Profile-Guided Optimization (PGO) in Rust: practical guide

Alexander Zaitsev

Benchmarks

Application	**Improvement**	**Library**	**Improvement**
Rustc	up to +15% compilation speed	serde_json	~15% improvement
Vector	+15% EPS	xml-rs	~35% improvement
Rust Analyzer	+20% speedup	quick-xml	~25% improvement
PostgreSQL	up to +15% faster queries	tonic	~10% improvement
SQLite	up to 20% faster queries	rustls	~6% improvement
ClickHouse	up to +13% QPS	axum	~10% improvement
MySQL	up to +35% QPS	tantivy	~30% improvement
MongoDB	up to **2x** faster queries	wgpu	~25% improvement
Redis	up to +70% RPS	tracing libs	~35-40% improvement

Few words about me

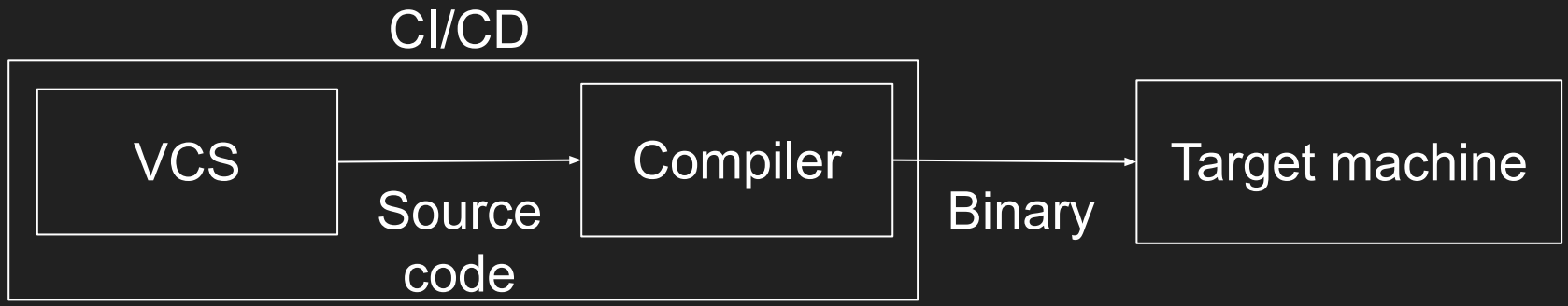
- Used to be a C++ engineer (now my C++ skills are Rust-y ;), later worked as an architect
- Was working with ISO WG21 (C++ committee): Numerics TS + constexpr containers stuff
- Spent several years on “hacking” LLVM compiler/static analyzers/C++ standard library (libc++), etc.
- Spent the last 2 years with PGO on a **daily** basis
- Now work as a Co-Founder/CTO @ Cytopus

Our plan for the talk

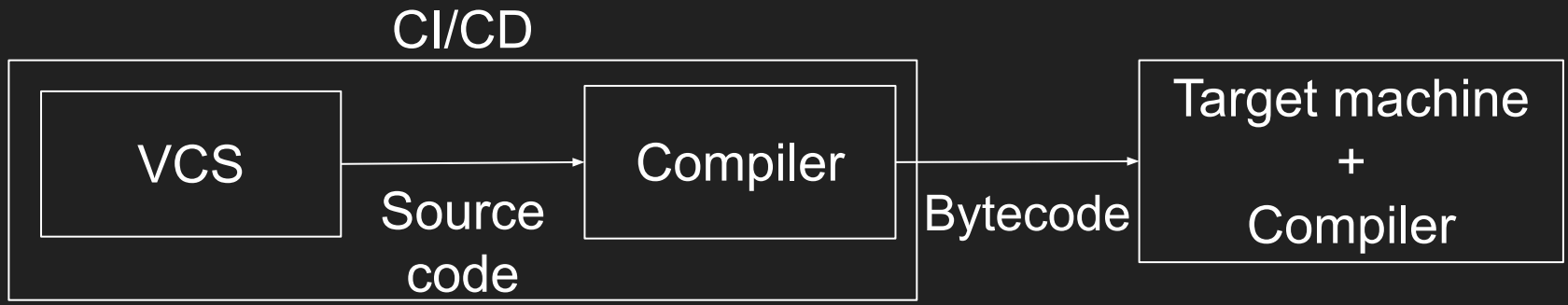
- A theoretical part about PGO
- PGO state in the Rust ecosystem
- Practical PGO problems (and sometimes solutions for them!)
- Few words about the things beyond PGO

