Google Open Source

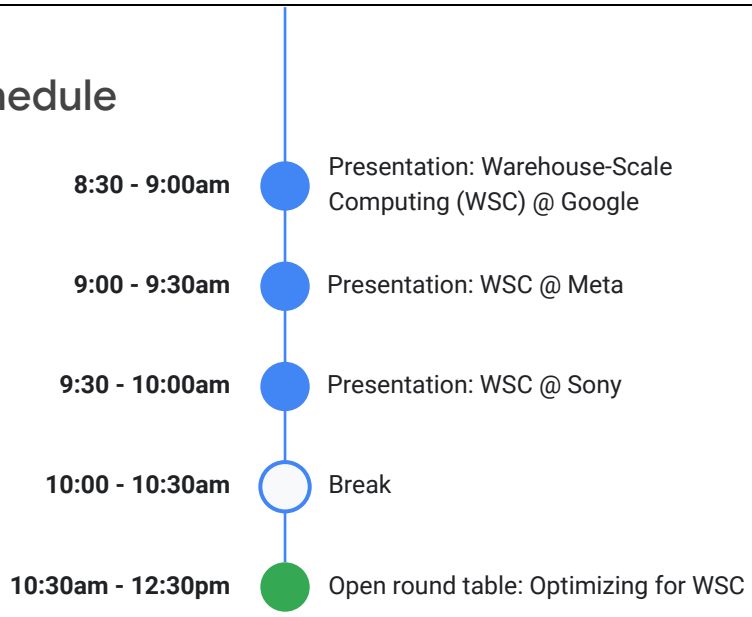
Practical Compiler Optimizations for Warehouse-Scale Applications

Daniel Hoekwater (he/him) - Google
Teresa Johnson (she/her) - Google
October 10, 2023

Katya Romanova - Sony
Matthew Voss - Sony
Konstantin Belochapka - Sony

Maksim Panchenko - Meta

Workshop Schedule



Google Open Source

Google Open Source

Warehouse-Scale Computing at Google

Daniel Hoekwater (he/him) - Google
Teresa Johnson (she/her) - Google
October 10, 2023

Agenda

- Motivation & Workshop Goals
- Performance Characteristics of Google Workloads
- Bread-and-Butter Optimizations at Google
- Deep Dive: ThinLTO
- Feedback-Driven Optimizations (iFDO, CSFDO, AFDO, Propeller)
- Deep Dive: Propeller

Motivation & workshop goals

- Motivation
  - Warehouse-scale and desktop apps are like apples and oranges
  - Google develops optimizations for WSC, and we're far from the only ones; we'd like to hear from you!
- Goals: align with LLVM community on
    - What do WSC workloads look like?
    - What optimizations matter most for WSC?
    - Is there overlap between companies? Room for collaboration?

Google Open Source

Motivation:

Warehouse-scale computing workloads differ greatly from desktop computing applications (1). However, it is common to overestimate the representativeness of desktop workloads, such as SPEC CPU (2). Google uses many first-party compiler optimizations (e.g. FDO, ThinLTO) for WSC workloads, but we're not the only ones who care about server-side performance. LLVM thrives on communication; we want to hear your experiences with optimizing the same space and work together to identify opportunities for collaboration.

