ML Optimizations in Production LLVM
The Next Research Challenges

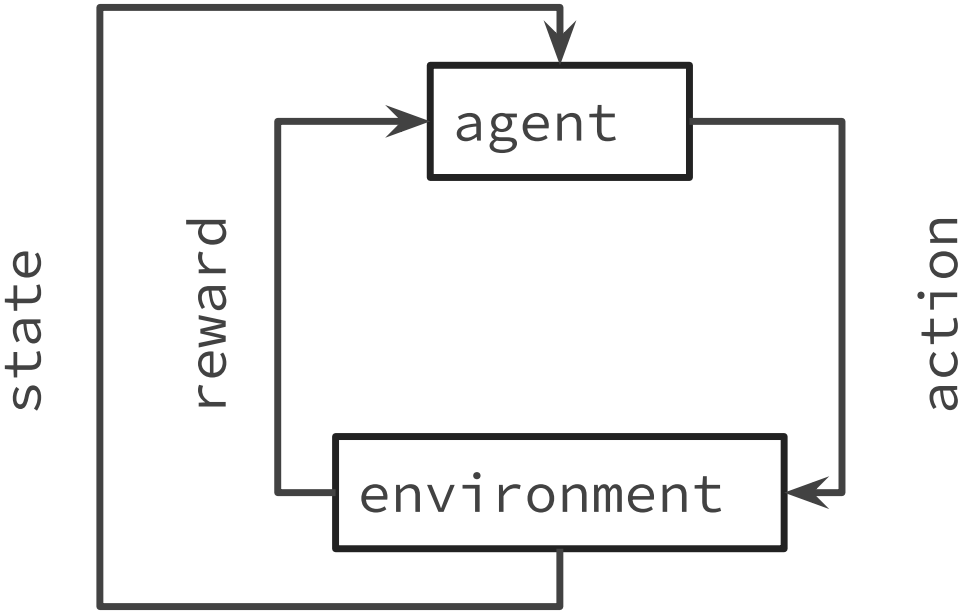
an engineer's opinion

Mircea Trofin | Google

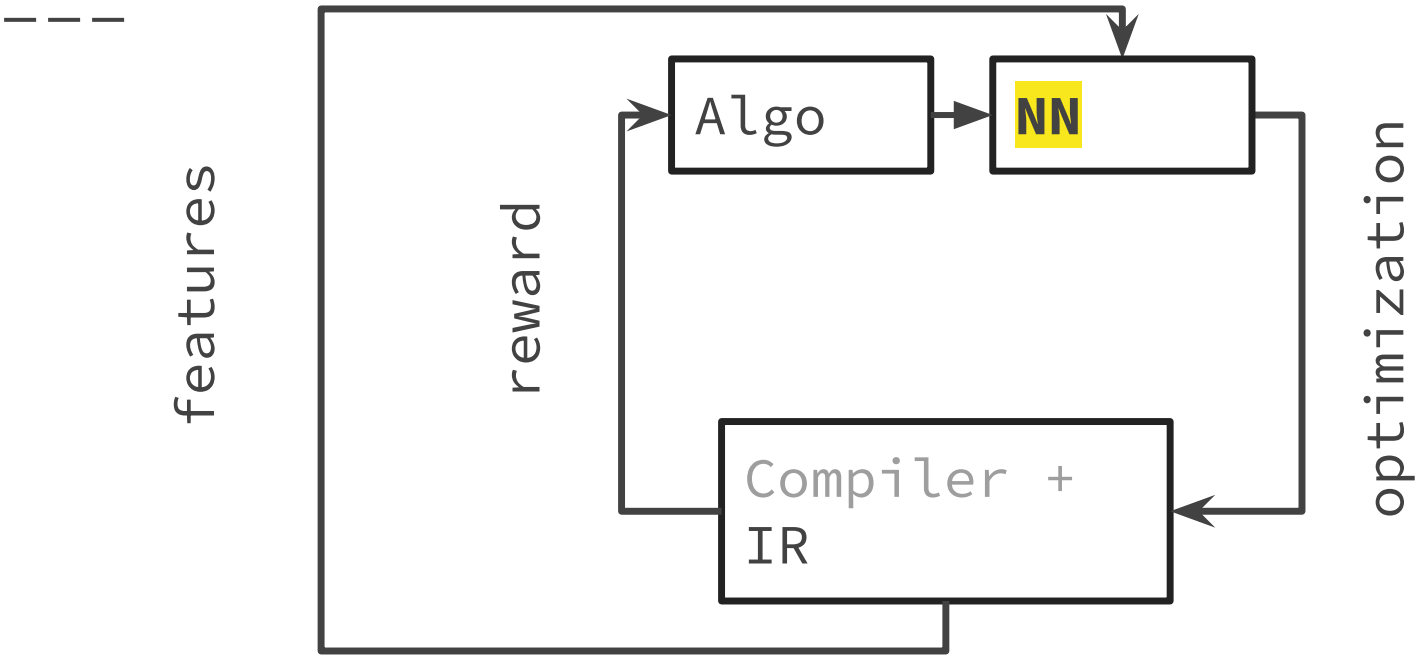
The “Trust me!” slide

- we started replacing optimization ***policies*** with RL-trained ones, in LLVM, **in production**, for the last ~5 years
  - first for size-constrained users (inline for size)
  - then performance (regalloc)
- users at Google:
  - Cloud infrastructure
  - Largest compute consumers in the fleet (incl. search)
  - Android AOSP & Toolchain
  - Chrome on Android
  - Fuchsia
  - Wearables
- common infrastructure in LLVM main tree
  - embedding / using ML models
  - data extraction
  - training corpus extraction
  - regression testing build bot
  - ...

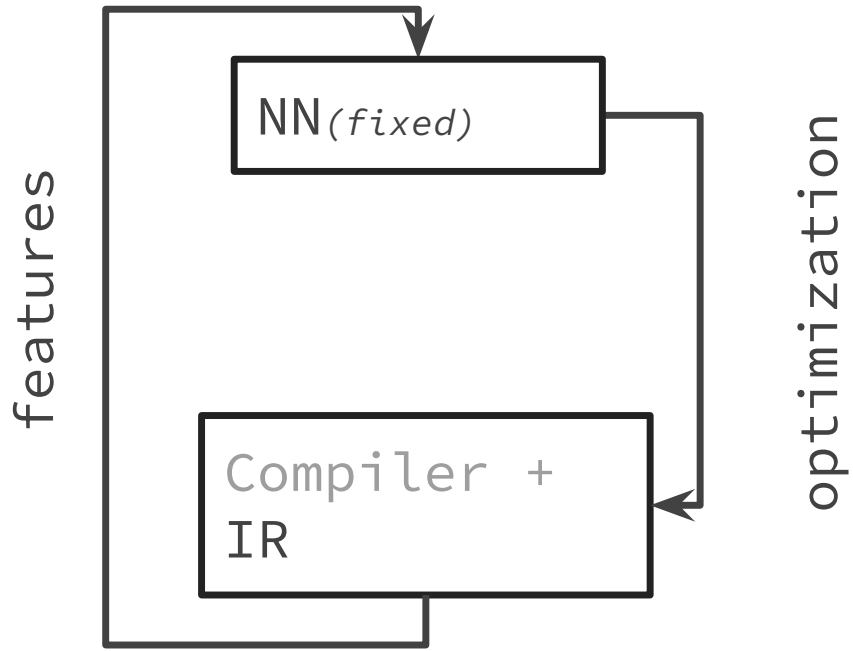
RL Training



RL Training for compilers



inference time



What this presentation *is*

my personal, practitioner opinions

…also, a collaboration[*] pitch:

"When we started applying ML in our production clang compiler we didn't really know what the big obstacles will be.
I think we found some interesting problems that, hopefully, will interest the research community"

[*] direct or indirect, as long as it's in github.com/llvm/llvm-project, we can all benefit!

Plan

- context
- what *(we think)* we learned
- the next problems

LLVM @ Google

- we use LLVM at Google for **all** critical code
- immediately-updated, **read-only** internal fork
  - validated common compiler update every few weeks
- one toolchain for all datacenter binaries
- trained policies are embedded in this build

- Chrome, Android, Fuchsia have similar toolchain deployments, respectively.

Assume “datacenter” for context in this talk

What's a datacenter application?

- a bunch of thread pools
- each thread executes an infinite loop pulling from message queues
- working on some memory shared between them.

```
while (read next message) {

  result = process_message();  <- this is what LLVM optimizes
  enqueue result out;

}
```

What do they spend their time doing? *(...on average...)*

- waiting on cache
  - either kind

=> IPC is low (~1)

- neither data nor instructions fit in any cache
- our profiles are kinda flat
- loops aren't that hot
  - if they were, you're doing data processing wrong: you should shard and parallelize.

But do compilers even matter for datacenter performance?