A bit of theory

Ahead-of-Time (AOT)



Just-in-Time (JIT)



Compiler optimizations and runtime information

- **Inlining**
- Loop roll/unroll
- Devirtualization
- Hot/cold code splitting
- Link-Time Optimization (LTO)
- And many other funny things!

Many compiler optimizations can be improved by providing runtime execution statistics!

- **adce**: Aggressive Dead Code Elimination
- **always-inline**: Inliner for **always_inline** functions
- **argpromotion**: Promote 'by reference' arguments to scalars
- **block-placement**: Profile Guided Basic Block Placement
- **break-crit-edges**: Break critical edges in CFG
- **codegenprepare**: Optimize for code generation
- **constmerge**: Merge Duplicate Global Constants
- **dce**: Dead Code Elimination
- **deadargelim**: Dead Argument Elimination
- **dse**: Dead Store Elimination
- **function-attrs**: Deduce function attributes
- **globaldce**: Dead Global Elimination
- **globalopt**: Global Variable Optimizer
- **gvn**: Global Value Numbering
- **indvars**: Canonicalize Induction Variables
- **inline**: Function Integration/Inlining
- **instcombine**: Combine redundant instructions
- **aggressive-instcombine**: Combine expression patterns
- **internalize**: Internalize Global Symbols
- **ipsccp**: Interprocedural Sparse Conditional Constant Propagation
- **jump-threading**: Jump Threading
- **lcssa**: Loop-Closed SSA Form Pass
- **licm**: Loop Invariant Code Motion
- **loop-deletion**: Delete dead loops
- **loop-extract**: Extract loops into new functions
- **loop-reduce**: Loop Strength Reduction
- **loop-rotate**: Rotate Loops
- **loop-simplify**: Canonicalize natural loops
- **loop-unroll**: Unroll loops
- **lower-global-dtors**: Lower global destructors
- **lower-atomic**: Lower atomic intrinsics to non-atomic form
- **lower-invoke**: Lower invokes to calls, for unwindless code generators
- **lower-switch**: Lower **SwitchInst**s to branches
- **mem2reg**: Promote Memory to Register
- **memcpyopt**: MemCpy Optimization
- **mergefunc**: Merge Functions
- **mergereturn**: Unify function exit nodes
- **partial-inliner**: Partial Inliner
- **reassociate**: Reassociate expressions
- **rel-lookup-table-converter**: Relative lookup table converter
- **reg2mem**: Demote all values to stack slots
- **sroa**: Scalar Replacement of Aggregates
- **sccp**: Sparse Conditional Constant Propagation
- **simplifycfg**: Simplify the CFG
- **sink**: Code sinking
- **simple-loop-unswitch**: Unswitch loops
- **strip**: Strip all symbols from a module
- **strip-dead-debug-info**: Strip debug info for unused symbols
- **strip-dead-prototypes**: Strip Unused Function Prototypes
- **strip-nondebug**: Strip all symbols, except dbg symbols, from a module
- **tailcallelim**: Tail Call Elimination

