Practical Use of BOLT

AMIR AYUPOV

Eighth LLVM Performance
Workshop at CGO

March 2-6, 2024

Edinburgh, UK

Meta

Agenda

1. Introduction
2. Prerequisites
3. Profile collection
4. Usage of BOLT
5. Logs and debugging
6. Interaction with PGO

Introduction

Introduction

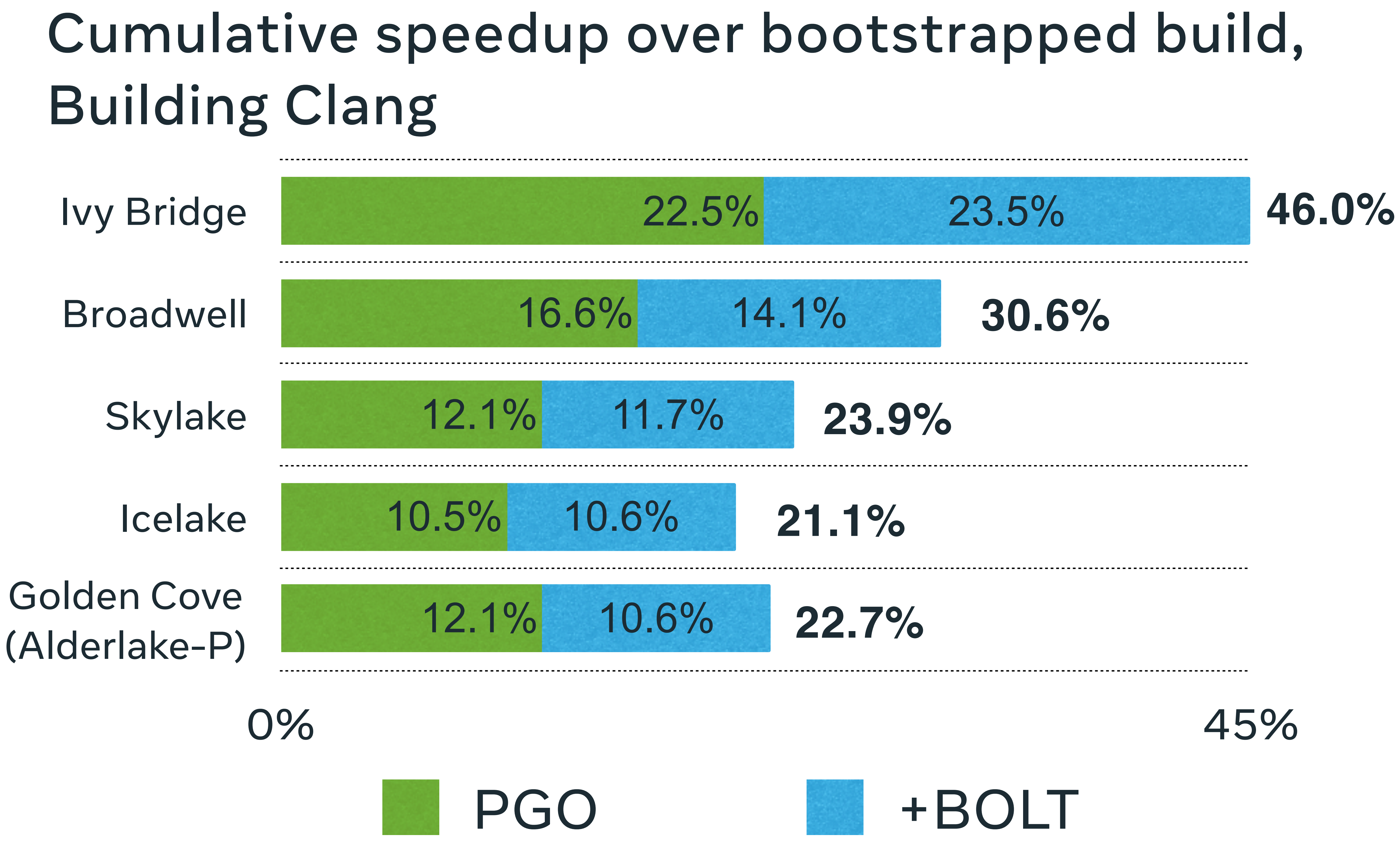
1. Why use BOLT?

Introduction

1. Why use BOLT?
    - Speedup on top of LTO and PGO

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO



2022 LLVM Dev Meeting: Optimizing Clang with BOLT using CMake

Introduction

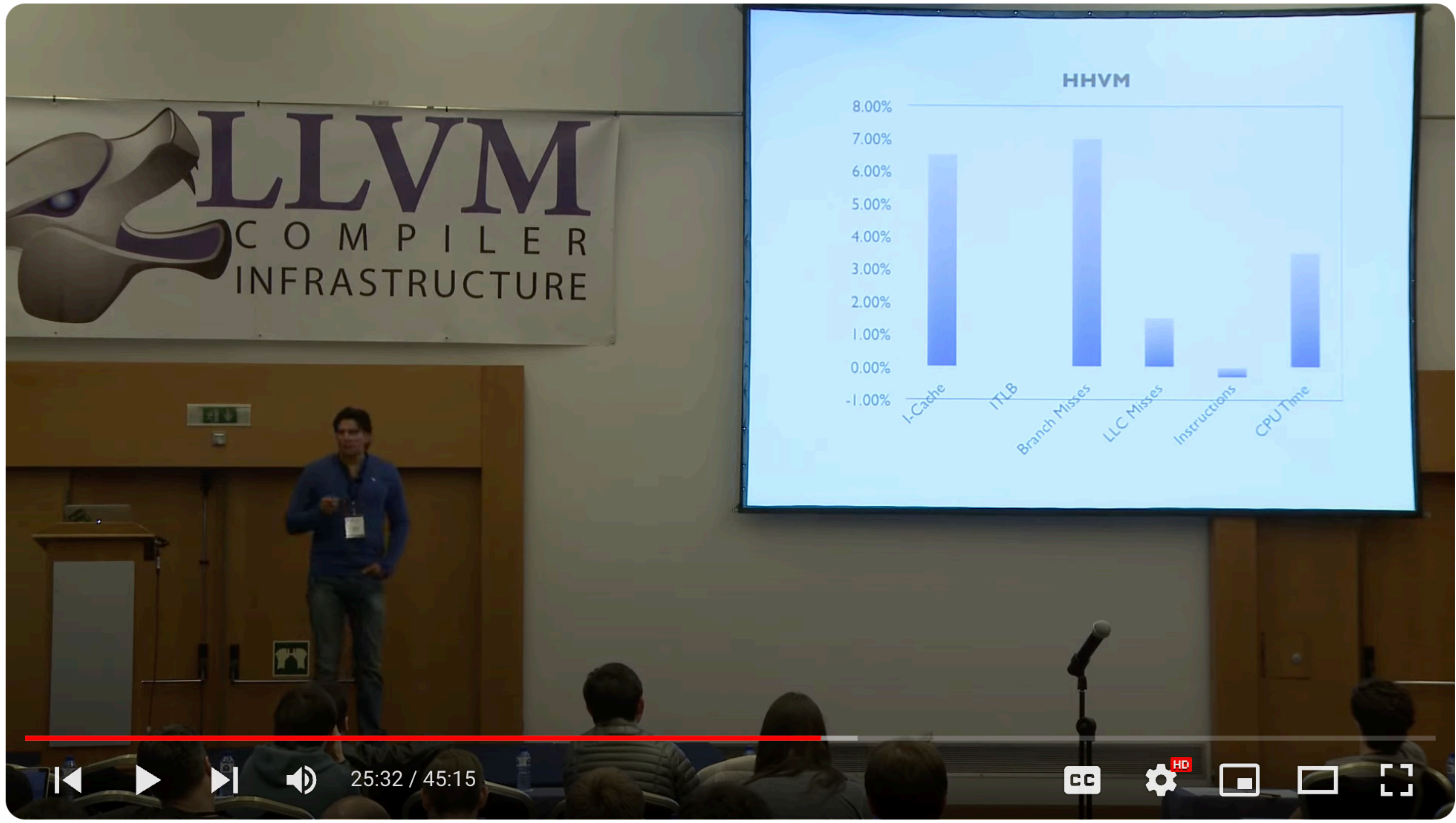
1. Why use BOLT?
    - Speedup on top of LTO and PGO
2. Showcases

