MLGO

A Machine Learning Guided Compiler Optimizations Framework

Trofin, Mircea, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li (2021)

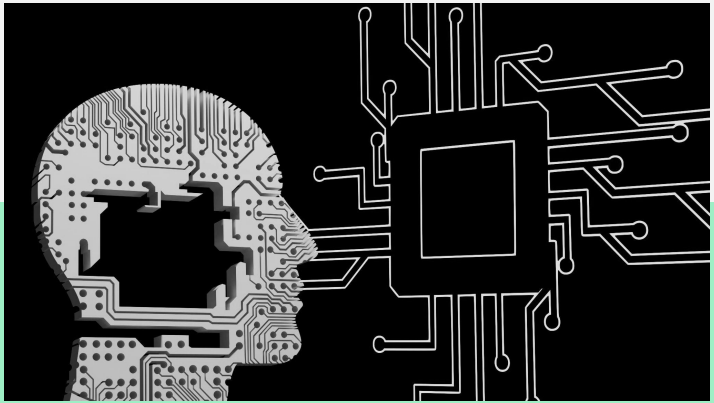
Group 3
Xuweiyi Chen
Jiaming Zheng
Congming Liao
Zin Hu

01

Introduction

Motivation

Replacing compiler optimization heuristics with machine-learned policies.



- Heuristics are human-trained.

  ML easily scales up to large corpora of training sets.

- Heuristics are human-written code that needs to be maintained, # features limited.

  ML leverages more features.

General Approach

Reinforcement learning

Evolution Strategies (ES)

Policy Gradient (PG)

Heuristics Based Optimization

Reduce code size with **Inlining**

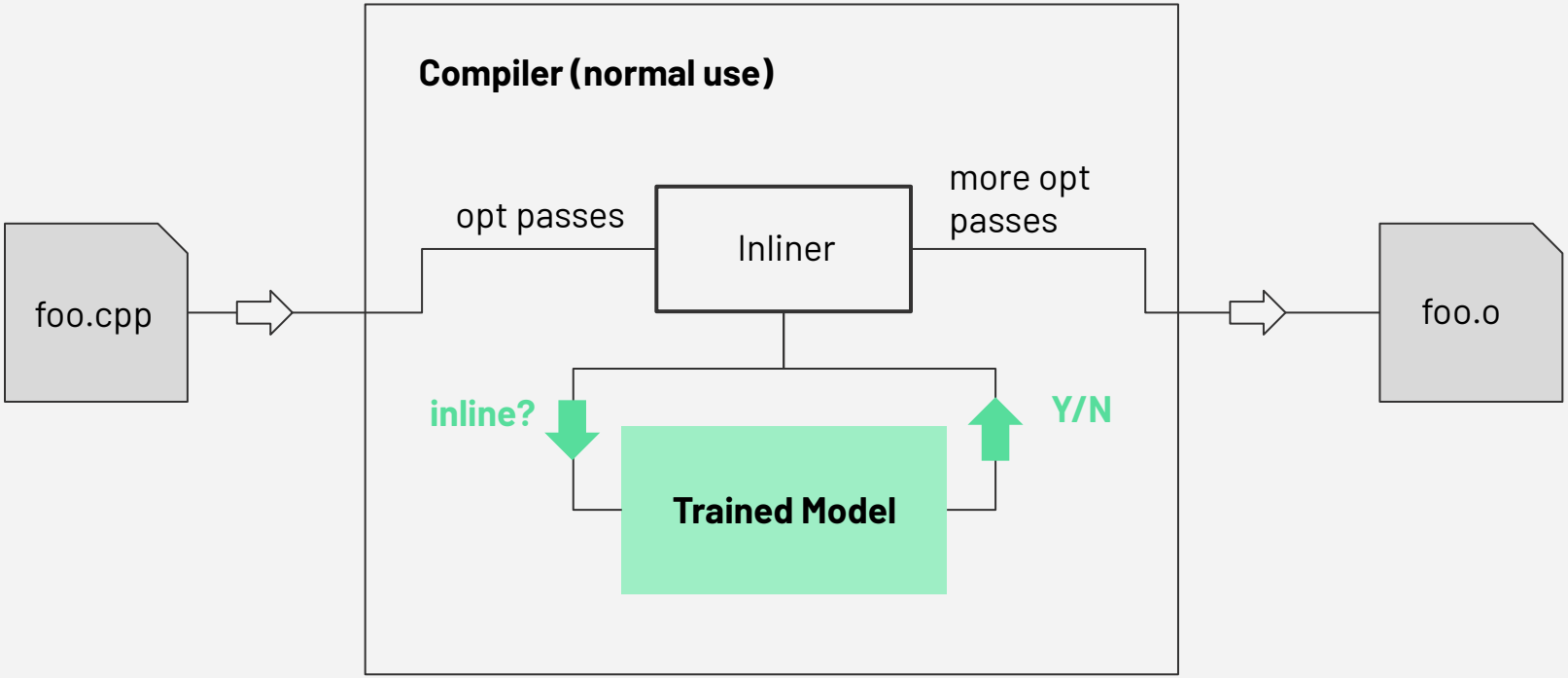
Improve performance with **Register Allocation**

