp* file) is compiled independently, thus using multiple CPUs. The only serial step is the linking at the end.

+ **incremental**: changes in one *.cpp* file only cause a single compilation and the link, which is much faster than recompiling from scratch.

When separate compilation breaks down (1/3)

main2.cc

```cpp
int doNothing() {
    return 0;
}

int main() {
    for (int i = 0; i < 1'000'000'000; i++) {
        doNothing();
    }
}
```

```
➜  code git:(master) ✗ g++ -std=c++17 -O2 -flto -o main2 main2.cc
➜  code git:(master) ✗ time ./main2
./main2  0.00s user 0.00s system 60% cpu 0.004 total
➜  code git:(master) ✗
```

When separate compilation breaks down (2/3)

main.cc

```cpp
#include "lib.h"

int main() {
    for (int i = 0; i < 1'000'000'000; i++) {
        doNothing();
    }
}

```

lib.h

```cpp
int doNothing();

```

lib.cc

```cpp
int doNothing() {
    return 0;
}

```

When separate compilation breaks down (3/3)

```
➜  code git:(master) ✗ g++ -std=c++17 -O2 -o main-with-lib main.cc lib.cc
➜  code git:(master) ✗ time ./main-with-lib
./main-with-lib  1.31s user 0.00s system 99% cpu 1.316 total
```

The program is actually calling the function a billion times now!

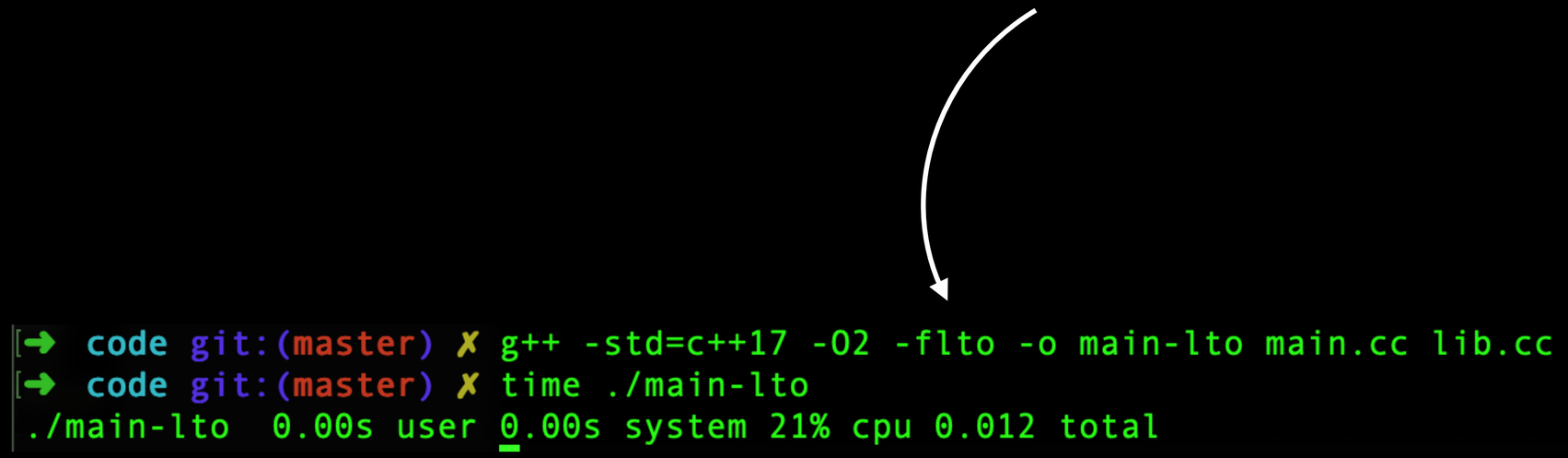
The magic of inlining

Direct benefits: inlining removes the overhead of the call/return pair, avoids setting up the stack for the callee, and passing arguments and return values.

Indirect benefits: gives the optimizer more context to work with, finding more places to apply standard optimizations.

Cons: increased compilation time. Potentially increased binary size, which is not great for CPU caches.