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16

Introduction

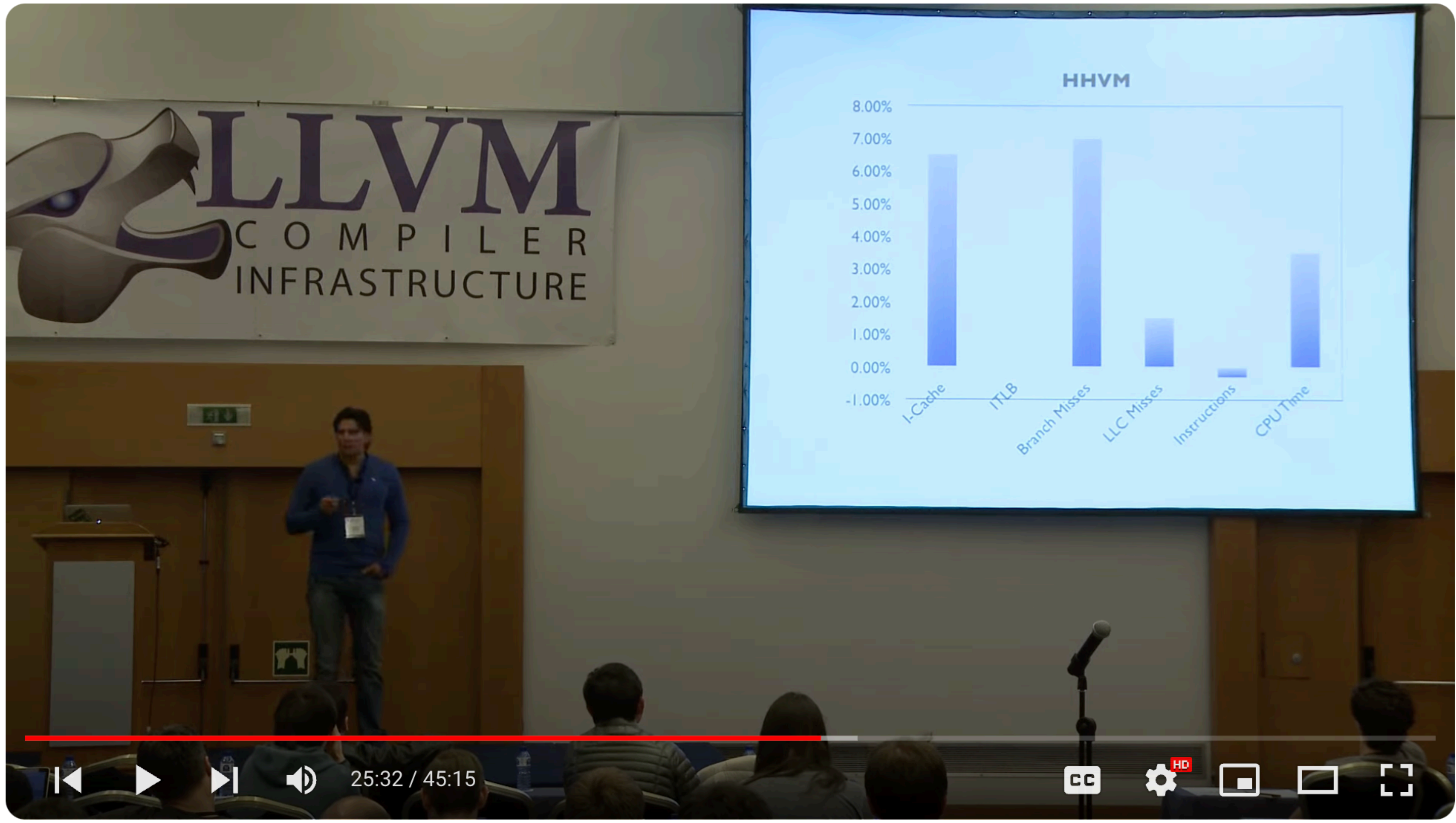
1. Why use BOLT?
    - Speedup on top of LTO and PGO
2. Showcases
    - Used at Meta: HHVM since ’16



2016 EuroLLVM Developers' Meeting: M. Panchenko "Building a binary optimizer with LLVM"

Introduction

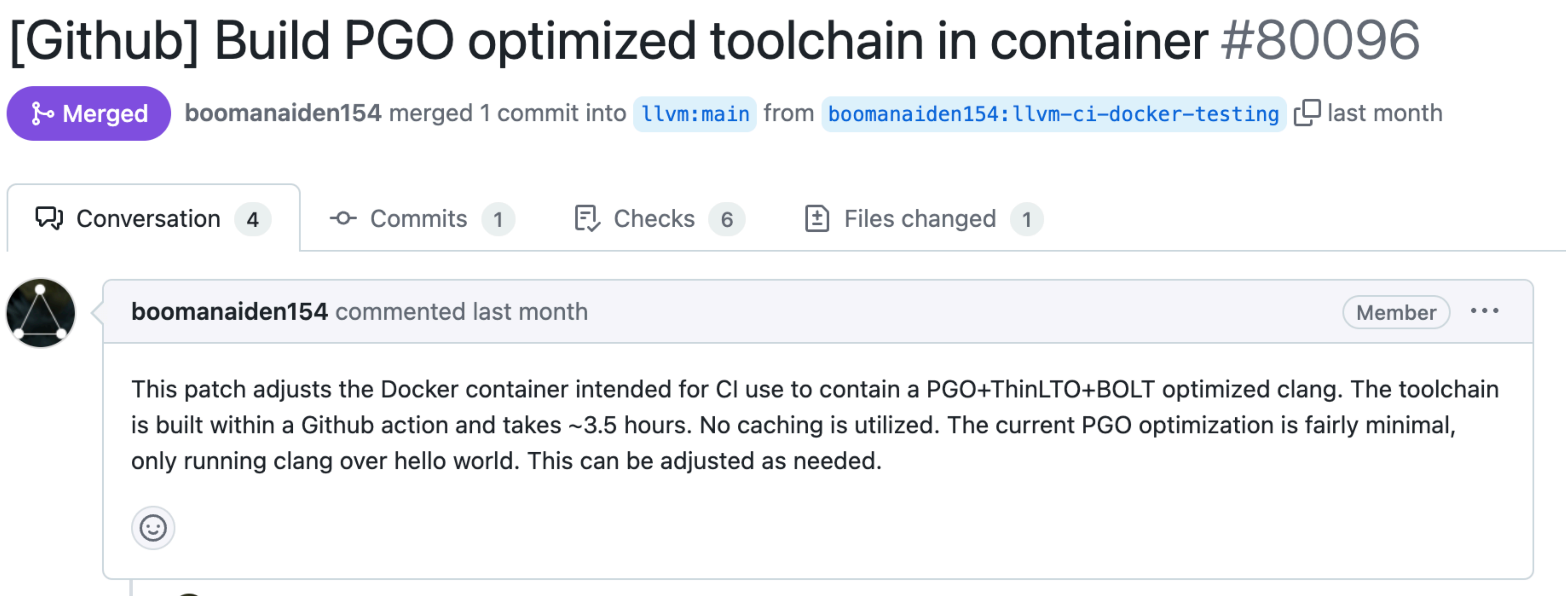
1. Why use BOLT?
    - Speedup on top of LTO and PGO
2. Showcases
    - Used at Meta: HHVM since ’16
    - Upstream adoption