Why Reinforcement Learning?

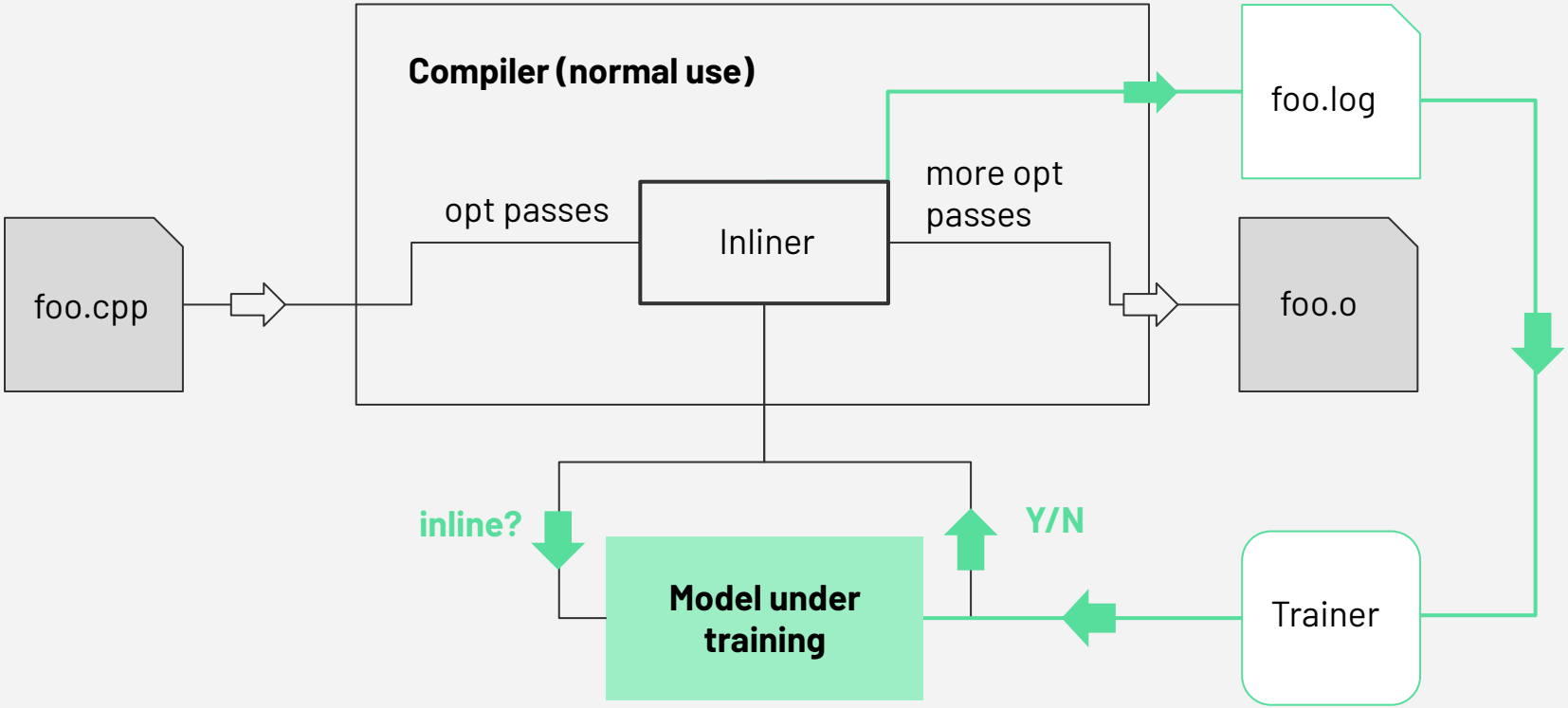
Lack of ground truth

Explore various strategies and improve strategies from experience

MLGO Overview (normal use)