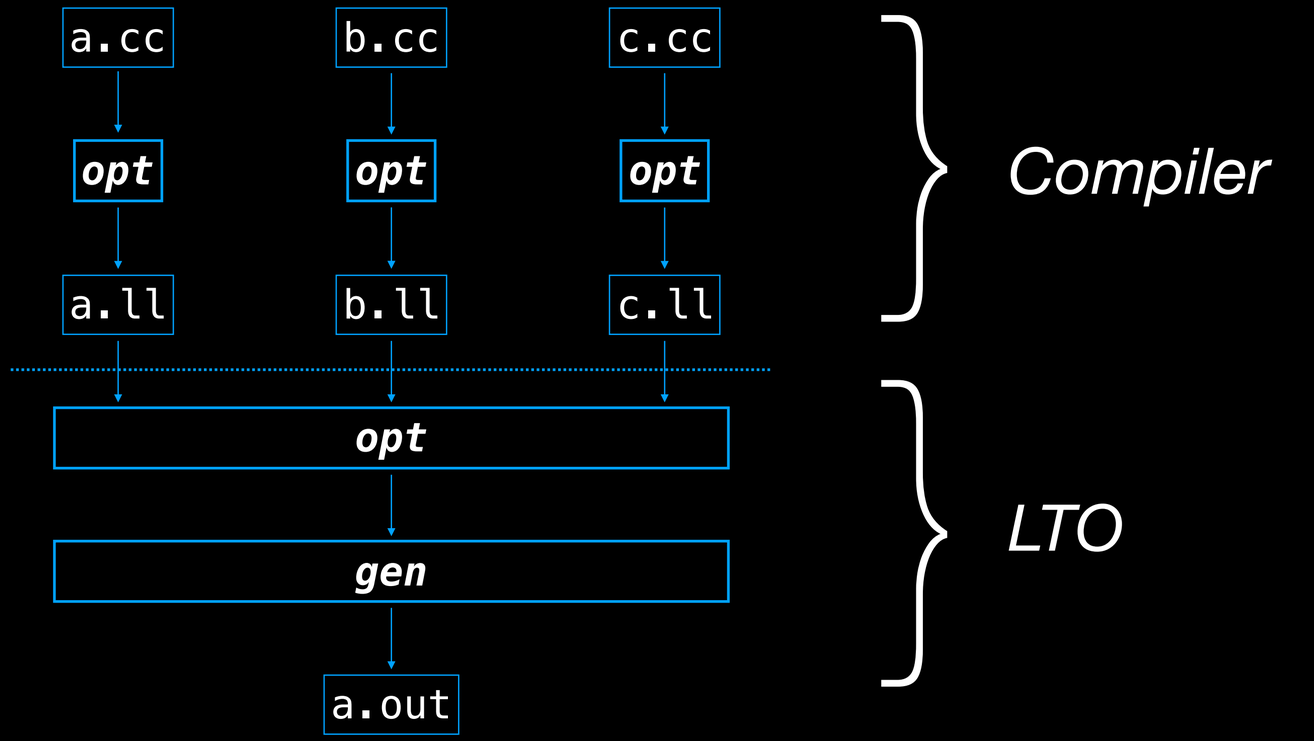
Give me the magic flag!



We add a magic *-flto* flag to the compiler invocation, and behold!

With the LTO (Link Time Optimization) flag, the compiler is again able to inline across the different *.cc* files.

What does LTO do?