2016 EuroLLVM Developers' Meeting: M. Panchenko "Building a binary optimizer with LLVM"

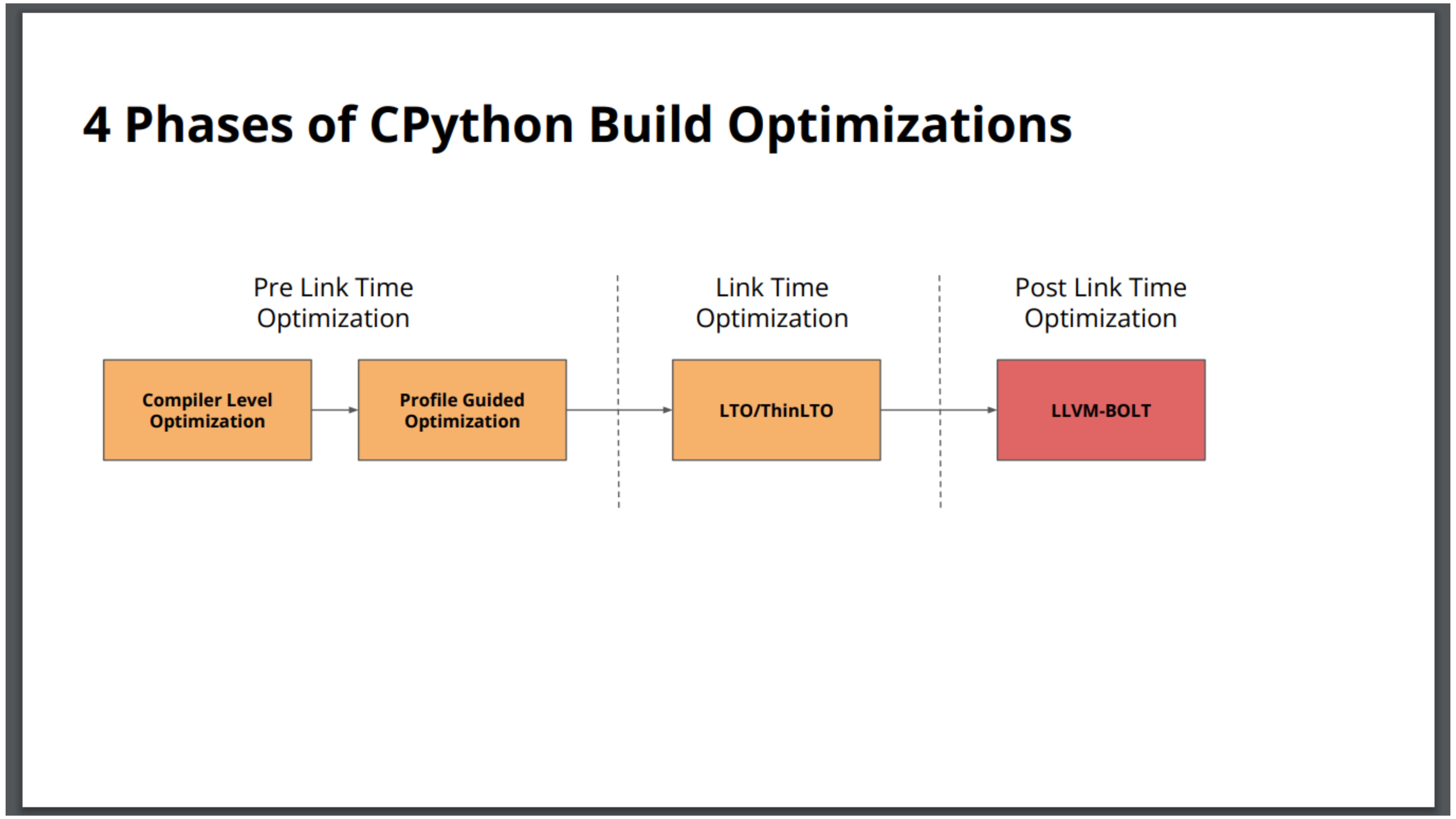
Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since '16
   - Upstream adoption



Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption



Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption

Code layout optimizations for rustc

The Rust compiler continues to get faster, with this release including the application of BOLT to our binary releases, bringing a 2% mean wall time improvements on our benchmarks. This tool optimizes the layout of the `librustc_driver.so` library containing most of the rustc code, allowing for better cache utilization.

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption

Faster Rust Toolchains for Android

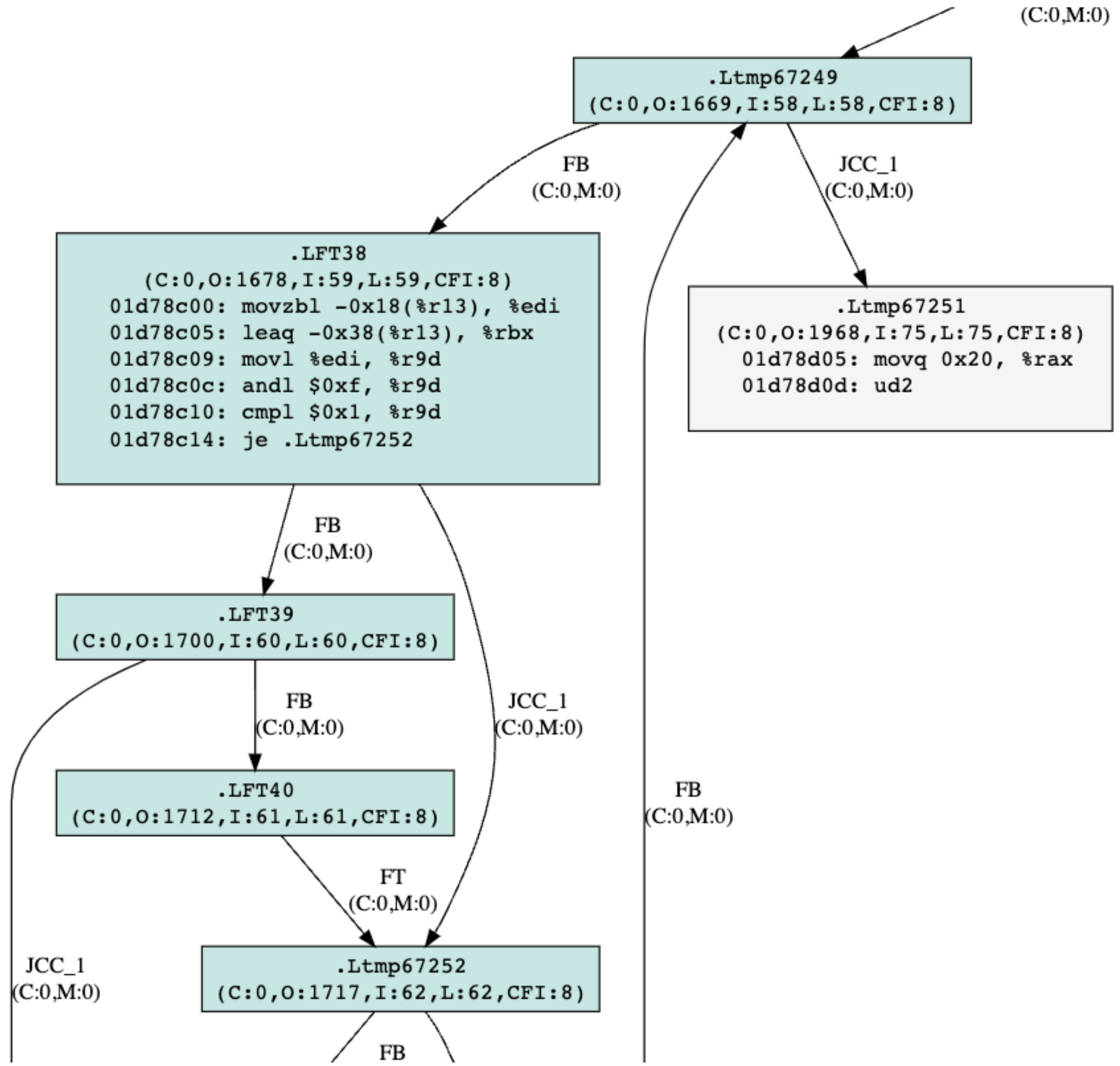
Posted by Chris Wailes - Senior Software Engineer