Code | Blame | 3276 lines (2797 loc) · 124 KB

```
74  static cl::opt<int> InlineThreshold(
75      "inline-threshold", cl::Hidden, cl::init(225),
76      cl::desc("Control the amount of inlining to perform (default = 225)"));
77
78  static cl::opt<int> HintThreshold(
79      "inlinehint-threshold", cl::Hidden, cl::init(325),
80      cl::desc("Threshold for inlining functions with inline hint"));
81
82  static cl::opt<int>
83      ColdCallSiteThreshold("inline-cold-callsite-threshold", cl::Hidden,
84                            cl::init(45),
85                            cl::desc("Threshold for inlining cold callsites"));
86
```

The solution: Profile-Guided Optimization

- Collect runtime statistics on a target machine aka a **PGO profile**
- Pass the profile to a compiler
- Use the profile during the compilation phase

So many names… for the same thing

- **Profile-Guided Optimization (PGO)**
- Feedback-Driven Optimization (FDO)
- Profile-Directed Optimization (PDF)
- Profile-Based Optimization (PBO)
- Profile Online Guided Optimization (POGO)

PGO: optimization example


```
fn some_top_secret_checker(var: i32)
→ bool {
    if var == 42 { return true }
    if var == 322 { return true }
    if var == 1337 { return true }

    return false
}
```

```
    mov  al, 1
    cmp  edi, 42
    je   .LBB0_4
    cmp  edi, 1337
    je   .LBB0_4
    cmp  edi, 322
    je   .LBB0_4
    xor  eax, eax
.LBB0_4:
    ret
```

PGO: optimization example


```
fn some_top_secret_checker(var: i32)
→ bool {
    if var == 42 { return true }
    if var == 322 { return true }
    if var == 1337 { return true }

    return false
}
```

```
    mov  al, 1
    cmp  edi, 42
    je   .LBB0_4
    cmp  edi, 1337
    je   .LBB0_4
    cmp  edi, 322
    je   .LBB0_4
    xor  eax, eax
.LBB0_4:
    ret
```

```
    mov  al, 1
    cmp  edi, 322
    jne  .LBB0_1
.LBB0_4:
    ret
.LBB0_1:
    cmp  edi, 1337
    je   .LBB0_4
    cmp  edi, 42
    je   .LBB0_4
    xor  eax, eax
    jmp  .LBB0_4
```

PGO kinds

- Instrumentation PGO
- Sampling PGO
- Different flavours and combinations (CSIR PGO, CSS PGO, Temporal, etc.)

How does Instrumentation PGO work?

1. Compile your application in the Instrumentation mode
2. Run the instrumented application on your typical workload
3. Collect PGO profiles
4. Recompile your application once again with the PGO profiles
5. …
6. Profit!

PGO Instrumentation: assembly


```
fn is_meaning_of_life(var: i32) {
    var == 42
}
```

```
cmp     edi, 42
sete    al
ret
```

PGO Instrumentation: assembly


```
fn is_meaning_of_life(var: i32) {
    var == 42
}
```

```
cmp     edi, 42
sete    al
ret
```

```
inc     qword ptr [ ... ]
cmp     edi, 42
sete    al
ret
```

PGO Instrumentation: assembly


```
fn is_meaning_of_life(var: i32) {
    var == 42
}
```

```
cmp     edi, 42
sete    al
ret
```

```
inc     qword ptr [ ... ]
cmp     edi, 42
sete    al
ret
```

```
__llvm_profile_raw_version:
    .quad   72057594037927944
__llvm_profile_filename:
    .asciz  "default_%m.profraw"
```

Instrumentation PGO: caveats

- You need to compile your application at least twice: for instrumentation and then for the actual optimization
- An instrumented binary is larger
- An instrumented binary is **slower**

PGO instrumentation: binary size increase

Application	Release size	Instrumented size	Ratio
ClickHouse	2.0 Gib	2.8 Gib	1.4x
MongoDB	151 Mib	255 Mib	1.69x
SQLite	1.8 Mib	2.7 Mib	1.5x
Nginx	3.8 Mib	4.3 Mib	1.13x
curl	1.1 Mib	1.4 Mib	1.27x
Vector	198 Mib	286 Mib	1.44x
HAProxy	13 Mib	17 Mib	1.3x

PGO Instrumentation: binary slowdown

Application	**Instrumented to Release slowdown ratio**
ClickHouse	**311x**
Tarantool	1.5x
HAProxy	1.20x
Fluent-Bit	1.48x
Vector	14x
clang-tidy	2.28x
lld	6.8x

How does Sampling PGO work?