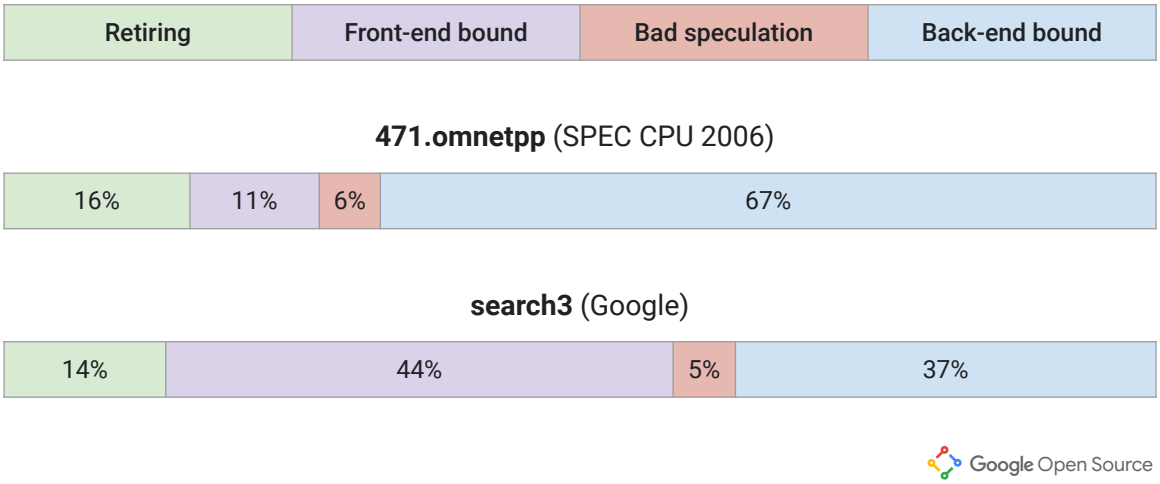
1. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers
2. RETROSPECTIVE: Profiling a warehouse-scale computer

Google workloads from the top down

Google Open Source

Background: top-down analysis

- Method for performance analysis & counters architecture (A Yasin, 2014)



Top-down analysis is an approach for categorizing processor pipeline slots. The classification criteria are fairly simple (1):

- Is a uop issued?
- If yes...
  - Does the uop ever retire?
  - If yes... classify as: Retiring
  - If no... classify as: Bad Speculation
- If no..
  - Backend stall?
  - If yes... classify as: Backend Bound
  - If no... classify as: Frontend Bound

Google workloads have higher icache miss rates and lower IPC than desktop benchmarks (2). Note: over time, memory bandwidth has become more scarce, leading to backend stalls making up a growing fraction of cycles (3).

1. A Top-Down method for performance analysis and counters architecture
2. Profiling a warehouse-scale computer
3. RETROSPECTIVE: Profiling a warehouse-scale computer

Characteristics of Google workloads (spoiler: they're big)

- Massive binaries with large text sections
- Many basic blocks, most of which are cold
- Incremental builds
- Shared code/modules
- Wide dependency trees

Google Open Source

Google's workloads are massive, which drives our optimization decisions. (1)

- Huge instruction footprint, high fraction of cycles spent on front-end stalls (1)
  - Many basic blocks, most of which are cold (1)
    - "Generally, in roughly half of even the hottest functions, more than half of the code bytes are practically never executed, but likely to be in the cache." (1)
  - Large text sections
    - "Instruction footprint (...) over 100 times larger than (...) an L1 instruction cache" (1)
- Builds are incremental, i.e. they don't change all at once (3)
- Serial builds are infeasible (2)
  - Shared code/modules (2)
  - Wide dependency tree (2)

1. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers
2. Smart Build Targets Batching Service at Google
3. ThinLTO: Scalable and Incremental LTO

Bread-and-butter optimizations at Google

- Cached, distributed build: Bazel w/ BuildRabbit
- Parallelized, incremental link-time optimizations: ThinLTO
- Feedback-driven compiler optimizations: FDO
- Profile-guided relinking optimizations: Propeller

- ... and many more: TCMalloc, Hugepages, Memprof (WIP)

Google Open Source

Bazel (Building a Distributed Build System at Google Scale)
- Cached
  - Incremental builds and shared modules mean independent builds have lots of overlapping compile units
- Distributed
  - Massive binaries can't be serially-compiled
  - Wide dependency trees are easily parallelized

ThinLTO (ThinLTO: Scalable and incremental LTO)
- Parallelized: motivated by...
  - Massive binaries with large text sections
  - Distributed build
- Incremental: motivated by...
  - Incremental builds

FDO (AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications)
- Motivated by...
  - Many basic blocks, most of which are cold

Propeller (Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications)
- Motivated by...
  - Massive binaries with large text sections

    - Distributed build
    - Many basic blocks, most of which are cold
    - Incremental builds

Additional optimizations:

- Software prefetching (AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers, APT-GET: profile-guided timely software prefetching)
- Custom tuned memory allocation: TCMalloc (google/tcmalloc)
- Hugepages (Temeraire: Hugepage-Aware Allocator | tcmalloc)
- Profile-guided heap optimization (Work in progress) ([llvm-dev] RFC: Sanitizer-based Heap Profiler)

TCMalloc

- Thread-cached memory allocation
  - Lots of threads: allocate with per-CPU caches
  - Opaque allocation: expose allocation metrics and tuning knobs
  - Place allocated memory in hugepages
  - 👍 reduce contention during alloc
  - 👍 improve TLB through hugepage placement
  - 👎 can degrade performance with bad memory allocator

```
$ # Simply add -ltcmalloc option*
$ clang -ltcmalloc -O3 example.c -o example
```

*alternatively, see TCMalloc Quickstart

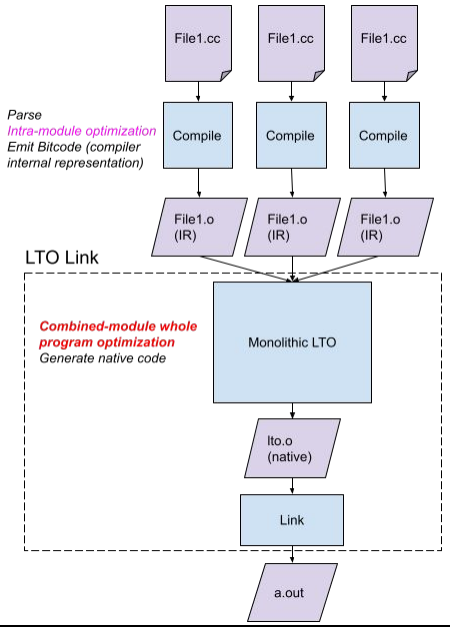
Google Open Source

hugepages: Temeraire [Temeraire: Hugepage-Aware Allocator | tcmalloc]

Note: There are a couple of versions of tcmalloc: the old, unmaintained version (which is bundled with gperftools) and the newer, released version (which is independently hosted on GitHub). For all intents and purposes, the old version is dead; if you want to test out tcmalloc for your use case, we recommend you start with the [TCMalloc Quickstart].

Deep dive: ThinLTO

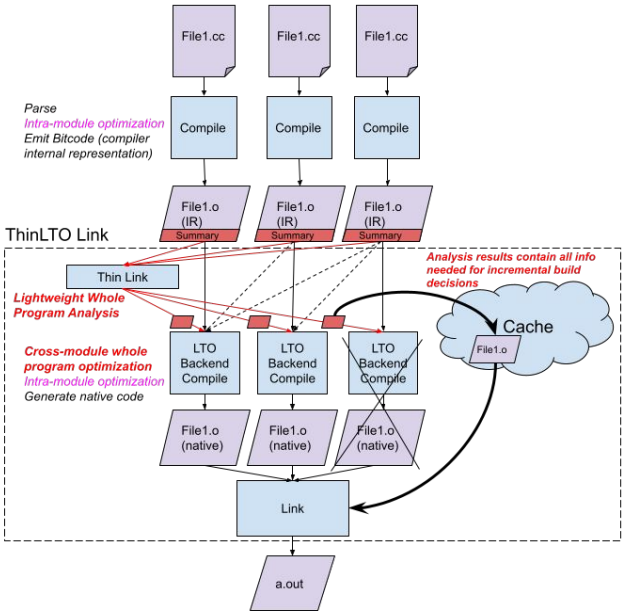
Monolithic LTO



- First part of compile is fully parallel
- First part of compile is fully incremental
- Enables cross-module whole program optimization
- LTO compilation is not parallel
- LTO compilation is not incremental
- LTO compilation does not scale in memory

Traditionally, link time optimization works by compiling to objects of compiler IR (instead of native), with the link merging all IR into a single unit. The optimization pipeline is then run on the merged unit before code generating a single native object feeding a native link. This enables the most aggressive cross-module and whole program optimizations, however, the large size of the merged unit is not scalable in time or memory, nor is it incremental (any change to any input file requires full reoptimization of the entire program).