Introduction

1. Why use BOLT?
    - Speedup on top of LTO and PGO
2. Showcases
    - Used at Meta: HHVM since ’16
    - Upstream adoption
    - As a tool: disassembly with profile

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption
   - As a tool: disassembly with profile



Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption
   - As a tool: disassembly with profile
3. Where to use

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption
   - As a tool: disassembly with profile
3. Where to use
   - CPU frontend bound workloads

Introduction

1. Why use BOLT?
   - Speedup on top of LTO and PGO
2. Showcases
   - Used at Meta: HHVM since ’16
   - Upstream adoption
   - As a tool: disassembly with profile
3. Where to use
   - CPU frontend bound workloads
   - >5MB of code, >10% FE bound, >10 icache MPKI

Input binary prerequisites

Input binary prerequisites

1. Linux ELF X86 and AArch64
    - Experimental: RISC-V, MachO

Input binary prerequisites

1. Linux ELF X86 and AArch64
   - Experimental: RISC-V, MachO
2. Relocations: function reordering, bulk of BOLT's effect
   - `-Wl,--emit-relocs`

Input binary prerequisites

1. Linux ELF X86 and AArch64
   - Experimental: RISC-V, MachO
2. Relocations: function reordering, bulk of BOLT’s effect
   - `-Wl,--emit-relocs`
3. PLT optimizations: little extra
   - `-Wl,-znow`

Input binary prerequisites

1. Linux ELF X86 and AArch64
   - Experimental: RISC-V, MachO
2. Relocations: function reordering, bulk of BOLT's effect
   - `-Wl,--emit-relocs`
3. PLT optimizations: little extra
   - `-Wl,-znow`
4. Unsupported: stripped symbols + split functions (default in Linux distros)
   - GCC8+: disable `-freorder-blocks-and-partition`
   - LLVM: don't enable `-split-machine-functions`

Profiling

Profiling

1. **Use sampling with LBRs**
    - `perf record -e cycles` **`-j any,u`** `-- /bin/ls /`

Profiling

1. **Use sampling with LBRs**
   - `perf record -e cycles -j any,u -- /bin/ls /`
2. No LBRs:
   - Running in a VM, ARM: use BOLT instrumentation
   - ARM ETM: possible to convert into perf sample w/brstack format

Profiling

1. **Use sampling with LBRs**
   - `perf record -e cycles` **`-j any,u`** `-- /bin/ls /`
2. No LBRs:
   - Running in a VM, ARM: use BOLT instrumentation
   - ARM ETM: possible to convert into perf sample w/brstack format
3. Sampling requires longer profiling duration, but has ~0 overhead
   - Increase sampling frequency: `perf record` **`-Fmax`** …

Profiling

1. **Use sampling with LBRs**
   - `perf record -e cycles` **`-j any,u`** `-- /bin/ls /`
2. No LBRs:
   - Running in a VM, ARM: use BOLT instrumentation
   - ARM ETM: possible to convert into perf sample w/brstack format
3. Sampling requires longer profiling duration, but has ~0 overhead
   - Increase sampling frequency: `perf record` **`-Fmax`** …
4. Merging profiles: `merge-fdata`

Profiling

1. **Use sampling with LBRs**
   - `perf record -e cycles` **`-j any,u`** `-- /bin/ls /`
2. No LBRs:
   - Running in a VM, ARM: use BOLT instrumentation
   - ARM ETM: possible to convert into perf sample w/brstack format
3. Sampling requires longer profiling duration, but has ~0 overhead
   - Increase sampling frequency: `perf record` **`-Fmax`** …
4. Merging profiles: `merge-fdata`
5. Profile formats: YAML, fdata, perf.data, pre-aggregated

Profiling

1. **Use sampling with LBRs**
   - `perf record -e cycles -j any,u -- /bin/ls /`
2. No LBRs:
   - Running in a VM, ARM: use BOLT instrumentation
   - ARM ETM: possible to convert into perf sample w/brstack format
3. Sampling requires longer profiling duration, but has ~0 overhead
   - Increase sampling frequency: `perf record -Fmax …`
4. Merging profiles: `merge-fdata`
5. Profile formats: YAML, fdata, perf.data, pre-aggregated

Optimizations

Optimizations

1. State of the art:
   - Function splitting: `-split-functions -split-strategy=`**`cdsplit`**
   - Function reordering: `-reorder-functions=`**`cdsort`**
   - Block reordering: `-reorder-blocks=`**`ext-tsp`**

Optimizations

1. State of the art:
    - Function splitting: `-split-functions -split-strategy=`**`cdsplit`**
    - Function reordering: `-reorder-functions=`**`cdsort`**
    - Block reordering: `-reorder-blocks=`**`ext-tsp`**
2. Extra:
    - Use THP pages for hot text: `-hugify`
    - PLT optimization: `-plt`
    - More aggressive ICF: `-icf`
    - Indirect Call Promotion: `-indirect-call-promotion`
    - `--help`

Debug information

Debug information

1. Not updated by default: `-update-debug-sections`

Debug information

1. Not updated by default: `-update-debug-sections`
2. DWARF5 is supported

Debug information

1. Not updated by default: `-update-debug-sections`
2. DWARF5 is supported
3. Split DWARF is supported

Debug information

1. Not updated by default: `-update-debug-sections`
2. DWARF5 is supported
3. Split DWARF is supported
4. Can create accelerator tables (gdb_index, debug_names)

Reducing bloat

Reducing bloat

1. Reuse .text section: `-use-old-text`

Reducing bloat

1. Reuse .text section: `-use-old-text`
2. Disable hugify (aligns to 2MB)

Logging

Logging

1. Lots of useful information printed by default:
    - Profile quality: function coverage, profile staleness

Logging

1. Lots of useful information printed by default:
   - Profile quality: function coverage, profile staleness
2. Optimization effect based on dynamic instruction counts: `-dyno-stats`
   - Rule of thumb: 1B executed instructions

Logging

1. Lots of useful information printed by default:
   - Profile quality: function coverage, profile staleness
2. Optimization effect based on dynamic instruction counts: `-dyno-stats`
   - Rule of thumb: 1B executed instructions
3. Verbose logging if something is wrong: `-v=2`

