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

8:30 - 9:00am  Presentation: Warehouse-Scale Computing (WSC) @ Google

9:00 - 9:30am  Presentation: WSC @ Meta

9:30 - 10:00am Presentation: WSC @ Sony

10:00 - 10:30am Break

10:30am - 12:30pm Open round table: Optimizing for WSC
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 workloads from the top down
Background: top-down analysis


<table>
<thead>
<tr>
<th></th>
<th>Retiring</th>
<th>Front-end bound</th>
<th>Bad speculation</th>
<th>Back-end bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>471.omnetpp</td>
<td>16%</td>
<td>11%</td>
<td>6%</td>
<td>67%</td>
</tr>
<tr>
<td>search3 (Google)</td>
<td>14%</td>
<td>44%</td>
<td>5%</td>
<td>37%</td>
</tr>
</tbody>
</table>
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
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)
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
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
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

- 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
```
Feedback-Driven Optimization (FDO)
WSC applications typically miss [L2 icache] in the range of 5-20 MPKI, an order of magnitude more frequently than the worst cases in SPEC CPU2006

*Profiling a warehouse-scale computer*
S. Kanev, et. al, 2014
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!
Traditional vs AutoFDO

**iFDO**

- Code (v1) on load test
- Profile collection
- Code (v1) Release

**AFDO**

- Code (v1) Release
- Profile collection
- Code (v2) Release
Lowering the bar

- AutoFD0: 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
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
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
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
  ```
Deep dive: Propeller
Propeller: "A Framework for Post Link Optimizations"

- Propeller: "A Framework for Post Link Optimizations"
  - *FDO has limited view of the program
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
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
```
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
We'll be right back!
(discussion at 10:40 AM)