Profiling Challenges for PGO Pipeline

Maksim Panchenko
Meta

Overview

- History of PGO at Meta
- Optimal PGO Pipeline
- Real-world Use Cases
- Future Improvements

WSC Disclaimer

Most of optimizations for WSC are also applicable to large client-side apps:

- Compilers
- WWW Browsers
- OS Kernels
- Games

WSC Disclaimer

Most of optimizations for WSC are also applicable to large client-side apps:

- Compilers
- WWW Browsers
- OS Kernels
- Games

Key focus:

- 0.5% CPU time improvement is significant
- Code size is secondary to application performance
    - Larger, faster code (e.g., due to aggressive inlining) is considered better

PGO History at Meta

“Hot Text”

- Linker Map / Function Order version of PGO
- Production Profile → HFSort/C3 → Linker Map File
  - *Optimizing function placement for large-scale data-center applications*, G. Ottoni and B. Maher, CGO ‘17
- Huge reduction in iTLB misses (> 40%) and CPU cycles (> 5%)
  - Ivy Bridge - “small” TLB
- Huge Pages for code
  - x86-64: 4 KB pages by default
    - 12 MB → 3000 pages
  - Remap “Hot Text” to 2 MB pages during startup
    - 12 MB → 6 pages
  - ~2% CPU time reduction on top of HFSort/C3

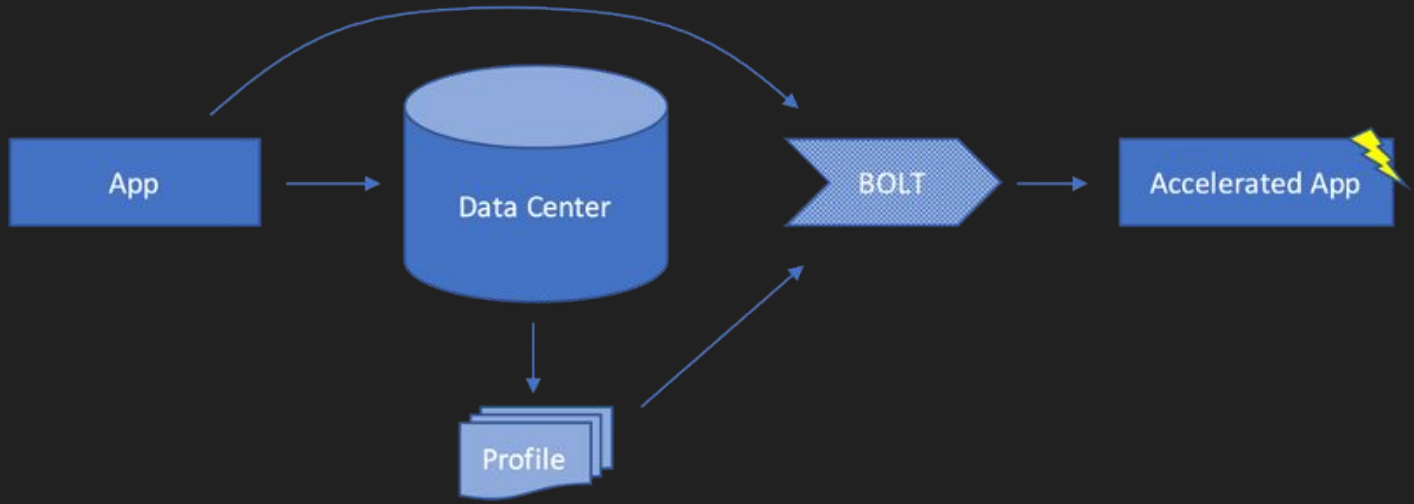
“Hot Text” Profile

- Collect *cycles/instructions* samples in production
- Update linker script / function order for every release
  - Hot fixes may or may not require an update
- Performance comparison against the release
  - Minor performance differences
  - Namespace changes

PGO in Binary Optimizer

- PGO was borked for GNU g++ w/ exceptions back in 2015
- Code layout optimizations
  - Compiler-agnostic
    - Support GCC, Clang, ICC
  - Linker-agnostic
    - BFD ld, gold, lld
- BOLT - Binary Optimization and Layout Tool
  - Open-sourced in 2018
  - *BOLT: a practical binary optimizer for data centers and beyond*, M. Panchenko et al., CGO ‘19
  - 7% CPU time improvement on top of “Hot Text” for HHVM
    - Fewer I$ misses and branch mispredictions
  - Up to 20% on top of PGO+LTO using real-world benchmarks
    - Greatly Exceeded Expectations
  - Integrated HFSort / function layout