ThinLTO



- First part of compile is fully parallel
- First part of compile is fully incremental
- Enables cross-module whole program optimization
- LTO Thin Link summary-based analysis is serial but very lightweight (memory, time)
- LTO backend compilation is fully parallel
- LTO backend compilations can be distributed
- LTO backend compilation is fully incremental
- LTO backend compilation scales in memory

ThinLTO addresses this by performing the serial whole program step on small lightweight summaries of each IR object file. The analysis results and decisions (such as cross-module imports for enabling cross-module inlining) are then fed to completely independent ThinLTO backend compilations. These backend compilations first apply the whole program decisions, including importing copies of function bodies as directed from other IR objects, before invoking the normal optimization and code generation pipelines. The imported code copies are dropped after inlining. The generated native objects go through a normal native link.

The entire ThinLTO link can either be performed in a single linker process (the default), where the parallel backends are invoked as threads, which makes the user interface the same as non-LTO compilation. Or they can be performed as separate compilation processes and integrated with a distributed build system. Because the analysis results contain all the information needed to determine whether each ThinLTO backend needs to be re-executed (e.g. the whole program decisions affecting each module, as well as hashes of the individual IR files), we can use that to access a build cache and make incremental build decisions.

ThinLTO

- After compilation:
  - Generate bitcode summaries in parallel
  - Perform LTO IR optimizations and codegen in parallel
- Read indexed summaries and perform serial whole program optimization
- 👍 far superior scaling than LLVM/GCC LTO
- 👍 incremental and distributed build friendly
- 👍 safe for always-on optimization, doesn't degrade performance
- 👎 not quite as effective as standard LTO

```
$ # Simply add -flto=thin option
$ clang -flto=thin -O2 file1.o file2.o -o a.out
```

Google Open Source

Since Google binaries are huge, standard LLVM LTO is infeasible.
ThinLTO scales LTO by:

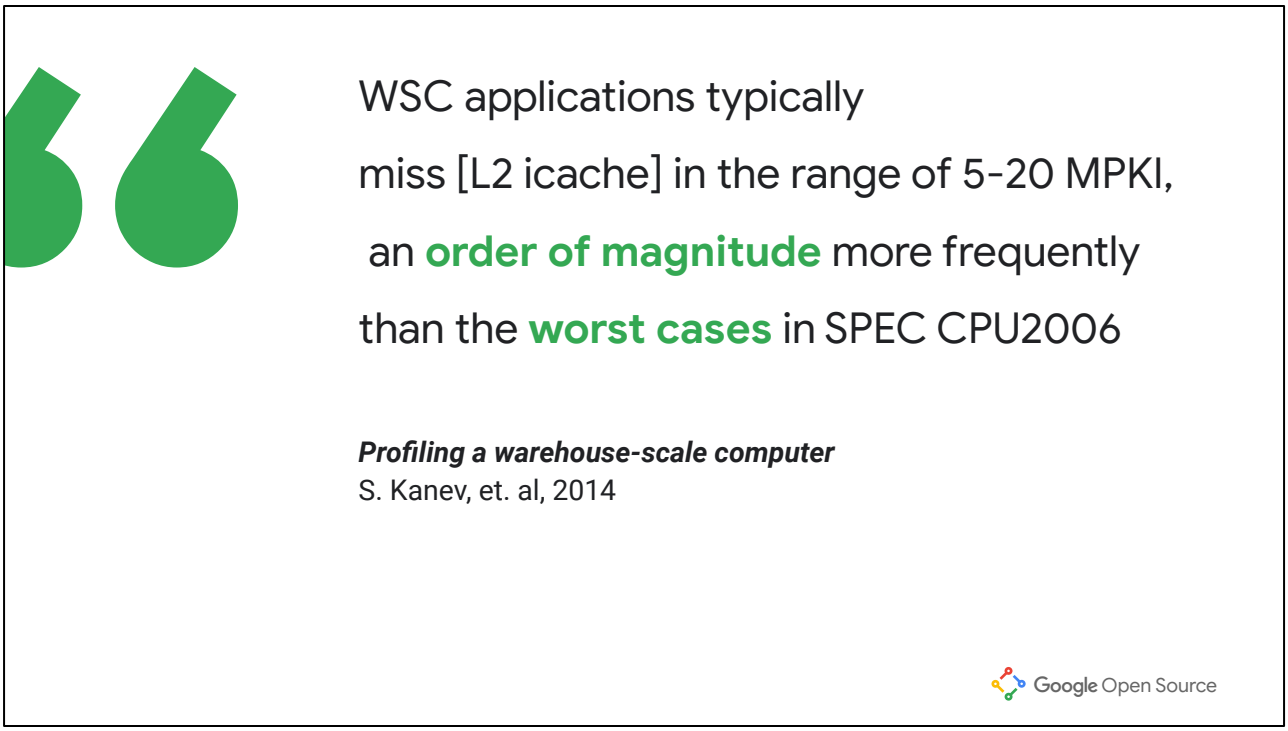
- Generating bitcode summaries in parallel
- Performing LTO IR optimizations and codegen in parallel