1. Run your usual application
2. Collect runtime information via an (external) profiler (like Linux perf or Intel VTune or any other profiler)
3. Recompile your application once again with runtime information
4. …
5. Profit!

Sampling PGO: caveats

- BSS hardware support can lead (and leads) to better results but can be unavailable in your hardware/OS:
  - Intel x86-64 - LBR: since Nehalem (2008), Linux - sometime between 2010-2011
  - AMD x86-64 - BRS: since Zen 3 (2020), Linux 5.19 (2022)
  - <vendor_name> ARM64 - BRBE: since ARMv9.2-A (2023), Linux 6.7-rc1 (2024) OR Coresight features (Android)
  - RISC-V - CTR. Status: proposal (WIP)
  - Does not work with virtualization

Sampling PGO: caveats

- BSS hardware support can lead (and leads) to better results but can be unavailable in your hardware/OS
- Limited tooling support
  - Google AutoFDO - buggy as hell
  - llvm-profgen - works only with BSS profiles

Instrumentation vs Sampling

- Instrumentation allows to achieve better optimization
  - According to Google: Sampling PGO has ~80-90% efficiency of Instrumentation PGO
- Sampling has far less runtime overhead
  - ~2% with Sampling compared to +inf with Instrumentation
  - You can even tweak the amount of overhead via a sampling rate
- Instrumentation has better OS and tooling support

PGO state in Rust

PGO support in Rust compilers

- **Rustc** - supports both instrumentation and sampling PGO
  - Same for **Ferrocene** compiler
- **gcc-rs** - No support
- **mrustc** - No support
- Other Rust compilers (if any) - I don’t know :)

PGO support in other ~~wrong~~ languages

- C, C++ - the maturest existing implementations in the world
- D, other GCC or LLVM-based compilers - almost the same as C++ but without some of the most advanced PGO features
- Go (the official compiler) - supports but can do little (for now)
- C# - supports, called “Dynamic PGO”
- GraalVM targets (Java, Kotlin and other) - supports, not enough publicly available information, bad docs :(
- Other languages/compilers - with 0.999(9) probability PGO is not supported

Instrumentation PGO in Rustc: an example

1. Compile the program with Instrumentation:

   rustc -Cprofile-generate=/tmp/pgo-data main.rs

2. Run the instrumented program with a training workload
3. Convert the .profraw file into a .profdata file using LLVM's **llvm-profdata** tool:

   llvm-profdata merge -output=merged.profdata default.profraw

4. Compile the program again with the profiling data:

   rustc -Cprofile-use=merged.profdata main.rs

cargo-pgo - the best PGO friend

- Written by Jakub “Kobzol” Beranek
- GitHub: https://github.com/Kobzol/cargo-pgo
- Supports Instrumentation PGO and LLVM BOLT

With this tool your PGO workflow could be something like this:

1. cargo pgo build
2. Run on a training workload
3. cargo pgo optimize build
4. …
5. Profit!