MLGO Overview (training)



RL Policy Training

The current call graph and the call site being visited to be the state

$A = \{0, 1\}$, where 1 means inline and 0 means not inline

After action taken, compiler determines what the next state is.

$$S(Caller_{before}) - S(Caller_{after}) + \mathbb{I}(Callee_{deleted}) \cdot S(Calless)$$

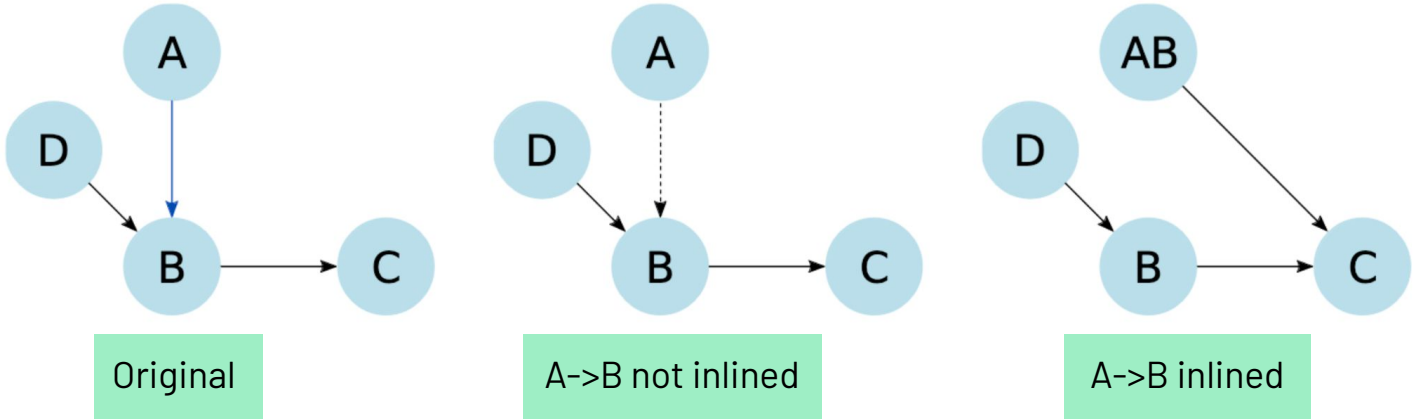
where $S(f)$ is the native size of function f.

02

LLVM Inliner

LLVM Inliner

- A pass operating on a strongly-connected component(SCC) of **static call graph**.
- The inlined callee's **call sites** are **added** to a work list and **iteratively** considered **for inlining** in a top down fashion.
- Perform a series of **optimizations on the SCC**, and these optimizations also influence the decision of inlining or not.



Theodoridis, T., Grosser, T., & Su, Z. (2022). Understanding and exploiting optimal function inlining. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3503222.3507744

General LLVM Heuristic Steps

01

Computes the **static** "**cost**" of the callee post inlining.

If shown the some call sites are constant during compiling, that information is used to confirm whether inline or not.

02

The **cost** is then compared with **threshold**.

Call site hotness, inline keywords, especially callee with a single basic block.

03

Decide inline or not with step 2 comparison.