The final step of ThinLTO is reading the indexed summaries and performing a single serial whole program optimization

Since bitcode summary generation, LTO IR optimizations, and codegen are all parallelized stages, they have suitable scaling for distributed build.
Since each parallel task is deterministic, previous results are easily cached, making incremental builds require incremental work.
Since ThinLTO is safe for reasonable inputs and doesn't degrade performance, it can be applied to a heterogeneous set of workloads all at once, rather than manually tailoring the optimization for each workload.
Source: [ThinLTO: Scalable and Incremental LTO]

ThinLTO is supported by LLVM and easily enabled with a single flag.
For more details, see the LLVM documentation:
[ThinLTO — Clang 18.0.0git documentation]

Feedback-Driven Optimization (FDO)



The primary motivation for FDO is mitigating frontend stalls, which occur in warehouse-scale applications much more often than in traditional benchmarking suites such as SPEC CPU (1).

1. Profiling a warehouse-scale computer

Full quote: "The most probable cause is a non-negligible fraction of long latency instruction miss events – most likely instruction misses in the L2 cache. Such a hypothesis is confirmed by the high exhibited L2 instruction miss rates from Figure 8. WSC applications typically miss in the range of 5-20 MPKI, an order of magnitude more frequently than the worst cases in SPEC CPU2006, and, at the high end of that interval, 50% higher than the rates measured for the scale-out workloads of CloudSuite [14]."

FDO at a glance

- Motivation: short basic blocks, most of which are cold
- Solution: use profile data to drive optimization decisions
  - Function & basic block layout
  - Function splitting
  - Function inlining
  - Loop unrolling, branch optimization
  - Speculative code motion, hoisting
  - And many more!