---

systems design / architecture has a bigger impact

…but the compiler has its place

0.x% *is* a nice performance improvement!

the fleet size makes sub-percentage improvements *very* impactful ($$)

- They reduce the operational cost
- They ***compound***

Our main performance optimization tools

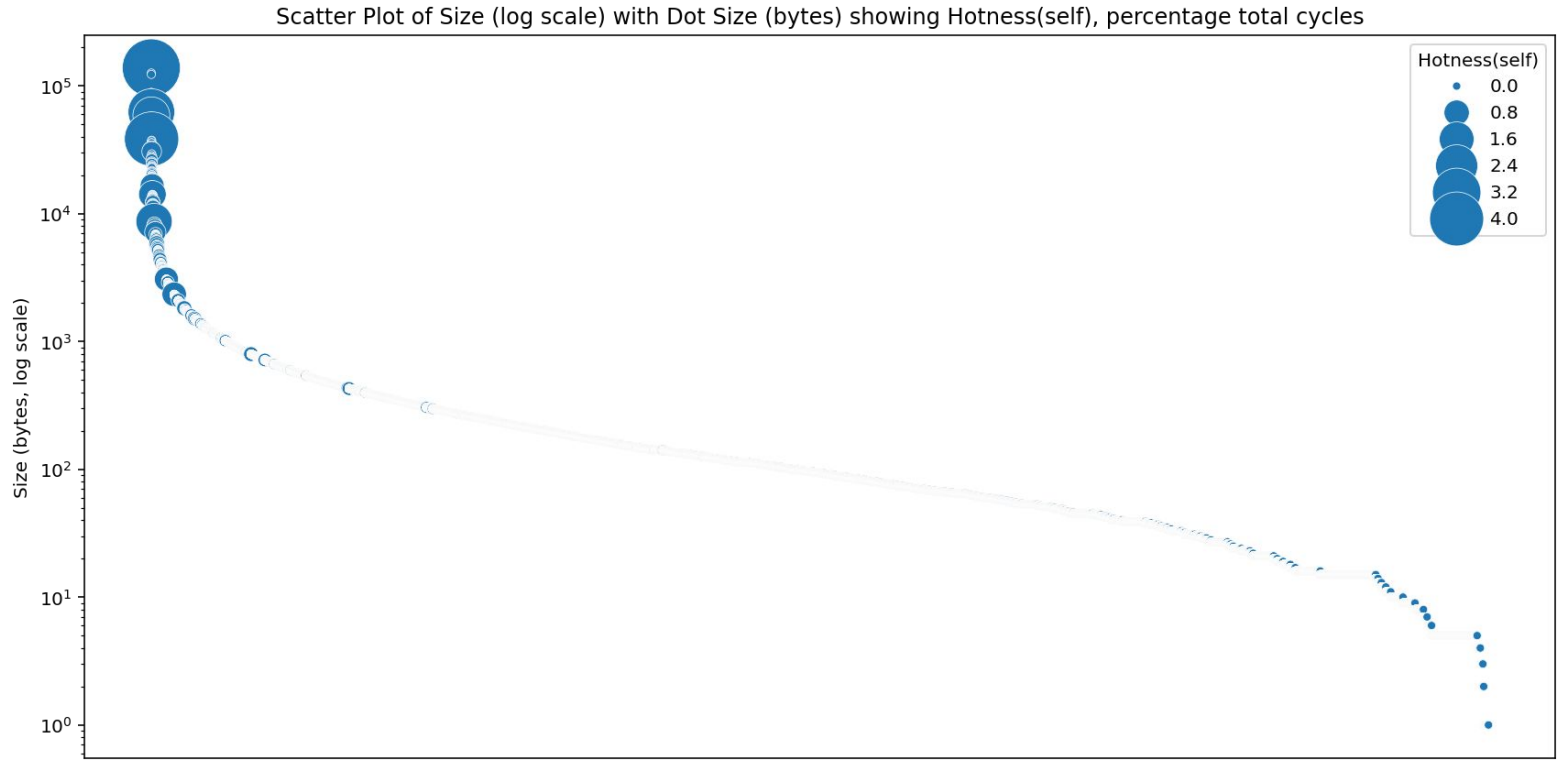
- **PGO** of any kind (e.g. instrumented or sampled)
  - FYI, performance results without some sort of PGO mean little (to us)

- “IPO” encompasses ThinLTO, inlining, ICP, etc
- “block layout” == Bolt/Propeller