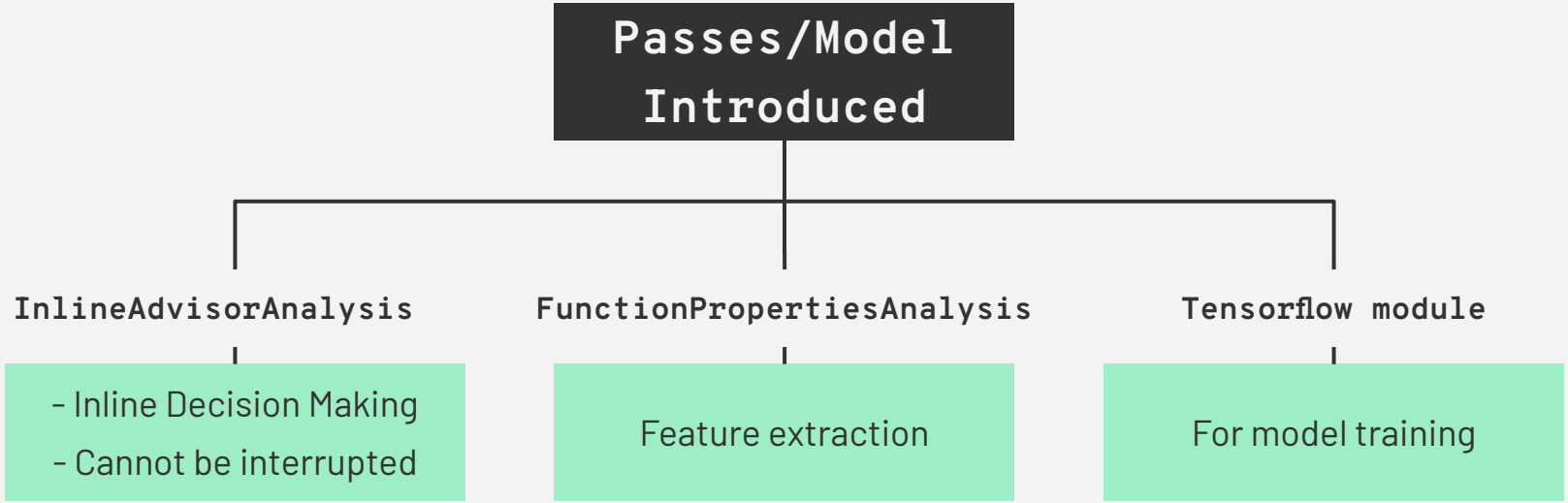
Note that inline can be delayed if inlining the caller itself to its own caller would result in better saving.

RL-driven inlining in LLVM

- **Challenge**: too complex state space. Encoding the call graph at each decision point would not be possible. Instead, come up with a self-designed feature space. Table→
- **Drawback**: greatly reduces info, limited global and local call site info. Won't hurt: same info available to current inlining heuristics.
- **Side-step** lack of partial reward: Evaluate native size with/without inlining (total R) .

Type	Features
caller feature	caller_basic_block_count caller_conditionally_executed_blocks caller_users
callee feature	callee_basic_block_count callee_conditionally_executed_blocks callee_users
Call site feature	callsite_height cost_estimate number_constant_params
Call graph feature	edge_count node_count

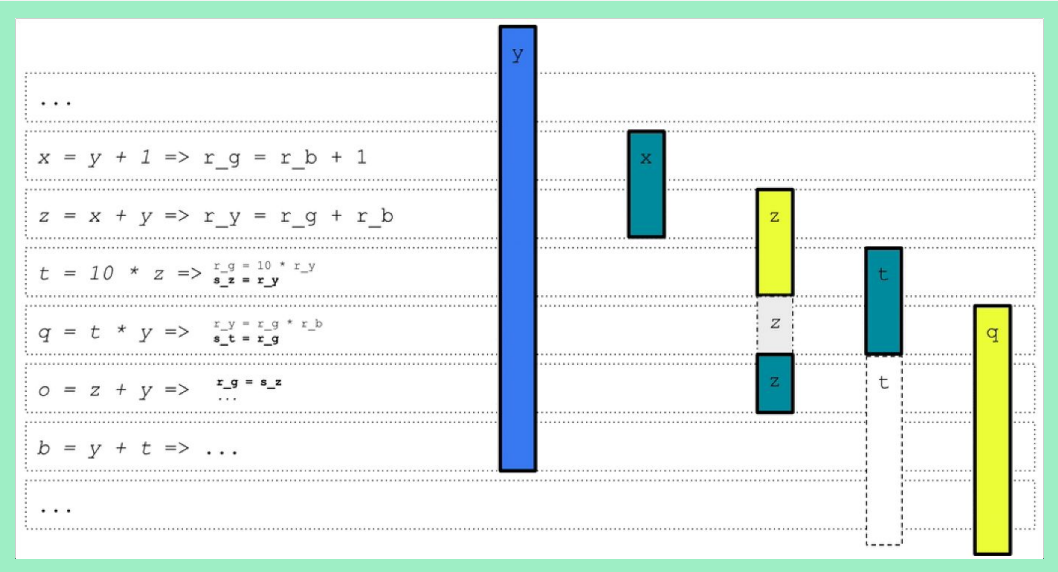
LLVM implementation



03

Register Allocation

Register Allocation



Register Allocation pass

make decisions to make room for q.