Debugging

Debugging

1. Build in debug mode

Debugging

1. Build in debug mode
2. Debug logging if something goes terribly wrong: `-debug-only=bolt`
    - Beware: `-debug` enables everything
    - Find relevant file and LLVM_DEBUG

Debugging

1. Build in debug mode
2. Debug logging if something goes terribly wrong: `-debug-only=bolt`
   - Beware: `-debug` enables everything
   - Find relevant file and LLVM_DEBUG
3. Suspect function:
   - Read CFG: `-print-only=func.* -print-cfg`
   - Look at CFG: `-dump-dot-all`

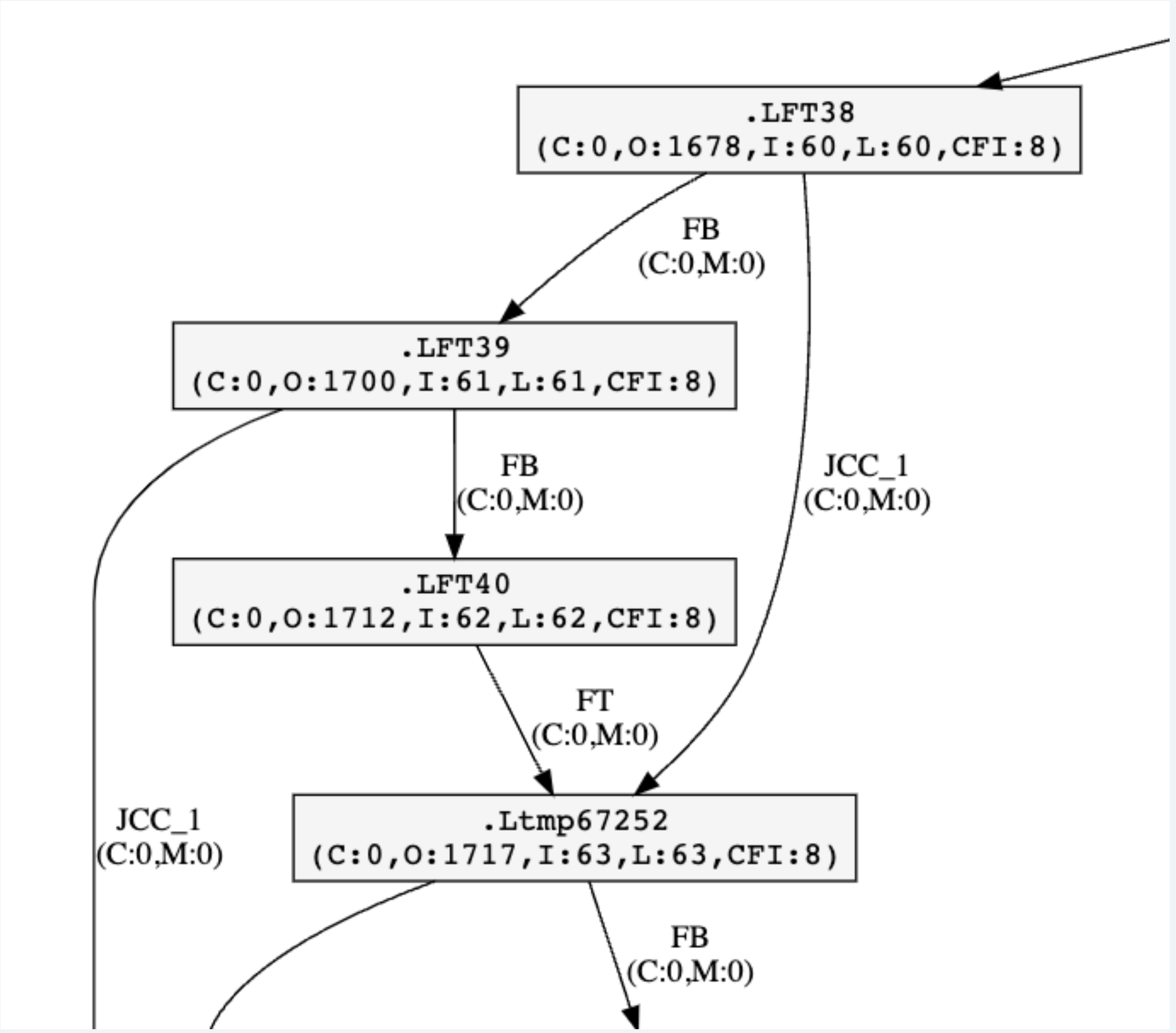
dot format

`llvm-bolt`

`  -dump-dot-all`

Outputs

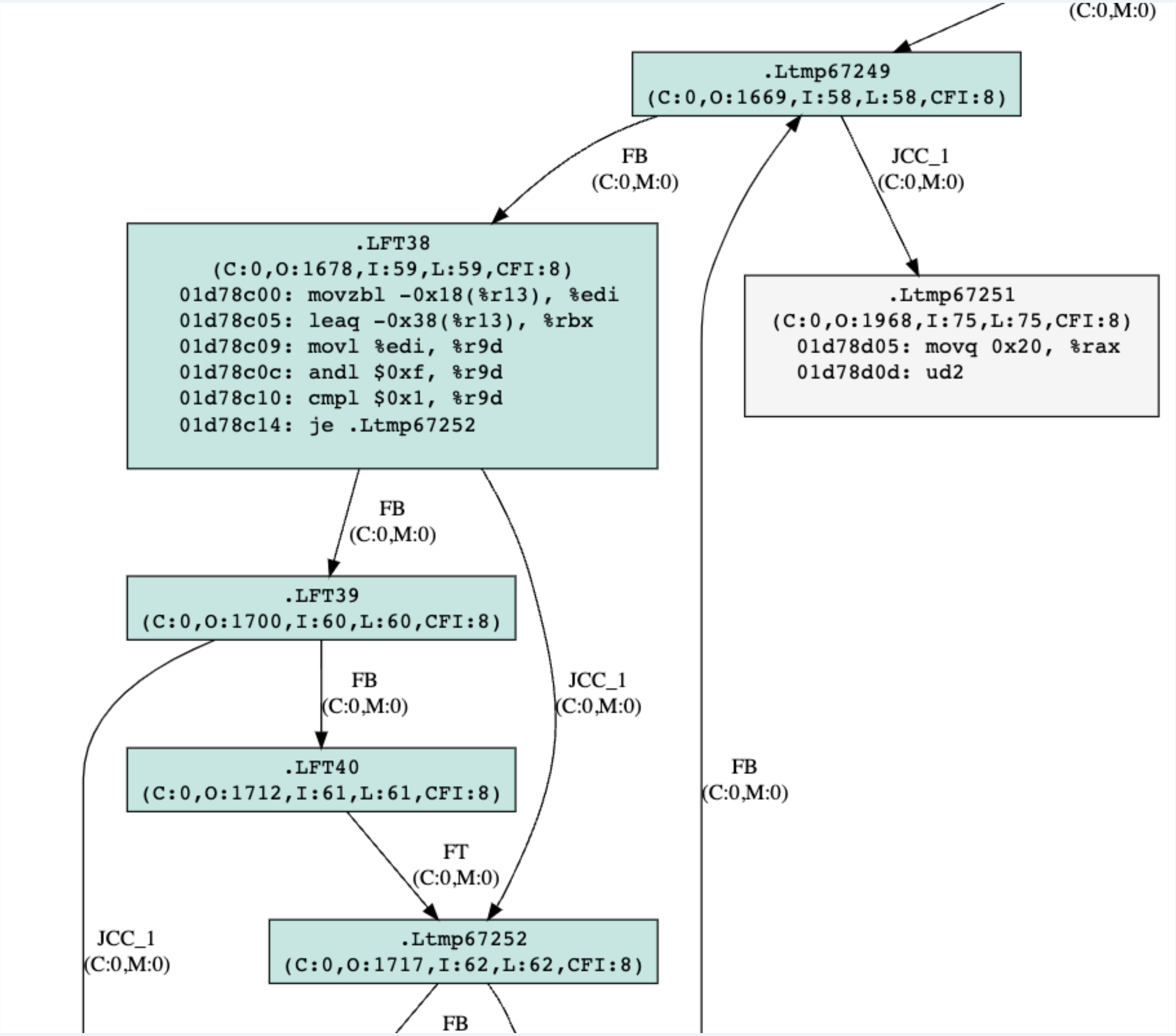
`funcname-00_build-cfg.dot`



Interactive HTML

```
llvm-bolt
  -dump-dot-all
  -print-loops -dot-tooltip-code

bolt/utils/dot2html/dot2html.py
  main-25_zero-idiom.dot{,.html}
```