PGO issues in Rustc

- Documentation
- Tooling
  - cargo-pgo is not ideal
  - AutoFDO migration process
- Missing most advanced PGO modes
- Bugs
  - The most annoying one is about LTO + PGO (link)
- #[no_std] is not supported by default*
  - But can be achieved with minicov
- WASM is not supported (yet)

Current PGO states across Rust applications

- Almost no applications support building with PGO in their build scripts
- OS distributions also don’t build its software with PGO
  - However, some OS enables PGO for more packages like Gentoo, ClearLinux, **CachyOS**

Rule of thumb: if you want PGO for something - you need to rebuild it

ptr1337 02.11.2024, 12:21

Okay, got AutoFDO working in pkgbuild yippi

Doing now a super optimized AutoFDO + Propellor + ThinLTO Kernel

1 1

and when bolt someday works we can do a

AutoFDO + Propellor + ThinLTO + BOLT Kernel xD

and then everything explodes

PGO integration state for Rust apps

- Rustc - **PGO is enabled** (thanks, @Kobzol)!
  - Ferrocene - no PGO
- Rust Analyzer - no PGO
- Vector - no PGO (at least there is a page about PGO!)
- Quickwit - no PGO
- RedoxOS - no PGO
- Iggy-rs - no PGO
- Other Rust projects - I am pretty sure the same :(

You know what to do ;)

Hi Alexander,

There are various blockers that make PGO uninteresting at this point of time:

- From the feedback that we have, it is not the case that compilation times are a pressing issue for our customer base. This may change in the future.
- The targets that PGO supports in the upstream project are not the targets that we currently support: Our primary targets are aarch64 bare metal and QNX, while upstream PGO primarily targets Linux x64 and Windows.
- It adds significant complexity to the build process and the build process is part of the certification
- It makes reproducible builds extremely hard, making it harder for certification
- It significantly increases build times – and they're already a major issue both upstream and for us – we do essentially replicate all builds for our supported targets.
- There's an unknown risk of miscompilation – a major issue for any certification effort.
- Getting a reliable training set for the optimization is somewhat hard – it's not necessarily a valid assumption that the default set used by the upstream project matches what our customers intend to use.

Given all these open issues, the only option would be to support PGO for quality managed build only, and we currently see very limited interest in that. We don't even see interest in a faster build without assertions enabled, which would be a significantly easier step with a much lower risk.

Your PGO ~~traps~~ way in Rust

Q: How should I collect PGO profiles?

Short answer: it depends

Long answer: it depends (but longer)

- From unit-tests - please don’t!
- From (micro)benchmarks - it depends
- From manually-crafted training scenario - good option
- From production - great option (but please be careful)

There is no silver bullet here - it depends on your case

Q: How long should I collect PGO profiles?

Short answer: it depends

Long answer: it depends (but longer)

The question is not about **how long** you collect your PGO profile but how **representative** your PGO profile is.

Possible options:

- Collect from the whole workload - max representation but can be too expensive/time-consuming
- Collect from a part of the workload - can work as well but measurements are required anyway
- Any other options in between

Q: PGO optimizes some cases and pessimizes others

Short answer: you are unlucky

Long answer:

- Try to build per-workload application
- If it’s not possible, try to merge multiple profiles into one via tools like **llvm-profdata**

Q: PGO-optimized build is not reproducible!

Short answer: it’s a **complicated** topic

Long answer: oh…

There are two major cases:

- PGO-optimized build reproducibility - save somewhere a PGO profile and use it for all builds
- PGO profile reproducibility - ~~impossible~~ too difficult to achieve in practice for various reasons. More details - LLVM forum (click)

davidxl 10d

The value profile data depends up update order in the current implementation, so enabling atomic update does not guarantee full reproducibility. Disabling value profiling + atomic update may get it but this is not fully validated.

2 Replies

Reply

Xinliang David Li

David is a Principal Engineer at Google. He manages the Compiler Optimization Team and leads the effort to generate the fastest possible code for Google's data center softwares. His research interests include highly scalable cross module optimizations, scalable profile guided optimizations (instrumentation, PMU sample, trace-based), memory hierarchy optimizations (data and instructions), micro-architecture optimizations, post link optimizations, and ML based optimization frameworks.

Q: What to do with outdated PGO profiles?