Extra load/store

degrades performance.

MLGO Regalloc

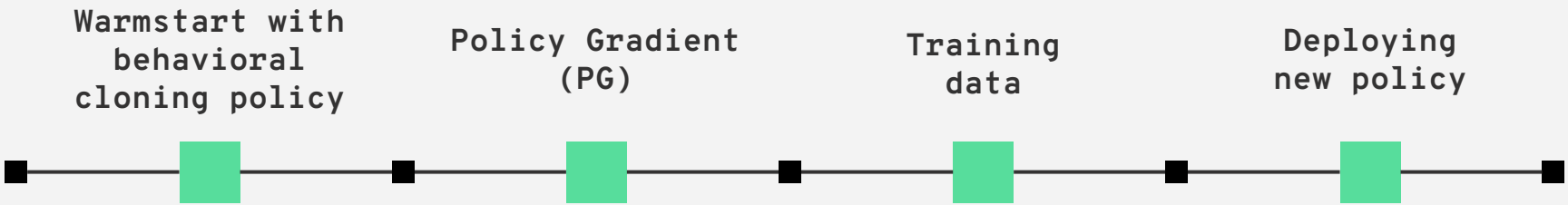
Heuristics

The problem of optimal register allocation is NP-complete. For n>2: Graph color Coalescing

MLGO

Aims at learning eviction policy for register allocation.

Regalloc Policy Training



Train the model using behavioral cloning on the default heuristic.

computing the gradients w.r.t. the reward function

Internal Google code base from the Chromium.

Tested on the code of Fuchsia — OS designed to power diverse hardware and software.

Evaluating the Performance

Benchmark

- More time consuming.
- More noisy than size measurements.

Reward Metrics

- Rewards: the block freq-weighted sum of introduced moves per function
- Using performance counters to track instructions executed, loads, and stores

04

Evaluation

Compared to state of the art
LLVM heuristic-driven Oz

Size Reduction

7%

Good

Generalizability

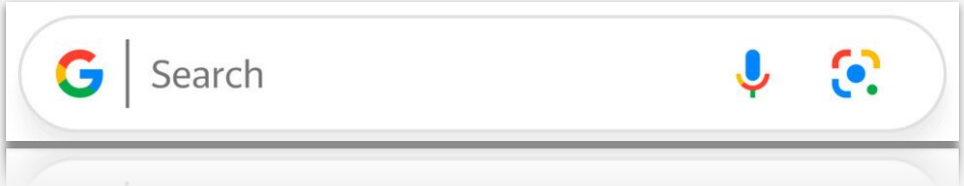
Generalize well to a diversity of real-world targets, + after months of active development

Inlining for Size Results

	PG	ES	ES (L)
Size Reduction	4.95%	3.74%	5.94%
Parallelism in Data Collection	100	488	488
Training Time	~12h	~60h	~150h

Table 2. Policy Gradient v.s. Evolution Strategies

Trained the policy for Google search (over 28,000 IR modules)