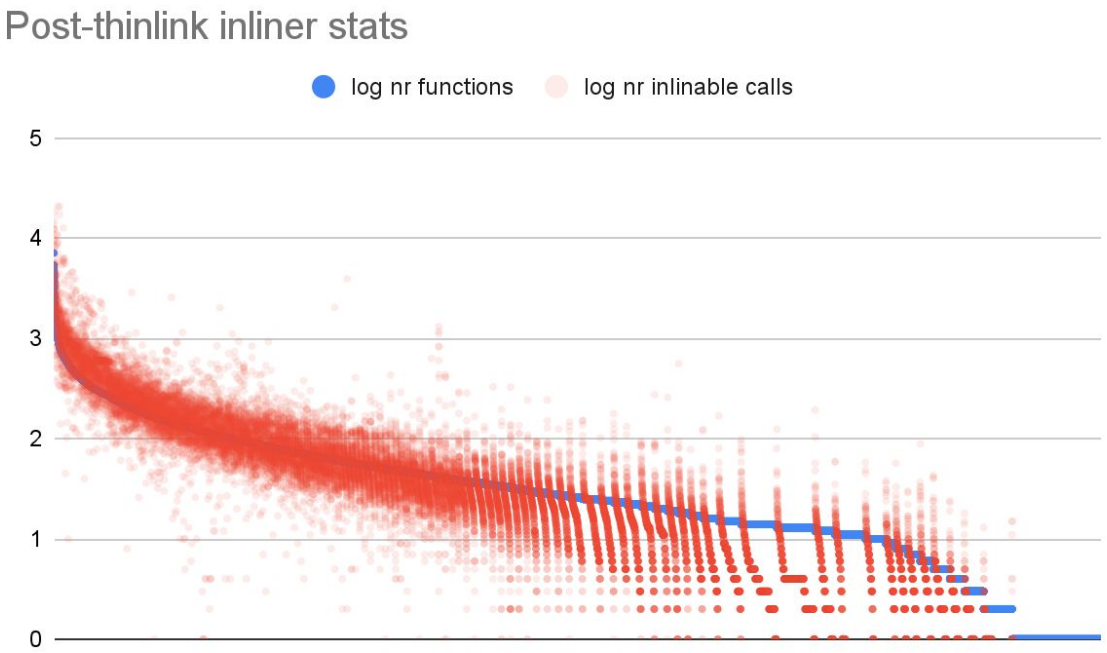
- most of all other opts fall in **percent/sub-percent** improvement ranges (each)

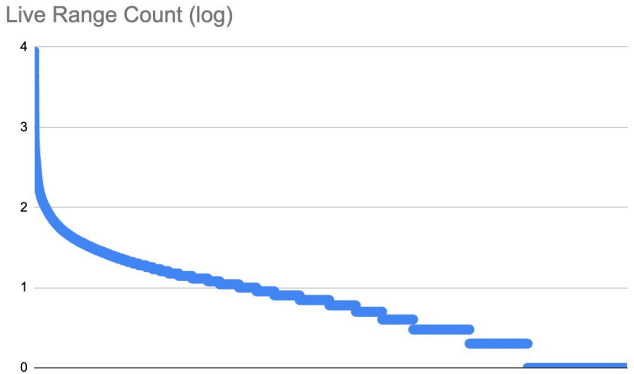
a binary

- 47K modules
- 1.4M functions …but only ~8% of .text is executed at steady state



some statistics about *compiling* that binary





Key points about our problem domain

~~*“solve something small, then scale”*~~

- *the scale is inseparable from the problem*
  - *in fact, it’s part of why we are interested in ML*

  solve a specific, tractable, ***pertinent*** *(i.e. comparable scale)* ***customer problem*** first, then move to additional ***problems***

- *baseline is SoTA (PGO + ThinLTO)*

So why are *we* doing this ML thing?