Profiling with BOLT



BOLT Profile

- Binary-Level Profile
  - Branches and Fall-throughs recorded as offsets from function start
  - Alternative format: tied to internal CFG
  - 100% accurate when applied to the profiled binary
- No need to rebuild the binary after profiling
- Hot fixes
  - Recollect profile if possible
- Profiling time: 3 minutes for HHVM (after JIT warm-up)

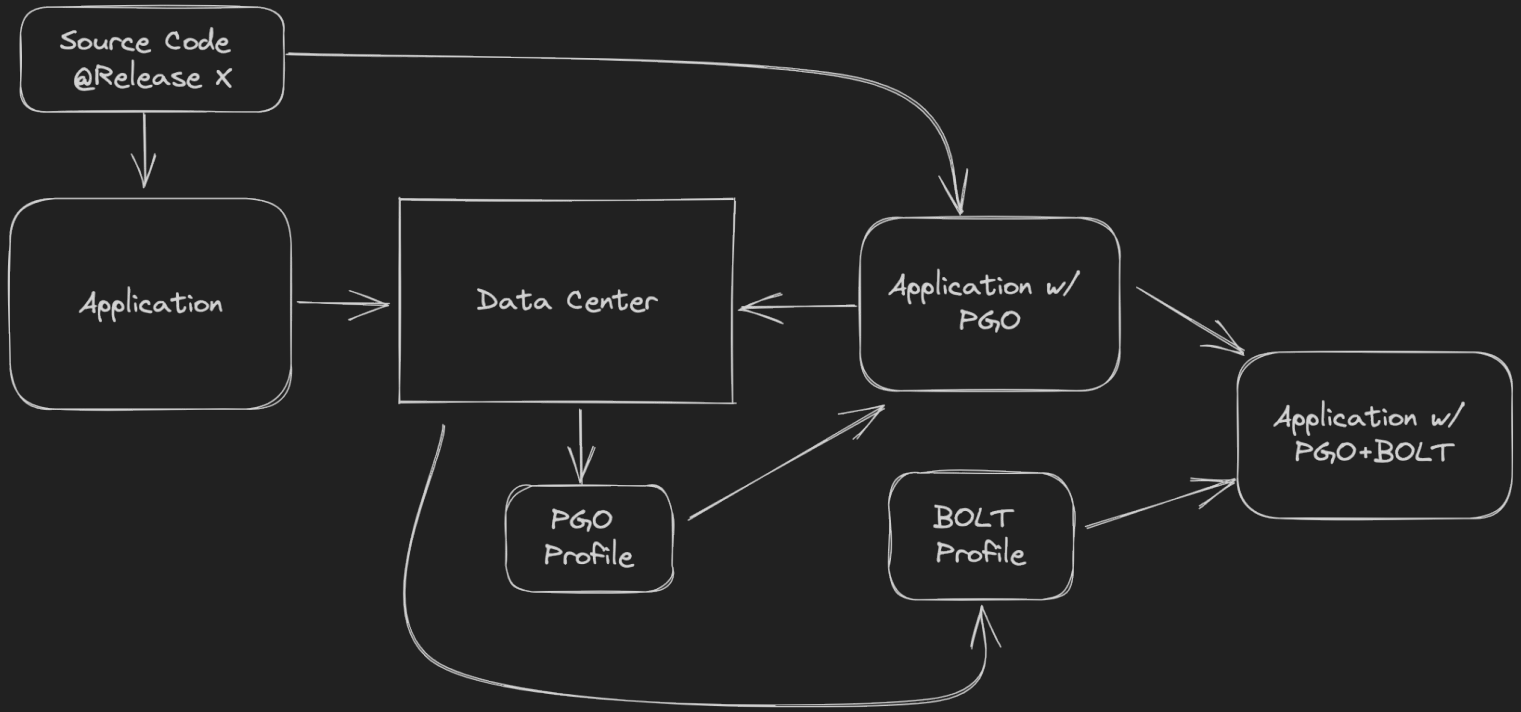
Evolution of BOLT

- LLVM project
  - Not tied to Clang
  - Golang support with Huawei patches
- Built-in instrumentation when LBR-like sampling not available
- x86-64, Aarch64, RISC-V (*)
- 64-bit ELF
  - x86-64 Linux Kernel (*)
- Lightning BOLT
  - ~1 minute to optimize large application without DWARF update
  - Seconds to rewrite Linux kernel