Efficiency

PG > ES
PG consumes ~4% training resources of ES

Performance

Better policy with larger NN
Cost: ↑ training resources

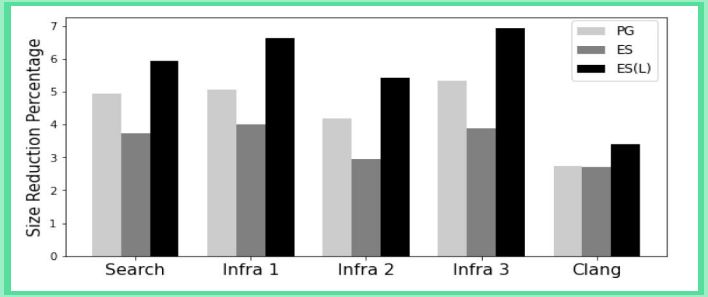
Generalizability across Software

Comparison

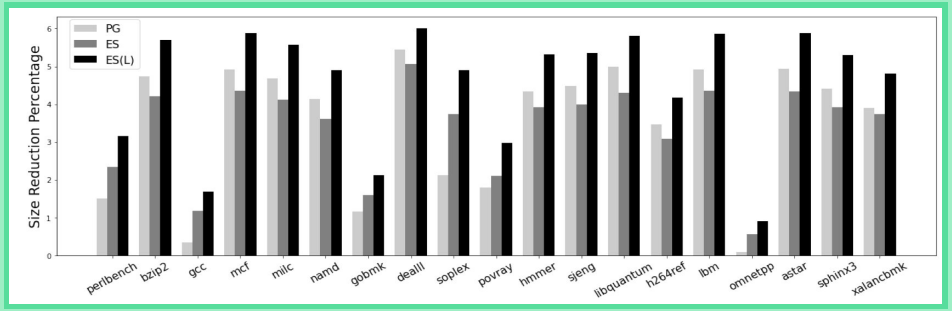
ES(L) > PG > ES

Consistency

A policy performs better on a certain software also perform better on other software.



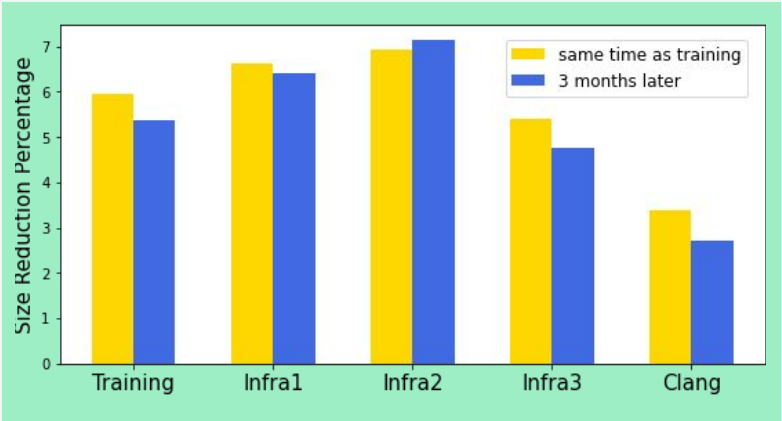
Performance on different apps, Clang



Performance on SPEC 2006

Generalizability across Time

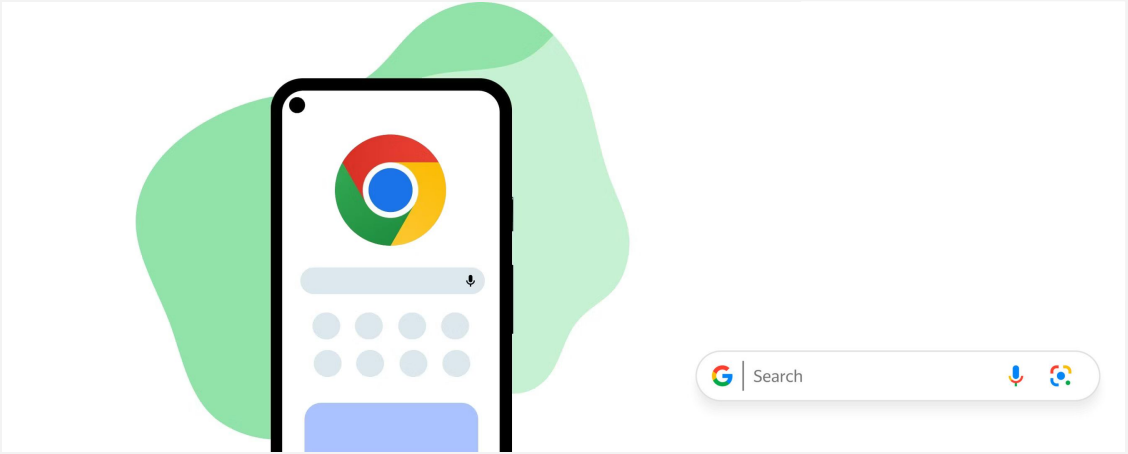
- Effectiveness may degrade
- Still decent wins compared with current-Oz



3 months of active development

1.5% ↑ QPS

Google uses it on a number of projects



Thank you!
Questions?