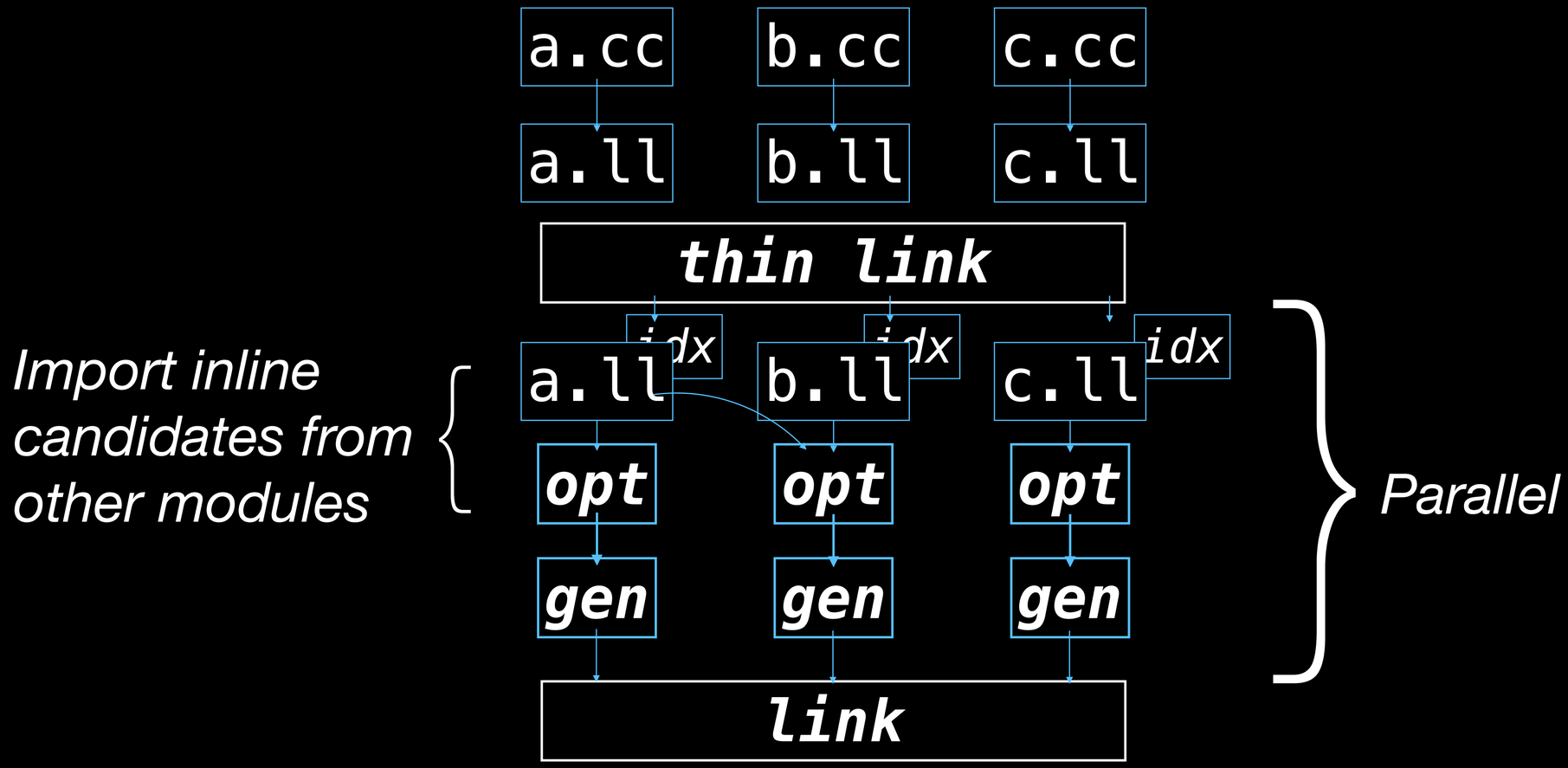
What’s wrong with LTO

Let’s call this *conventional* (or *monolithic*) *LTO.* It’s useful, but:

- **slow**, because the final optimization step is serial and can’t be parallelized

- **non incremental**, because changing a single source file causes a lot of work to be wasted

Parallel LTO



Recap about LTO

- LTO allows the compiler to jointly inline and optimize functions defined in different translation units.
- Conventional, monolithic LTO doesn’t scale well for larger programs, both in terms of time and memory.
- Parallel LTO approaches, gcc’s WHOPR and LLVM’s ThinLTO scale to large projects, like Chrome and Firefox.

Feedback Directed Optimization (or PGO)

Back to inlining

Q: How does the compiler decide which functions to inline?