Google Open Source

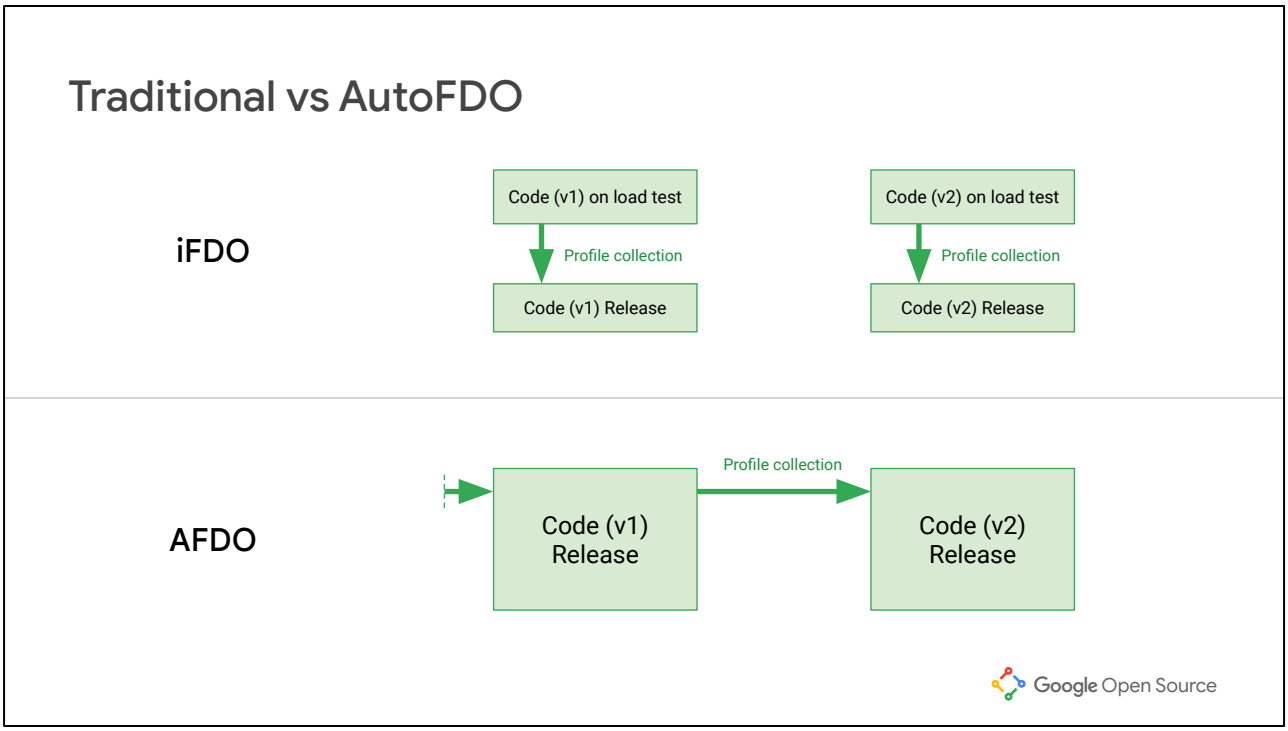
Motivation: most basic blocks are cold... like, really cold.

- "even among the hottest and most well-optimized functions in our server fleet, more than 50% of code is completely cold" (1)

Solution: use profile data to drive optimization decisions

- Function & basic block layout (2)
- Function splitting (3)
- Function inlining (4)
- Loop unrolling, branch optimization
- Speculative code motion, hoisting
- And many more!

1. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers
2. Codestitcher: Inter-Procedural Basic Block Layout Optimization
3. [llvm-dev] [RFC] Machine Function Splitter
4. ⚙ D98213 [InlineCost] Enable the cost benefit analysis on FDO



"Traditional FDO follows a three-step pattern:

1. Compile with instrumentation
2. Run a benchmark to generate representative profile
3. Recompile with the profile"

With traditional FDO, sequential releases are independent of each other and require representative benchmarks.

AutoFDO forms a loop:

1. Collect a profile from your binary running in production
2. Use the profile to generate an optimized binary, which you push to production
3. Go to step 1.

Source: [AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications]

Lowering the bar

- AutoFDO: automatic profile collection from production workloads
  - FDO requires representative load test
  - Instead, profile the workload directly
  - 👍 implicitly representative profiles
  - 👍 low bar to entry
  - 👎 vulnerable to source drift
  - 👎 relies on debug info for sample attribution

Google Open Source

AutoFDO: automatic profile collection from production workloads (1)

- FDO requires representative a load test to inform what will probably happen in production: "run a benchmark to generate representative profile"
- "The AutoFDO profile for a target program is collected directly from an optimized binary running in production."
- Since the profile comes from the workload itself, it is implicitly a "representative loadtest."
  - "AutoFDO has simplified the deployment of FDO in our datacenters to require only adding some compiler flags. These advances have led to an 8X increase in customer adoption"
  - Release build latency is faster without an additional load testing step
- Vulnerable to source drift: "AutoFDO collects profiles from production binaries, then uses these profiles when building new releases in which the source code for a binary may have changed relative to the profiled binary."
- Relies on debug info and profile weights, which are not always preserved through optimization passes. (2)
  - Keeping around debug info takes up additional storage
  - More importantly, attributing samples to a specific IR (and less successfully, Machine IR) basic block with debug info is imperfect

1. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications
2. [RFC] Profile Information Propagation Unittesting - IR & Optimizations - LLVM Discussion Forums

Quality profiles from the fleet

- FSAFDO: flow-sensitive AutoFDO
  - AFDO profiles merge samples from cloned basic blocks
  - Add hierarchical metadata to discriminate between cloned blocks
  - 👍👍 gains profile granularity
  - 👎 increases profile size and compile time (≤5%)
  - 👎 doesn't handle ambiguous profiles

Google Open Source

FSAFDO: flow-sensitive AutoFDO

- A basic block can behave differently depending on which block was executed directly before it; flat basic block profiles such as those collected in AFDO can't account for this difference.
- Add extra metadata to between map profile samples back to basic blocks
- Gets more precise profiles, which benefit basic block placement and branch folding.
- Increases profile size (to account for metadata)
- If two identical source lines are merged (or outlined), profile samples for the merged block are still ambiguous.

Source: [llvm-dev] [RFC] Control Flow Sensitive AutoFDO (FS-AFDO)

Refining profile quality with *FDO

- Instrumented FDO: enables sample-to-block mapping beyond debug info
  - AFDO requires debug info to identify blocks
  - Use instrumentation to recover control flow
  - 👍 stable speedup 👍 high sample -> block correlation
  - 👎 requires representative loadtest
- CSFDO: context-specific profiles
  - FDO has ambiguity between `f(x)` inlined by `g(x)` vs by `h(x)`
  - Collect another round of profiles after inlining
  - 👍 increases profile quality 👎 requires additional profiling

Google Open Source

Source: https://reviews.llvm.org/D54175

Helpful resource: The many faces of LLVM PGO and FDO

In practice

- AutoFDO

```
$ clang -O3 example.c -o example # Build 1
$ perf record -b ... ; create_llvm_prof ... # Profile
$ clang -fprofile-sample-use=profile example.c -o example # Build 2
```

- CSFDO

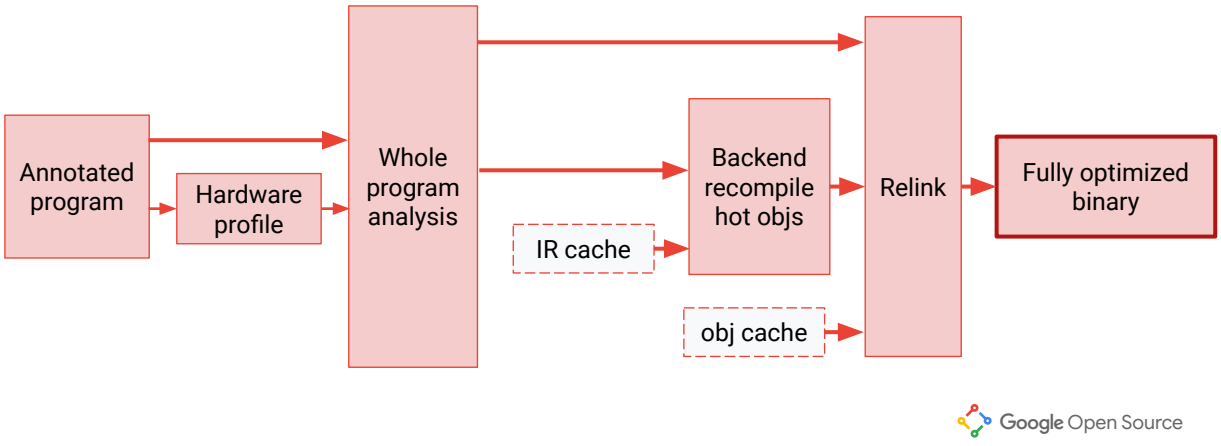
```
$ clang -fprofile-generate=$FDO_DIR example.c -o example # FDO-instrumented build
$ ./example # FDO profile
$ llvm-profdata merge -output=example.profdata $FDO_DIR
$ clang -fprofile-use=example.profdata -fcs-profile-generate=$CSFDO_DIR \
       example.c -o fdo_example # CSFDO-instrumented build
$ ./fdo_example # CSFDO profile
$ llvm-profdata merge -output=fdo_example.profdata $CSFDO_DIR example.profdata
$ clang -fprofile-use=fdo_example.profdata \
      example.c -o cs_example # CSFDO-optimized build
```