Debugging

Debugging

1. Build in debug mode
2. Debug logging if something goes terribly wrong: `-debug-only=bolt`
    - Beware: `-debug` enables everything
    - Find relevant file and LLVM_DEBUG
3. Suspect function:
    - Read CFG: `-print-only=func.* -print-cfg`
    - Look at CFG: `-dump-dot-all`

Debugging

1. Build in debug mode
2. Debug logging if something goes terribly wrong: `-debug-only=bolt`
   - Beware: `-debug` enables everything
   - Find relevant file and LLVM_DEBUG
3. Suspect function:
   - Read CFG: `-print-only=func.* -print-cfg`
   - Look at CFG: `-dump-dot-all`
   - Skip it: `-skip-funcs=func.*`

Debugging

1. Build in debug mode
2. Debug logging if something goes terribly wrong: `-debug-only=bolt`
   - Beware: `-debug` enables everything
   - Find relevant file and LLVM_DEBUG
3. Suspect function:
   - Read CFG: `-print-only=func.* -print-cfg`
   - Look at CFG: `-dump-dot-all`
   - Skip it: `-skip-funcs=func.*`
4. Bughunter script

Bughunter script

Bughunter script

Bisecting to a function which causes a crash.

Pass the resulting function as

`--funcs=funcname`

to reproduce the issue.

Bughunter script

Bisecting to a function which causes a crash.

Pass the resulting function as

`--funcs=funcname`

to reproduce the issue.

`bolt/utils/bughunter.sh`

Invocation:

```
BOLT=/build/llvm-bolt \
BOLT_OPTIONS="-v=1" \
INPUT_BINARY=/path/to/binary \
# COMMAND_LINE="--version" or
# OFFLINE=1 \
bolt/utils/bughunter.sh
```

Output:

Text file containing the culprit function.

Performance debugging

Performance debugging

1. If BOLTed binary is slower
    - Check logs!
    - Profile is representative? Profile is correct?
    - Same binary used for profiling and optimization?
    - Noise?
    - Double-check stats?

Performance debugging

1. If BOLTed binary is slower
   - Check logs!
   - Profile is representative? Profile is correct?
   - Same binary used for profiling and optimization?
   - Noise?
   - Double-check stats?
2. If it’s really the case
   - Collect perf.data from BOLTed binary
   - Run `llvm-bolt-heatmap` and check layout

Interaction with PGO

Interaction with PGO

1. BOLT is a form of context-sensitive PGO
    - Not the only one: CSSPGO, CSIR PGO, FS-AFDO, Propeller
    - BOLT is 100% accurate and fast: no rebuilding or relinking
    - BOLT ❤️ PGO

PGO/BOLT pipeline

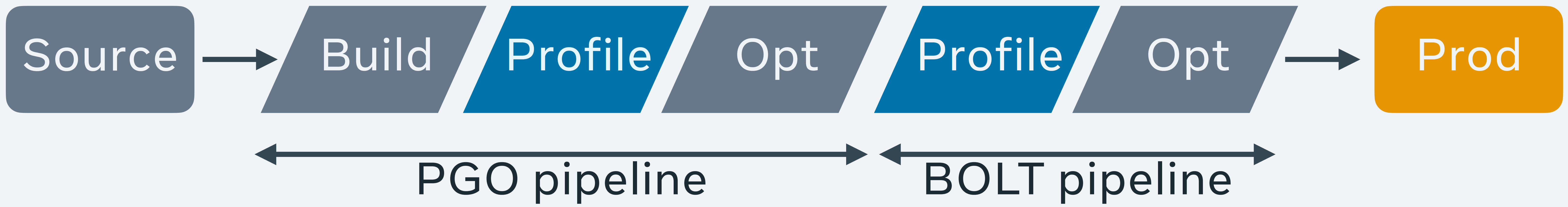
PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary

PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary

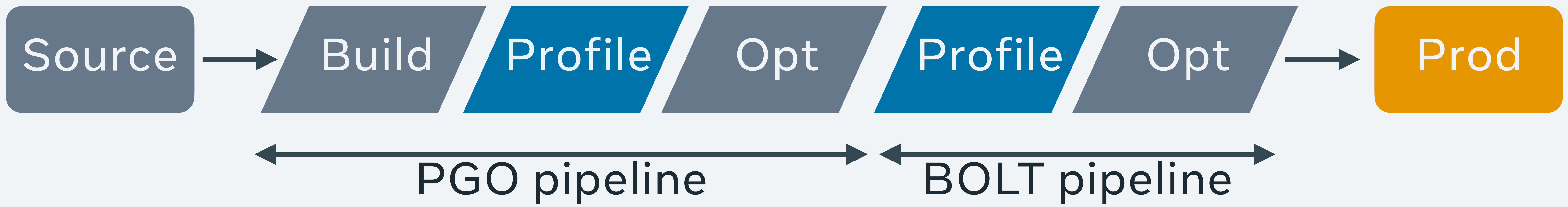
Ideal pipeline (“Zero Gap”)



PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary
- Compromise for CI/CD: “some gap”

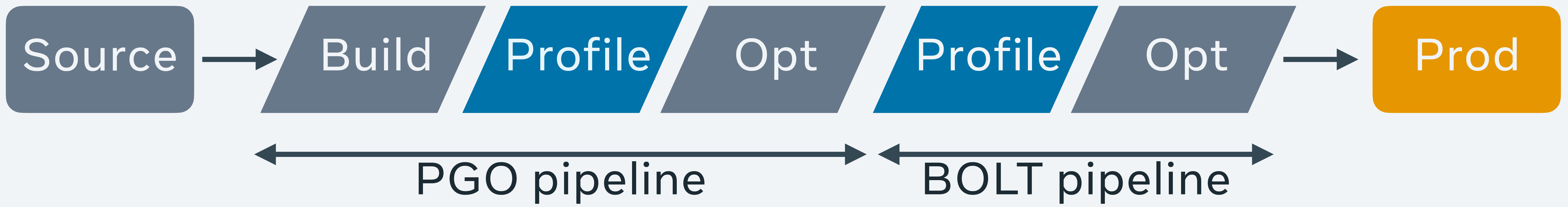
Ideal pipeline (“Zero Gap”)