Short: just regenerate them ;)

Long answer: regenerate them with some frequency:

- Each build - good for easy-to-gather PGO profiles
- Each release - good for most serious cases
- Continuous update - one of the best way but hard to achieve*

Continuous Profile-Guided Optimization

- AFAIK, the only thing right now is **Google Wide Profiler** (GWP) based solution, **closed-source**
- There is no ready-to-use open-source solution yet
- There is an idea about making such a platform as a part of **Grafana Pyroscope** or **Elasticsearch Universal Profiling**
  - Or even as a part of GCP Cloud Profiler
- Can be implemented on proprietary profiling platforms like Yandex.Perforator, Ozon.Vision, etc.

Q: Can I apply PGO to multilingua apps?

Short answer: yes

Long answer: yes but be careful:

- **Very** weak tooling support
- Incompatible PGO profile formats between different parts due to different compilers/compiler versions

Possible use cases: C/C++ + Rust apps, Rust + wrapper libraries (like Python via pyo3)

Q: What about PGO and constant-time computations?

Short answer: no worries - you are safe*

Long answer: the same guarantees as without PGO - it depends on your implementation:

- If you have a robust compiler optimization independent mechanism - you are fine
- If not - well… be careful

Q: Can I implement PGO manually in my code?

Short answer: yes, you can

Long answer: please do it only if you **really** need it

Manual implementation has the following disadvantages:

- Require more human resources
- Harder to maintain
- Isn’t flexible across workloads
- Could be difficult to implement all PGO optimizations

Please spend your time on something more useful like algorithmic optimizations

“Better algo always beats optimizer” - it depends ;)

rgerhards commented on Jun 28, 2023 Member

I would tend to say that this advise is best for distro package maintainers. While we build packages, the far majority of folks use the distro provided ones. I admit I am conservative on build tools. For example, we tried jmalloc in the past. Good performance, but we found a couple of definite jmalloc bugs, probably fixed now, but we decided to keep the old allocator in favor of robustness.

Everyone is free to rebuild. Build toolchain is far from our core competency, the team is small and as you can see there is a lot of work. While I appreciate the idea, I have to say that there are far more important things in front of it in the pipeline (think: better algo always beats optimizer).

Q: Why PGO doesn’t optimize my application?

Short answer: your software is written by optimization Gods

Long answer: you are lucky enough

- Unfortunately, not all software is so heavily optimized like FFmpeg/Symphonia
- Even if for now optimizations are good, later less optimized code can be introduced - and PGO would be a viable option
- If you are really sure that compiler should be able to optimize your program with PGO but it doesn’t happen - you can report to the upstream (GL&HF with that)

Optimization roadmap for a blazing fast app

- Enable **opt-level = 3**
- Enable **LTO** (Fat or Thin)
  - Probably **codegen-units = 1**
- Enable **PGO**
  - Instrumentation by default
  - Sampling otherwise
- Enable **Post-Link Optimization (PLO)**
- Enable more advanced things O_o