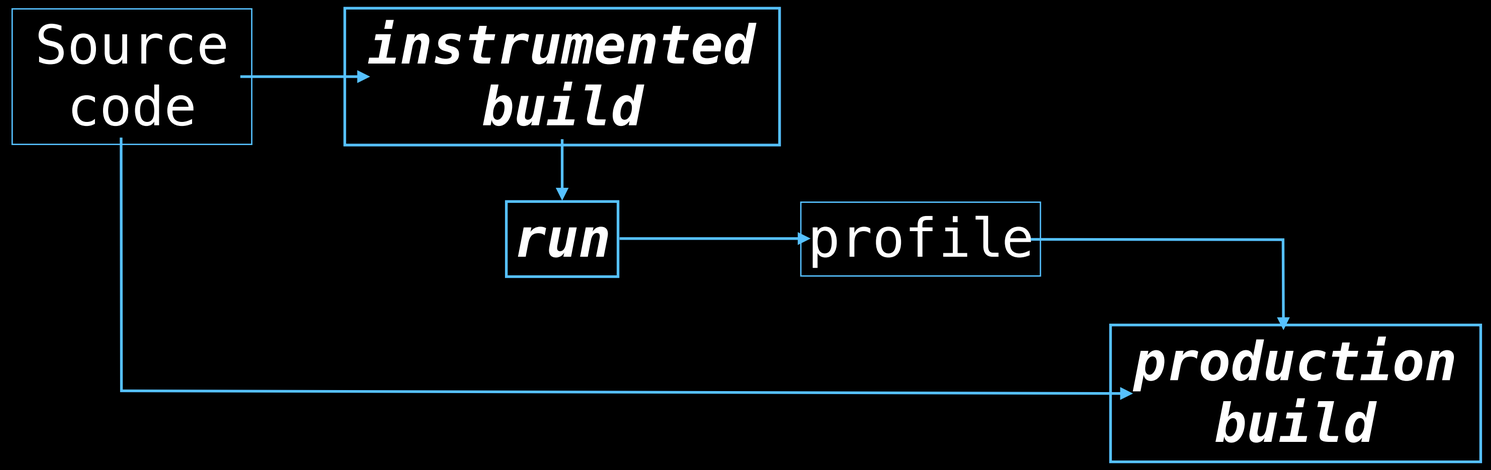
It usually uses manually-tuned heuristics, but maybe we can do better..

Profile Guided Optimization

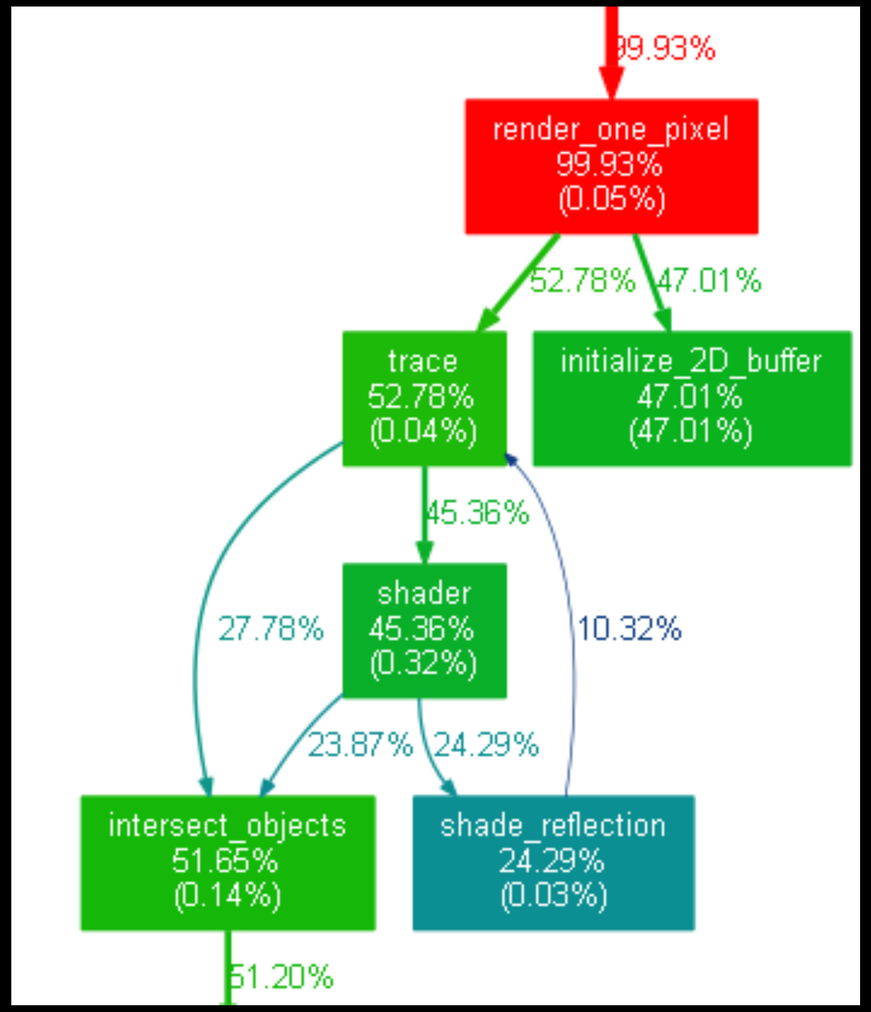
- The idea is to run the program, profile its runtime, and feed that information back to the compiler to inform its optimization decisions.
- **Examples**: inline decisions, loop unrolling, code layout in branches

PGO in practice

To use Feedback Driven Optimization, the build process must be extended:



The call graph



Each **node** in the call graph is a function body, each **edge** is a function call.