Sources: [FDO: Magic “Make My Program Faster” compilation option?](),

Deep dive: Propeller

Propeller: "A Framework for Post Link Optimizations"

- Propeller: "A Framework for Post Link Optimizations"
  - *FDO has limited view of the program



- *FDO has a limited view of the program
- Collect a round of sample profiles after linking
- Use precise post-link profiles to...
  - refine backend optimizations (using cached IR)
  - perform whole program analysis
- Relink the binary using cached objects

- Whole program analysis, relink are serial operations
- Backend recompile of hot objects is distributed and semi-incremental

Source: [Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications]

Near ground truth profiles with Propeller

- 👍 precise profiles don't require propagation
- 👍 whole-program block-level layout
- 👍 scalable & incremental due to IR and obj caching
- 👎 requires an additional round of profiling

Google Open Source

- Since profiles are directly mapped to machine basic blocks with the BB_ADDR_MAP, attributing profile samples to blocks doesn't require any additional work
- Applying profiles after linking makes it possible to factor in the whole program when making layout decisions
- The additional round of profiling provides data points for evaluating the consequences of previous FDO decisions such as loop unrolling, inlining, etc.
- Because Propeller uses IR caching, it enables code layout reoptimization without requiring disassembly. However, it *does* miss out on disassembly-driven optimizations such as peephole optimization, function alignment, conditional tail call simplification, etc. (Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization)

Source: Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications

In practice

```
$ # Propeller-annotated build
$ clang -fprofile-use=example.profdata -fbasic-block-sections=labels \
        example.c -o fdo_example
$ # Propeller profile
$ perf record -b ./fdo_example
$ create_llvm_prof --format=propeller --binary=fdo_example \
        --profile=perf.data --out=$PROP_DIR/cc_profile.txt \
        --propeller_symorder=$PROP_DIR/ld_profile.txt
$ # Propeller-optimized build
$ clang -fprofile-use=example.profdata \
      -fbasic-block-sections=list=propeller_cc_profile \
      --Wl,--symbol-ordering-file=propeller_ld_profile \
      example.c -o prop_example
```

Google Open Source

Full commands:

```
$ # FDO-instrumented build
$ clang -fprofile-generate=$FDO_DIR example.c -o
example
$ # FDO profile
$ ./example
$ llvm-profdata merge -output=example.profdata
$FDO_DIR
$ # Propeller-annotated build
$ clang -fprofile-use=example.profdata
-fbasic-block-sections=labels \
        example.c -o fdo_example
$ # Propeller profile
$ perf record -b ./fdo_example
$ create_llvm_prof --format=propeller
--binary=fdo_example \
        --profile=perf.data
--out=$PROP_DIR/cc_profile.txt \
        --propeller_symorder=$PROP_DIR/ld_profile.txt
$ # Propeller-optimized build
```

```
$ clang -fprofile-use=example.profdata \

-fbasic-block-sections=list=propeller_cc_profile \

--Wl,--symbol-ordering-file=propeller_ld_profile \
      example.c -o prop_example
```

Source: FDO: Magic “Make My Program Faster” compilation option?, GitHub - google/llvm-propeller: PROPELLER: Profile Guided Optimizing Large Scale LLVM-based Relinker

Conclusion

Summary & Implications

- Google WSC workloads are massive
  - Higher **icache miss rates** than desktop applications
  - **Distributed build** necessitates **distributed**, **incremental** optimizations
  - **FDO** is a must for performance-sensitive applications
- Important considerations for future server-side optimization work
  - Preserve **debug info** and **branch profile** information
  - Prioritize optimization **scalability** and **safety**
- We'd love to hear your experiences!

Thank you!

Daniel Hoekwater

Software Engineer

hoekwater@google.com

Teresa Johnson

Software Engineer

tejohnson@google.com

Google Open Source

We'll be right back!

(discussion at 10:40 AM)

Google Open Source