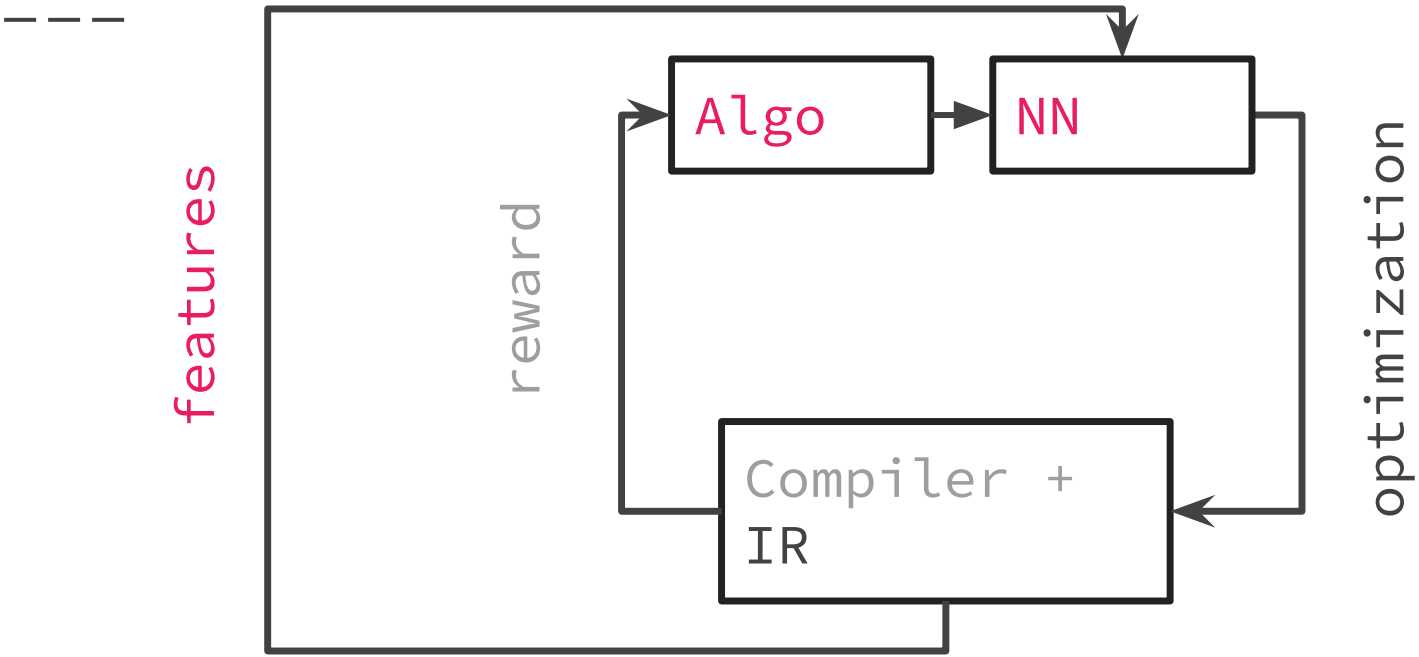
---

automated, **periodic**, **repeatable** discovery / improvement of optimization policies for *large* optimization problems

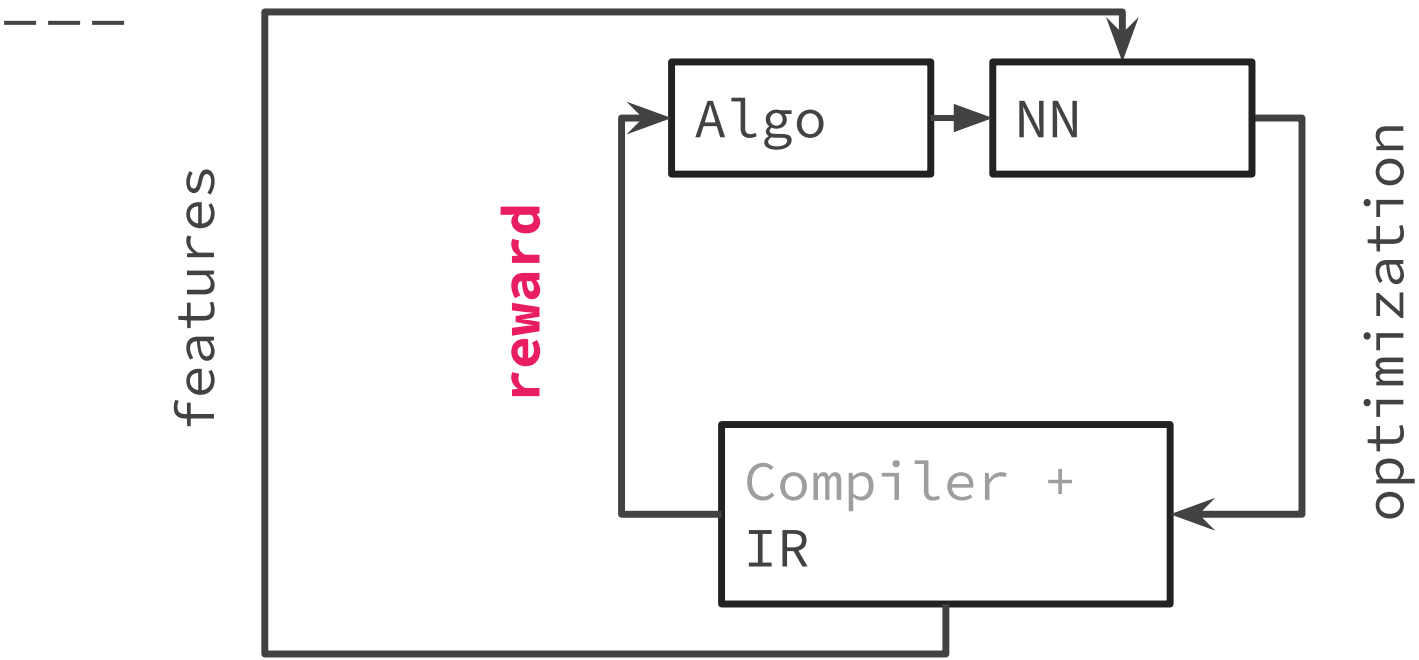
how generalizable? ~fleet-level (i.e. not a goal to be "perfect". just good enough)*

**corpspeak for "binaries we care about"*

Where we thought the problem(s) were



Where the problem is



Why?