Edges are annotated with the likelihood of that branch being taken at runtime.

LTO and PGO (LIPO)

Idea: use the call graph for inlining decisions with a greedy clustering algorithm (LIPO, built by Google for gcc, 2010).

1. Compute the sum total of all dynamic call edge counts.
2. Create an array of sorted edges in descending order of their call counts.
3. Find the cutoff call count:
   (a) Iterate through the sorted edge array and add up the call counts.
   (b) If the current total count reaches 95% of the count computed at step 1, stop. The cutoff edge count is the count of the current edge. A edge is considered *hot* if its count is greater than the cutoff count.
4. Start module grouping: For each call graph node, find all nodes that are reachable from it in a reduced call graph, which contains only hot edges. Add the defining modules of the reachable nodes to the module group of the node being processed.

Figure 4. Greedy clustering algorithm for module group formation.

AutoFDO

- Uses a **sampling-based** approach that doesn't need instrumentation.
- The overhead is really small thanks to CPU hardware counters and the *perf* tool in Linux. Production servers can be profiled while running!
- Just a few lines to use it in LLVM

```
$ clang++ -O2 -g code.cc -o code
$ perf record -b ./code
$ create_llvm_prof --binary=./code --out=code.prof
$ clang++ -O2 -g -fprofile-sample-use=code.prof code.cc -o code
```

Some numbers

How to explore FDO and PGO

- Build `binutils` with plugin support
- Build `clang` from master
- Download LLVM's `test-suite`, which comes with many nice benchmarks
- Profit

Building a nice linker

```
$ git clone --depth 1 git://sourceware.org/git/binutils-gdb.git binutils
$ mkdir build
$ cd build
$ ../binutils/configure --enable-gold --enable-plugins --disable-werror --prefix=$HOME/clang9
$ make all-gold
$ make install-gold
```