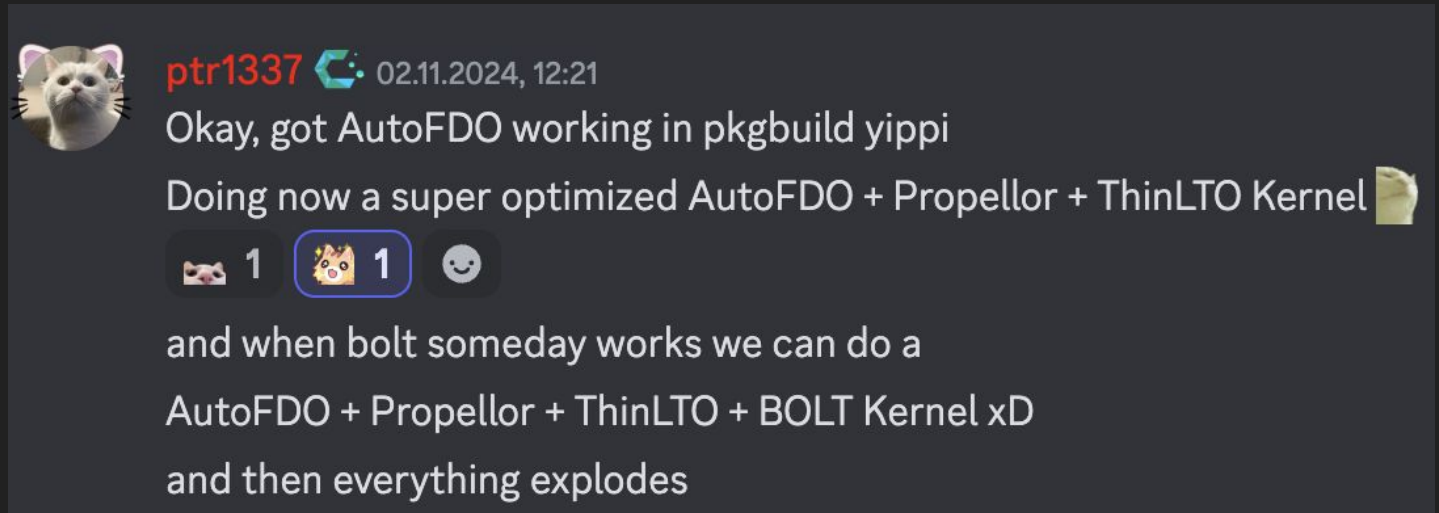
Post-Link Optimization (PLO)

- An optimization technique that optimizes a binary layout to minimize CPU instruction cache misses
- The default tool nowadays: LLVM BOLT (developed by Facebook)
- Less known tools: Google Propeller and Intel TLO
- Workflow is almost the same as PGO: instrument, train, optimize
  - Sampling is supported too (at least by BOLT)
- Their own issues: limited architecture support, BUGS, big instrumentation overhead, etc.

Post-Link Optimization (PLO)

- PGO does not implement **some** optimizations from PLO, and vice versa
- PLO is not a PGO substitution - it's an addition!
- So your final optimization pipeline can look something like this:

Release + LTO + PGO + PLO



Benchmarks: PGO vs PLO vs PGO + PLO

The benchmark - SQLite with the “speedtest” bench suite:

- Release (baseline): 17.022s
- PGO: 15.195s
- PLO: 15.887s
- PGO + PLO: 14.996s

More advanced things?

I have a bunch of them too ;)

- Application-Specific Operating Systems (ASOS)
- Or even Application-Specific Interpreters (I did this too, huh)
- Machine-Learning based compilers
- Maaaaaaaany other more academic things like OCOLOS and HALO

Good optimization candidates

- Compilers, interpreters and similar things like static analyzers, LSP servers, code formatters, etc.
- Databases
- Different parsers (like log solutions, XML/JSON/Protobuf, etc.)
- Operating systems, drivers
- Any mentioned at this conference software
- Actually, any application with many code branches
- <placeholder_for_your_application>

Links

There are so many links - cannot list them all here. But the main meta-link is…

Awesome PGO

The ideas behind Awesome PGO project

1. Introduce more **data-driven** optimization approaches for **everyone** - PGO is just a small step in this direction
   a. 500+ PGO issues on GitHub and ~250 PGO benchmarks for **different** projects is a small price for that :)
2. Increase the **default** performance level of the **whole** industry for various reasons: I like quick software, lower carbon emissions, better UX, cheaper infrastructure/smaller Total Cost of Ownership (TCO), easier to achieve non-functional requirements (NFRs), achieve your quarter performance goals (lol), etc. -> **improve the world**

That’s it for today! Thank you!

The PGO Gate
https://github.com/zamazan4ik/awesome-pgo

- **Emails**:
  - zamazan4ik@tut.by (primary)
  - zamazan4ik@gmail.com (secondary)
- **Telegram**: zamazan4ik
- **Discord**: zamazan4ik
- **Reddit**: zamazan4ik
- **GitHub**: zamazan4ik