- ML techniques are ***very*** data hungry (low “sampling complexity”)

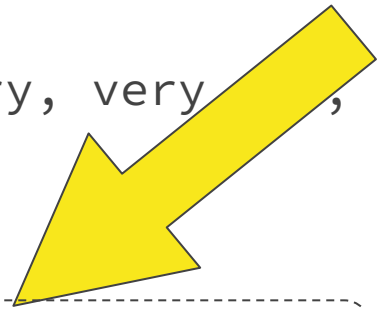
- “run benchmarks” is not the answer
  - micro- look nothing like our targets
  - macro- take hours to run and handle noise; require hardware isolation

so what then?

The ML/AI folks tell us they can train with very, very [illegible], and noisy, data points

-or-

We can estimate sufficiently well the performance impact of a policy without running code



…but before we continue - other things we learned

- is Neural Network incomprehensibility a problem?
  - No

- what about *<my favourite optimization>?*
  - having a solid reward is a prerequisite

- but… AI!
  - to be clear - “AI as compiler” is nowhere in *this (pragmatic)* scope
  - maybe AI can help solve the reward problem

ok, more about AI

- we ***do*** investigate AI as a possible policy generator
  - see Hongzheng Chen, @C4ML, tomorrow
  - still RL-like
  - AI doesn’t appear to meaningfully change the sampling complexity problem

What do we want?

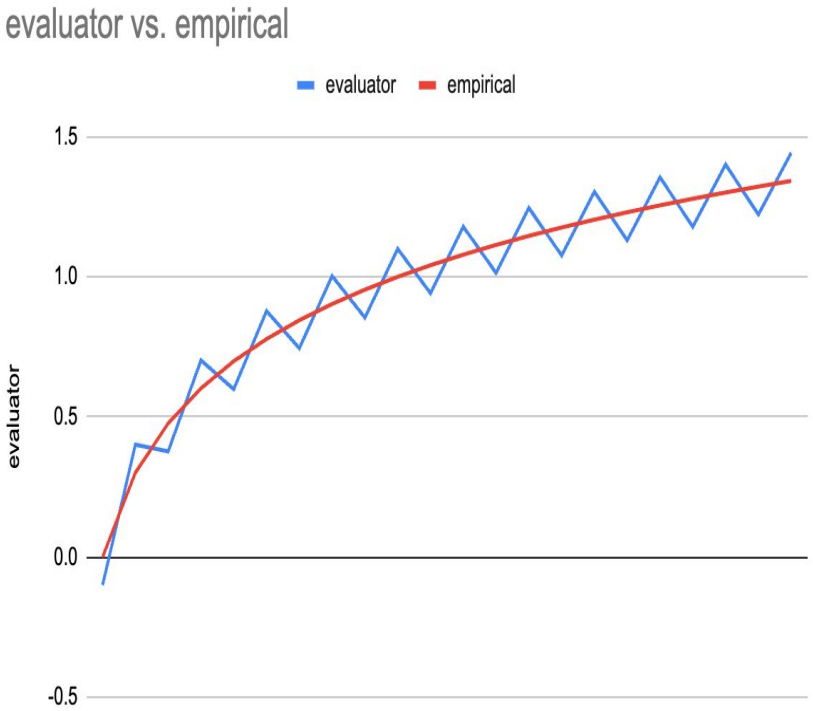
---

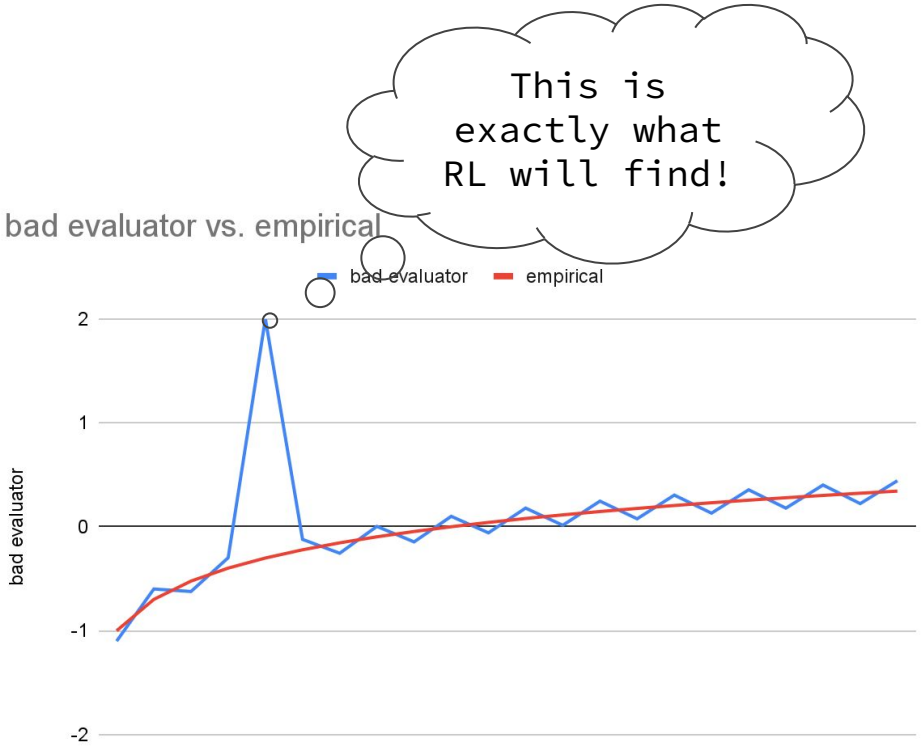
*not a particularly new nor earth-shattering idea, …**but… that pesky scale…***