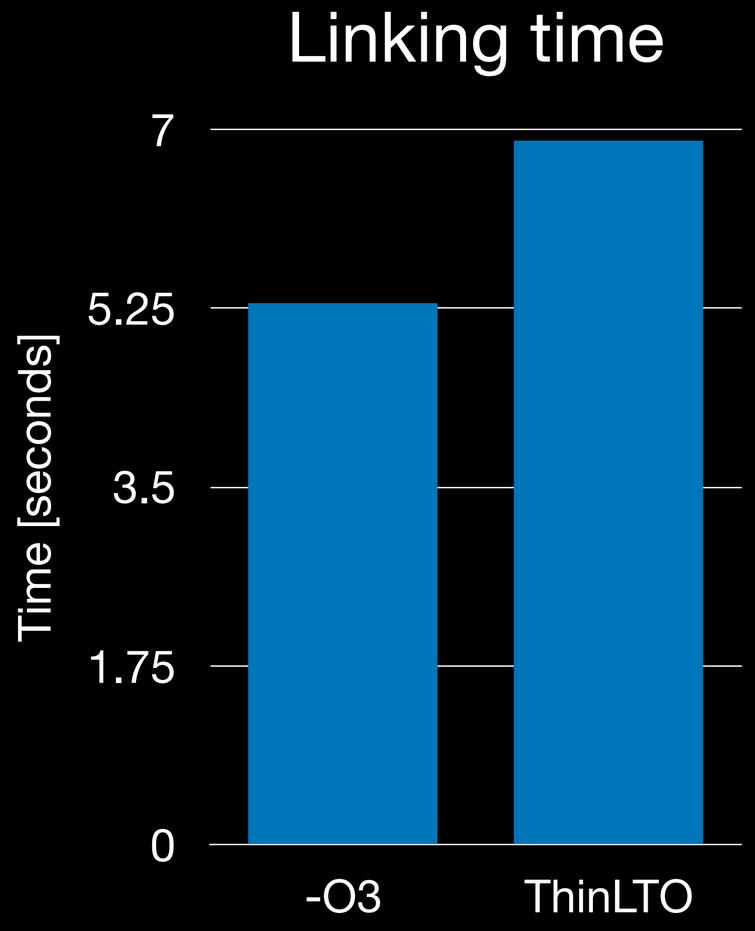
Building clang and the linker plugin

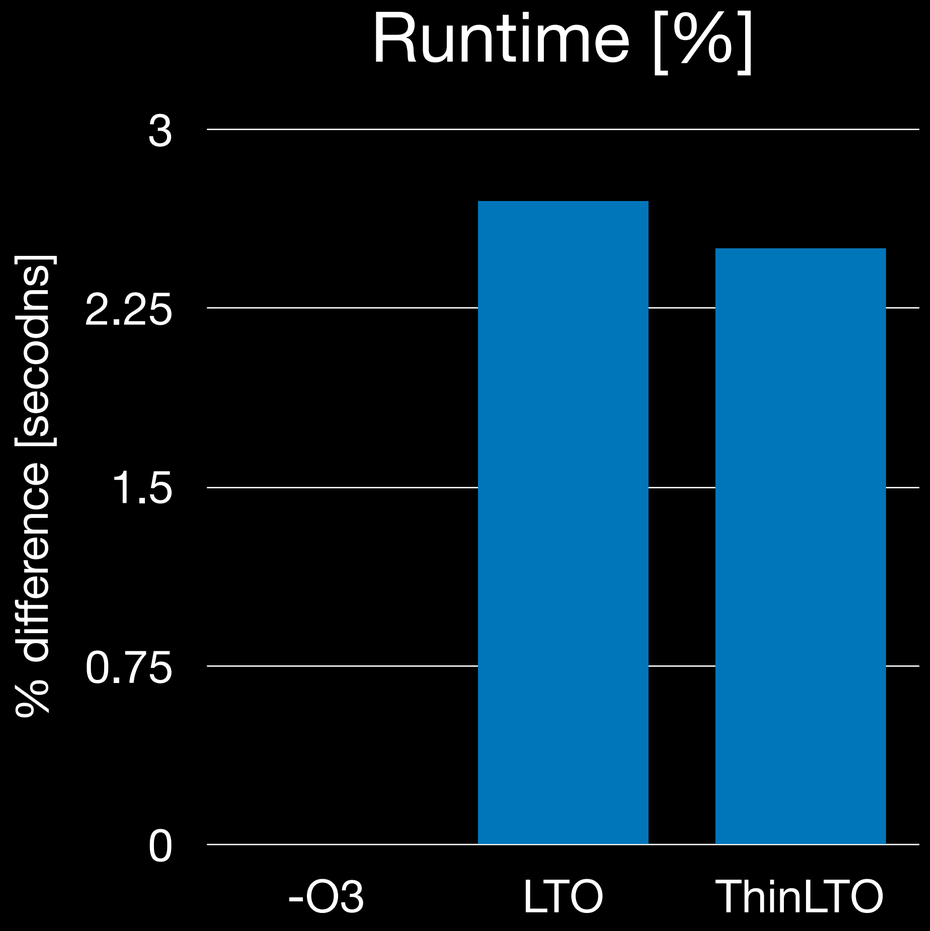
```
$ cmake \
   ../llvm  \
   -DLLVM_ENABLE_PROJECTS="clang" \
   -DCMAKE_BUILD_TYPE=Release \
   -GNinja \
   -DLLVM_BINUTILS_INCDIR=$HOME/code/binutils/include \ -
DCMAKE_INSTALL_PREFIX=$HOME/clang9
$ ninja install
```