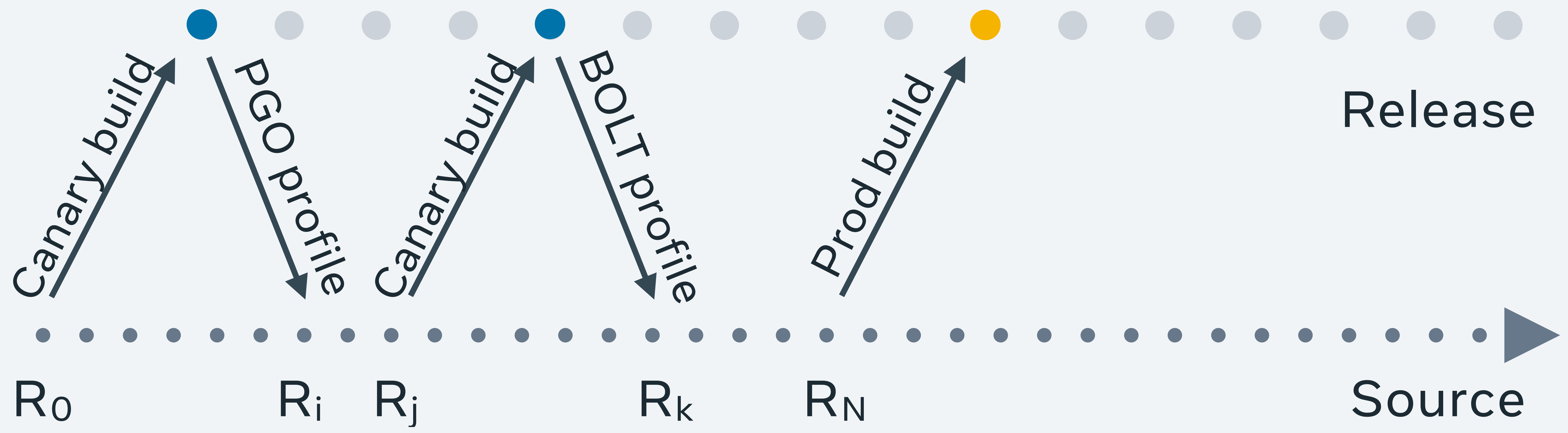
PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary
- Compromise for CI/CD: “some gap”

Ideal pipeline (“Zero Gap”)



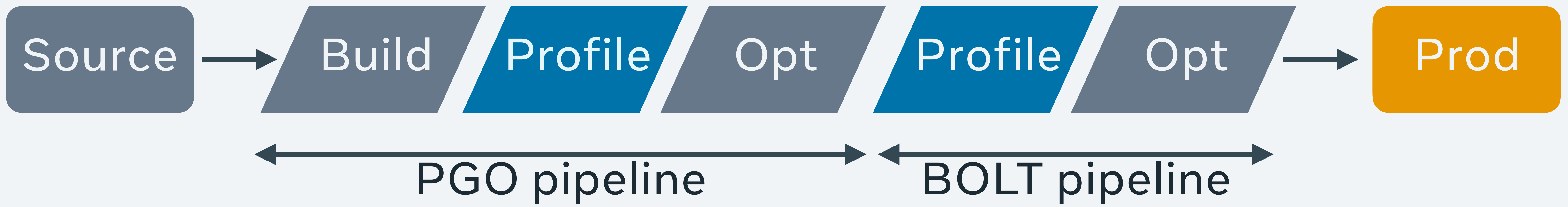
CI/CD + Continuous Profiling



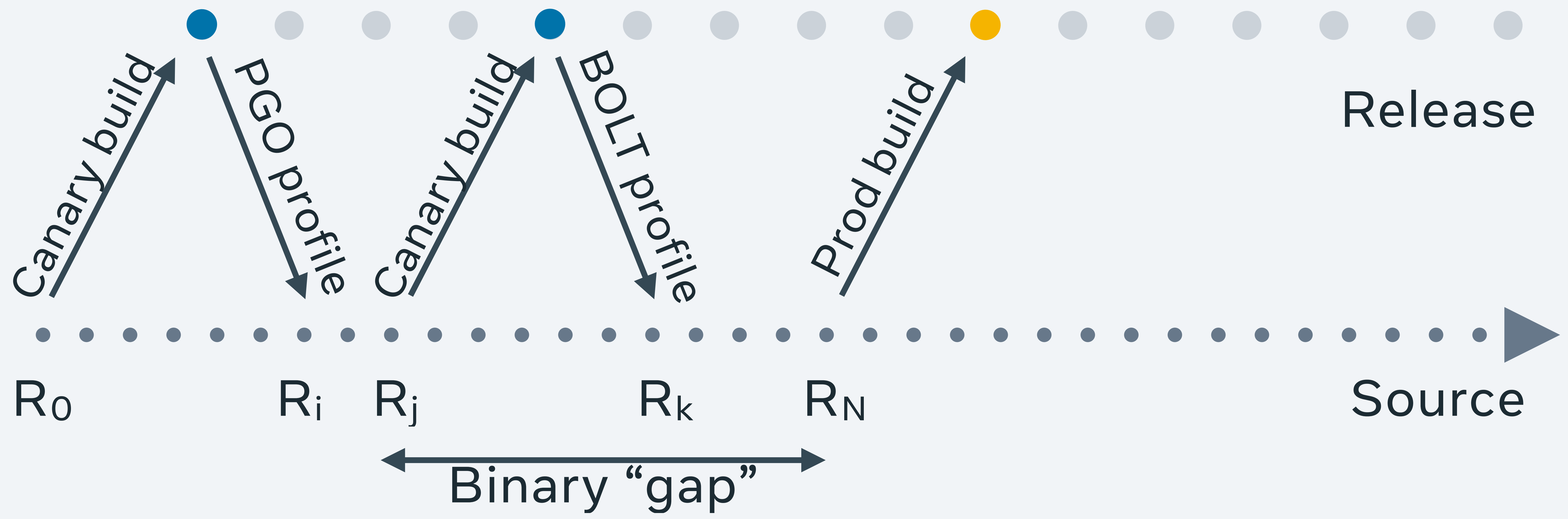
PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary
- Compromise for CI/CD: “some gap”

Ideal pipeline (“Zero Gap”)