want a fast* **rank-preserving latency evaluator**

*faster than running

A note on reward hacking





What we tried and what we learned

RegAllocScore.cpp

---

llvm/lib/CodeGen/RegAllocScore.cpp

- for regalloc
- “avoid fills (loads… refills…) in hot blocks”
  - stack -> reg
- spills also… but maybe to a lesser extent
  - reg -> stack

$$\text{sum}(BB_{freq} * BB_{\text{fill\&spill cost}})$$

- TL;DR; didn’t work too well
  - for out-of-order CPUs.
  - like the ones we have and love

2 observations

…well, besides the obvious “maybe the BB latency model is too naive”

1: Lots of problems in PGO

Open challenge: can we accurately estimate the change in #retired instructions per RPC handler when changing optimization policies

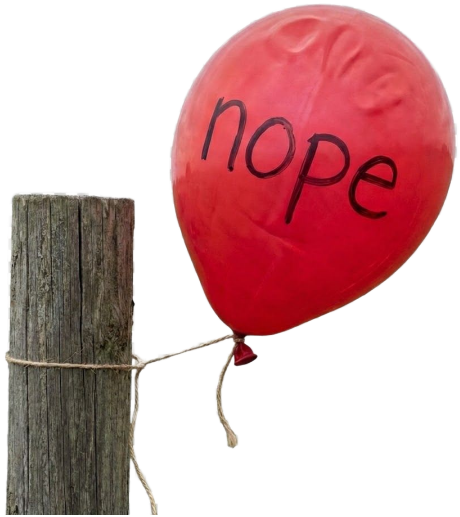
including IPO (inlining, ICP…)

IPO is the elephant in the room

- inlining *(for example)* confounds/dilutes profile info
- multi-module build and ThinLTO make it even worse (function entry counts don't update across distributed compilations)
- the profile of callees differs depending on callers
  - and caller's callers…
  - specifically, all the way to a root
  - main is not the root, request handler is
  - work handlers called by worker threads are

- instrumented contextual profiling [RFC][video][slides]
  - plus the Sampling-Based one from Meta
  - …but sampling-based is a bit loose with *back-propagation*
  - instrumentation gives us tight back-propagation control
  - comes with a "workflow - oriented" ThinLTO feature

so we done, right?



new problems:

- profile propagation
- inter-procedural graph is not fixed
  - pretty deep rathole, won't cover here
  - e.g. libc calls (mem*) get synthesized at various stages (incl. late)

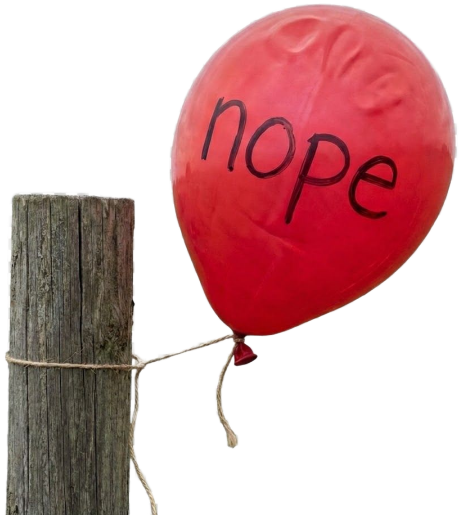
Passes work with profile information of poor quality

- even with instrumented FDO
- even with re-instrumenting after IPO (“CSFDO”)
- **it’s because we are sloppy**
  - if it’s not tested it ***will*** regress (entropy)
  - *“but my pass doesn’t use profiles”* => yes but if it changes the CFG you just made it all worse for everybody after
  - *“but BFI/BPI fixes it”* => not unless you believe you can create information after you lost it (*hint: not in this Universe*)
- this hurts 2 ways:
  - the reward
  - optimization: ML will find opportunities across many lukewarm blocks (so if the profile “lies”, the opportunity disappears… or flips)

The Profcheck Effort

- RFC - fix profile propagation, treat it like a functional regression
- **we’re on it!**

so we done, right?