Building the clang benchmark suite

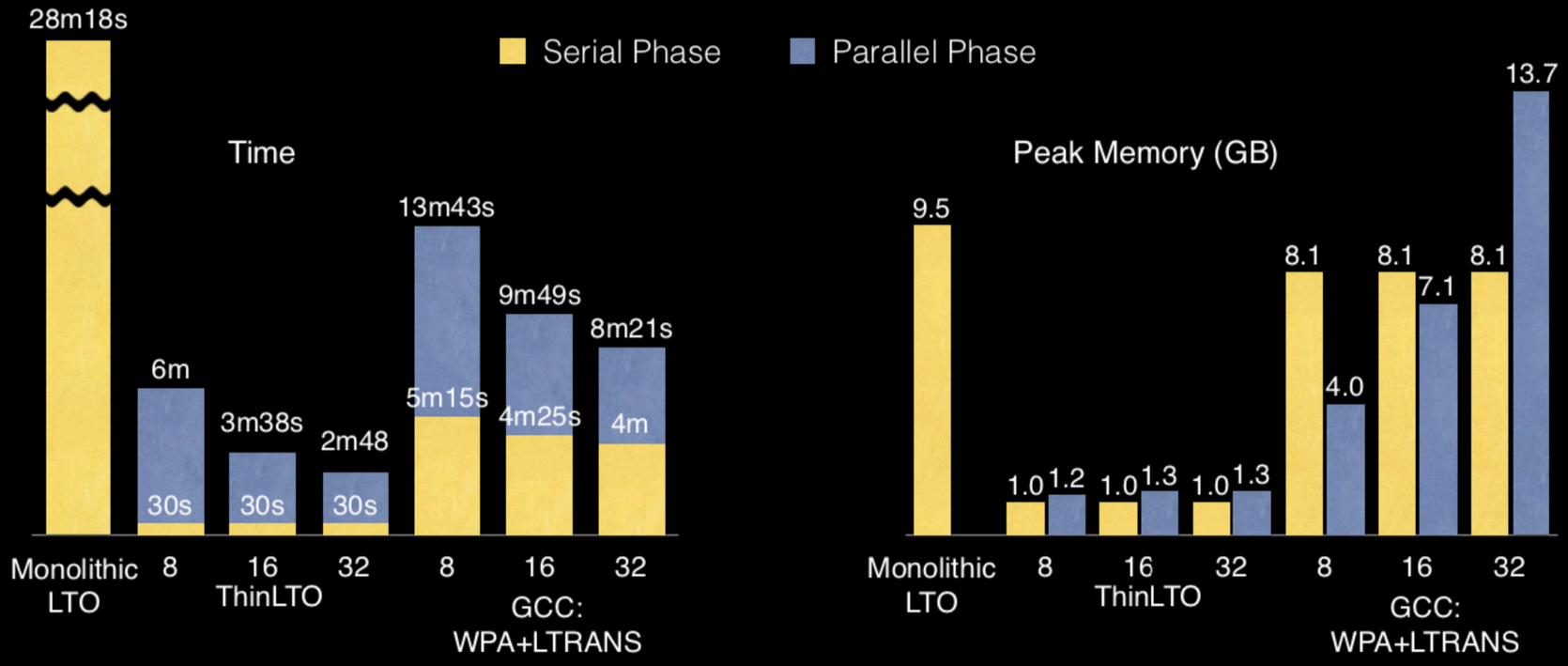
```
$ PATH=$HOME/clang9/bin:$PATH cmake \
   .. \
   -DCMAKE_C_COMPILER=$HOME/clang9/bin/clang \
   -GNinja \
   -DTEST_SUITE_BENCHMARKING_ONLY=ON \
   -DCMAKE_AR=$HOME/clang9/bin/llvm-ar \
   -DCMAKE_NM=$HOME/clang9/bin/llvm-nm \
   -DCMAKE_RANLIB=$HOME/clang9/bin/llvm-ranlib \
   -C ../cmake/caches/ReleaseLTO.cmake
```

Benchmarks


Linking time
Time [seconds]
7
5.25
3.5
1.75
0
-O3
ThinLTO


Runtime [%]
% difference [secodns]
3
2.25
1.5
0.75
0
-O3
LTO
ThinLTO

gcc WHOPR vs clang



Saving space in the Linux kernel

First, with LTO disabled:

```
$ make stm32_defconfig
$ make vmlinux
$ size vmlinux
   text    data     bss     dec     hex filename
1704024  144732  117660 1966416  1e0150 vmlinux
```

And with LTO enabled:

```
$ ./scripts/config --enable CONFIG_LTO_MENU
$ make vmlinux
$ size vmlinux
   text    data     bss     dec     hex filename
1281644  142492  112985 1537121  177461 vmlinux
```

From: *Shrinking the kernel with link-time optimization,* LWM.net

Thanks/Questions

nicolovaligi.com