⏩ Meta 2023: Clang ThinLTO + PGO + BOLT

- Clang's CSSPGO Context-Sensitive Sampling PGO
  - 1-3% performance improvement over AutoFDO
  - Some services use IRPGO
- Two profiles better than one
  - Step 0: Collect CSSPGO profile
  - Step 1: Build binary with CSSPGO profile
  - Step 2: Collect BOLT profile
  - Step 3: Apply BOLT profile to binary from Step 1
    - No need to rebuild the binary

Zero-Gap Release



Zero-Gap Profile

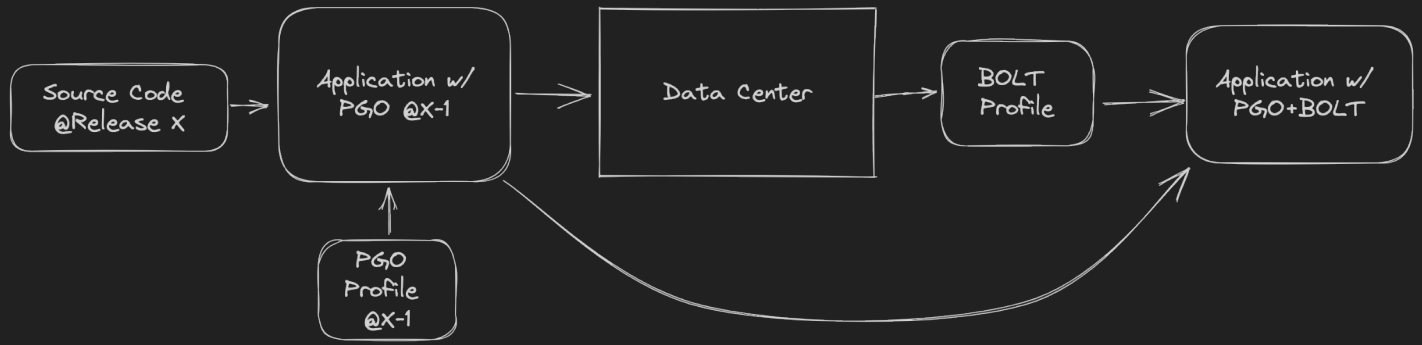
- Optimize what you profile
  - Profile collection starts after the new release is cut
- Requires prod canaries or representative workloads
  - Not always 100% representative
- Good when the release cadence is low
  - Once a week or less frequent
- Optimal pipeline if profile workload matches prod
- Drawbacks:
  - Longer release process
    - Requires separate deployment for the release
    - Two profiles have to be collected sequentially
  - Hot Fixes take longer to ship or ship with perf penalty
  - Accurate performance measurements against the release are difficult

Continuous Profiling

- Optimize with existing profile
  - Profile data is collected on previous release
- Best for high-frequency releases
  - Source code gap is “small”
- Profile is collected in production
  - No need for separate canary/deployment
- Drawback: “Stale” profile and performance loss
  - Compiler PGO has to match the profile to a different source code / IR
    - Inaccurate profile with larger source code gaps
  - BOLT: Binary is different
    - Sampling ⇒ Non-deterministic profile ⇒ Non-deterministic compiler output
    - Minor changes in CSSPGO profile lead to different inlining decisions
      - “Butterfly Effect”
      - > 50% functions not optimized

BOLT-Compatible CSSPGO Pipeline

- Mixture of Zero-Gap and Continuous profile
- CSSPGO uses continuous profile
- BOLT profile is collected on the new release
- Drawbacks:
  - Still needs separate deployment for the release
  - Perf loss due to stale CSSPGO profile



Other Profiling Challenges

Case Study - ZippyDB

- Extremely heterogeneous workload
- Best profile on canary → ~30% of functions running in production

WIP Improvements

- Stale profile matching
  - Compiler Incremental PGO: *--salvage-stale-profile*
  - BOLT: *--infer-stale-profile*
    - Uses CFG matching
    - More work required to work with BOLTed binaries
  - Closes the performance “Gap”
- Make sampling-based PGO more stable
  - Same source + “almost” identical profile ⇒ poor binary match
- Dynamic BOLT optimization
  - Leverage BOLT advantages
    - No need for sources
    - Fast code rewrite
  - Covers heterogeneous workloads well
  - No 100% parity with static BOLT using identical profiles

Thanks!