Remaining big profile propagation problems (examples)

- currently just *presence* of profile
- hard problem: value propagation.
  - some directions in research collaboration *(w. Prof D'Elia's group, Sapienza University, Rome)*
  - also, Elisa Frölich's OOPSLA paper)
- handling stuff like this:

```
%x1 = a && b

%x2 = %x1 && c

br i1 %x2, … !prof !1
```

```
%x1 = a && b

br i1 %x1, label %y1, .., !prof?
y1:
  %x2 = %x1 && c
  …
```

2: Traces FTW!

Open Challenge: can we closely approximate instruction traces from profiles, one-off traces, and any static info?

Basic Block thinking isn't sufficient

- our model ignored real instruction sequences
- CPU pipelines are longer than a basic block
- *…and we didn't even worry yet about caches*
- enter MCAD / MCA Daemon
- key insights:
  - look at ***traces*** instead of CFG
  - since we care about *differences*, **absolute latency prediction is less important** (mcad used llvm-mca)
    - Errors/limitations actually, may cancel out
    - E.g. for regalloc - cache misses.
- **Trace Synthesis**
  - the evaluator can't require re-collecting a trace
  - maybe we can synthesize them

It works... for regalloc

- Aiden Grossman @EuroLLVM 2025 [video][slides]
  - update at LLVM 🩷 ML 2025 [slides]

- Instruction trace -> basic block* trace
  - then re-fill the basic block trace with instructions after regalloc changes

**rather, segments of instructions between consecutively-ocurring branches. Including call/ret*

Pseudo-asm	A Trace	Derived BB-trace	Post-regalloc asm	***Synthesized trace***

// f1()
f1 entry:
  ins1
  ins2
  call f2()
  test
  cond br b2
b3:
  ins4
b2:
  ins5
 	
// f2()
f2 entry:
  ins3
  ret
	ins1
	ins2
	call f2()
	ins3
	ret
	test
	cond br b2
	ins5
		segment1.1
		segment2
		segment1.2
		segment3
			f1()
			f1 entry:
			  ins1
			  ins2
			  move1
			  call f2()
			  test
			  cond br b2
			b3:
			  ins4
			b2:
			  ins5
			  move2
			
			f2()
			f2 entry:
			  ins3
			  ret
				ins1
				ins2
				move1
				call f2()
				ins3
				ret
				test
				cond br b2
				ins5
				move2

But...

- even for regalloc, subsequent optimizations change the CFG
  - we disabled them for training… hoping improvements are resilient to re-enabling them
  - “hope” is not a long-term solution

Some ways to get past CFG mutation limitations

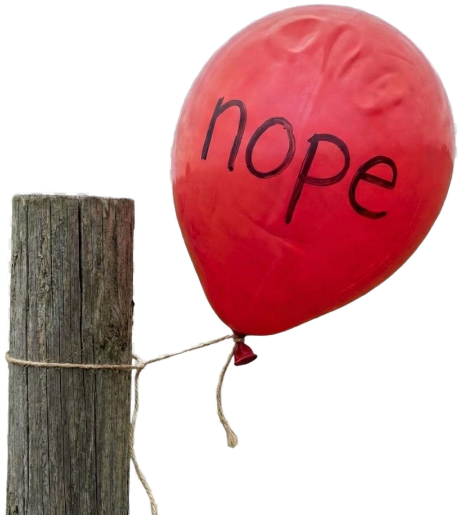
- can we synthesize plausible traces given profile + asm (i.e. at AsmPrint stage)
- can we estimate latency at an earlier representation (i.e. say after some function pass. Or after IPO), given whatever one-off info (baseline traces, diff in codegen)
- (ideally) can we collect full paths?
  - the scale is the challenge
- can we use inputgen?

In summary

- Main problem: rank-preserving predictor for PGO compiled large binaries with flat profiles
- translates into:
  - better profiling
  - better profile propagation
  - path (or trace) prediction

solutions here are also “good for the compiler”, independent of ML

and so ***then*** we're done, right?



caches? (any and all of them)

some initial work [slides]

…far from done

But we don’t have datacenter examples!

- fleetbench as *an* approximation
- **clang** is actually pretty representative
- explicit recommendation *against* SPEC

- **and please remember:**
  - **With (Thin)LTO**
  - **With PGO.**

The case for contributing to LLVM main branch

…as opposed to some fork

- get engaged in a ***collaborative*** community
  - create and build foundation for more ***new*** **research**
- life at tip-of-tree is good for you :)
  - LLVM is actively evolving: use the latest tech
  - no point in working off what was there 2 years ago
- empirical validation
  - on production workloads
  - stronger research results
  - remember - users keep us honest!
- case and point: IR2Vec [RFC] is now part of mainstream LLVM

Where to find us

https://discourse.llvm.org/tag/mlgo

Monthly (if agenda) meeting (see above)

LLVM 🩷 ML Workshop @LLVM Dev Meeting