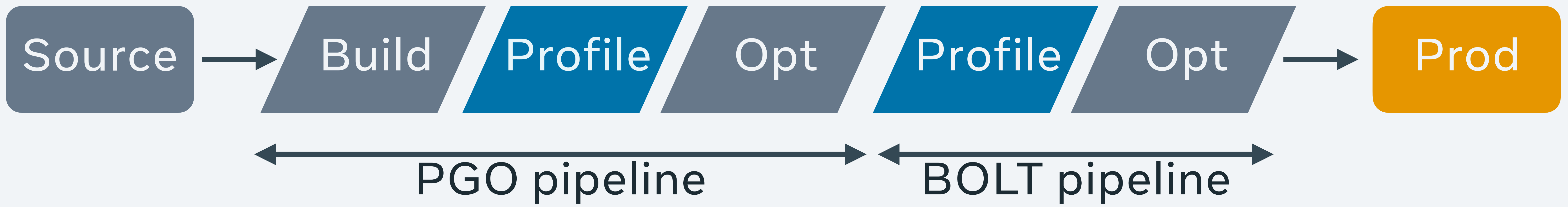
CI/CD + Continuous Profiling



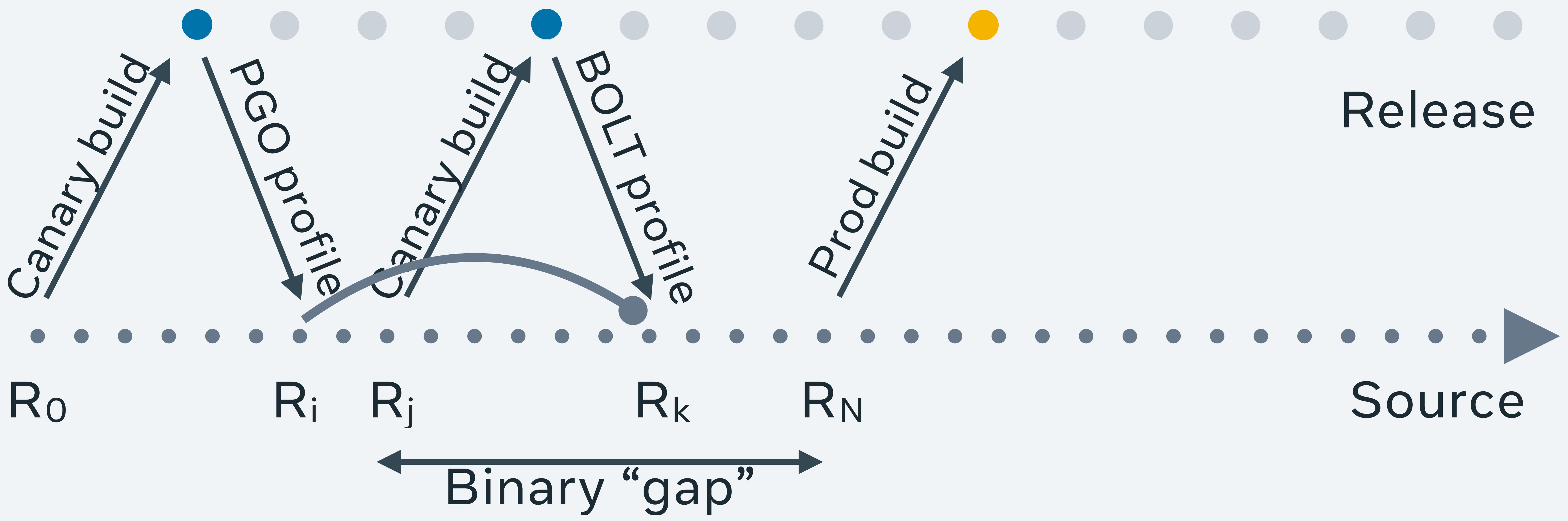
PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary
- Compromise for CI/CD: "some gap"
- BOLT-compatible PGO

Ideal pipeline ("Zero Gap")



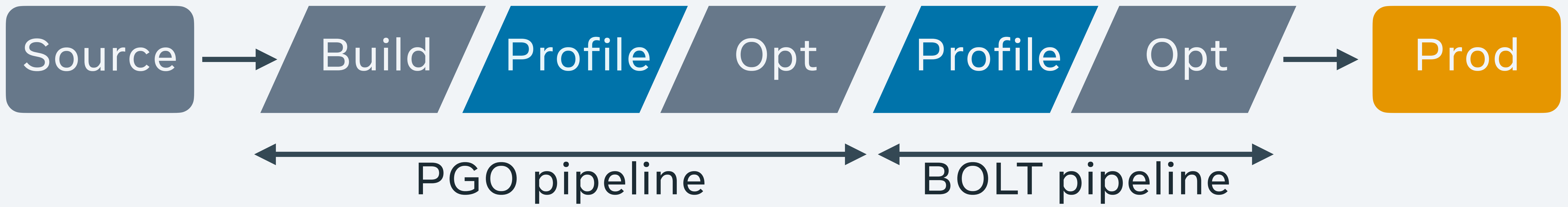
CI/CD + Continuous Profiling



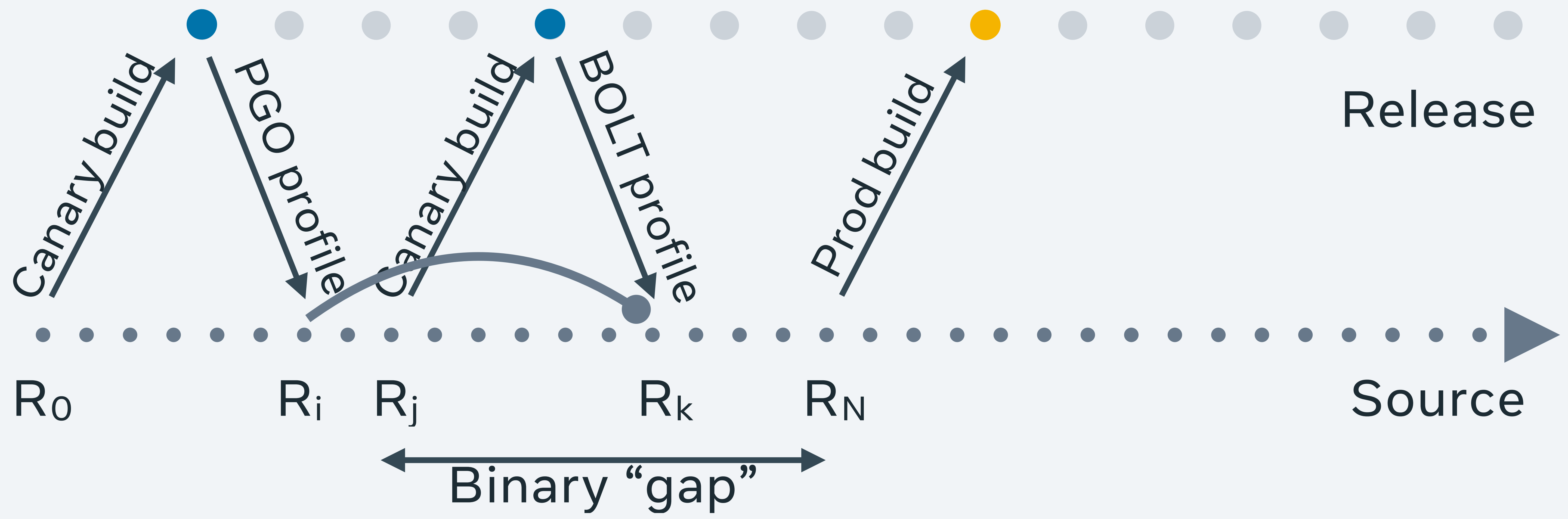
PGO/BOLT pipeline

- Max PGO/BOLT effect: profiled binary = optimized binary
- Compromise for CI/CD: “some gap”
- BOLT-compatible PGO
- PGO from BOLTed binary

Ideal pipeline (“Zero Gap”)



CI/CD + Continuous Profiling



Interaction with PGO

Interaction with PGO

1. BOLT is a form of context-sensitive PGO
   - Not the only one: CSSPGO, CSIR PGO, FS-AFDO, Propeller
   - BOLT is 100% accurate and fast: no rebuilding or relinking
   - BOLT ❤️ PGO

Interaction with PGO

1. BOLT is a form of context-sensitive PGO
    - Not the only one: CSSPGO, CSIR PGO, FS-AFDO, Propeller
    - BOLT is 100% accurate and fast: no rebuilding or relinking
    - BOLT ❤️ PGO
2. Dealing with non-zero binary gap:
    - `-infer-stale-profile` — *Stale Profile Matching, CC 2024*

Interaction with PGO

1. BOLT is a form of context-sensitive PGO
    - Not the only one: CSSPGO, CSIR PGO, FS-AFDO, Propeller
    - BOLT is 100% accurate and fast: no rebuilding or relinking
    - BOLT ❤️ PGO
2. Dealing with non-zero binary gap:
    - `-infer-stale-profile` — *Stale Profile Matching, CC 2024*
3. Collecting BOLT profile from BOLTed binary: `-enable-bat`
    - WIP streamlining use with stale matching

Meta