



# Accurate Runtime Performance Estimation for Predictably Training ML Guided Register Eviction Policies

Aiden Grossman

EuroLLVM 2025



**Why do this in the first place?**

**Rewards are critical for training learned heuristics.**





# Trace-based Cost Modeling

# Agenda

**01** Trace Based Cost Modeling for Register Allocation

**02** (Distributed) Training of Models

**03** Further Improvements

# Traces vs PGO Data

## PGO-Based

```
.loop:
    add    eax, dword ptr [rdi + 4*rdx]
    inc    rdx
    cmp    rcx, rdx
    jne    .loop
```

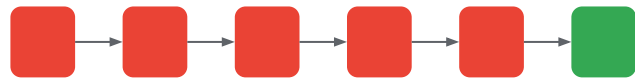
X5

```
.exit:
    add    eax, 5
    ret
```

X1

- Reward is a linear combination of instruction types multiplied by their block frequency.

## Trace-Based



- Reward is some operation (typically cycles) over the sequence of instructions. We have several options to choose from:
  - Analytical CPU pipeline models.
  - ML based CPU models.
  - Raw Instruction Counting.

# Sourcing Trace Data

- Previous work, llvm-mcad (EuroLLVM 2022 mshockwave@), (<https://youtu.be/ZGEP7JEIKNo>) that guided us down this direction used a QEMU plugin.
- Use DynamoRIO to produce traces due to good internal support.



# Basic Block Trace Modeling

We need to be able to collect a single trace and apply it to many variants of the same binary as rerunning each time defeats the point.

# Sourcing Basic Block Information

Take Advantage of some of the Propeller Infrastructure  
(Basic Block Address Maps).

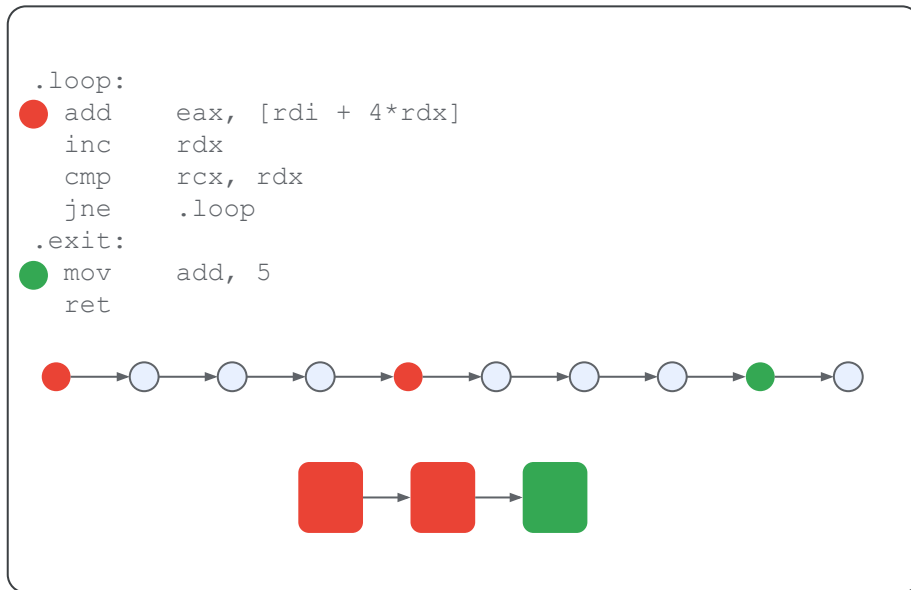




# BB Trace Extraction

How to turn an instruction stream into a BB stream that can be replayed.

- Add a basic block to the trace every time we encounter an instruction with a PC starting at the beginning of a BB.
- Sometimes we need to split BBs.
  - Call Instructions
  - TCMalloc RSeq
  - Inline Assembly

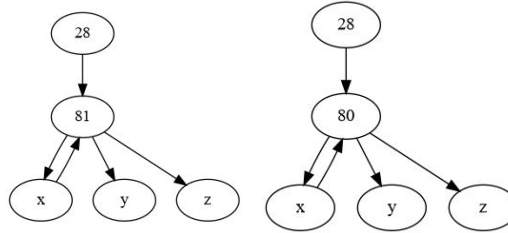
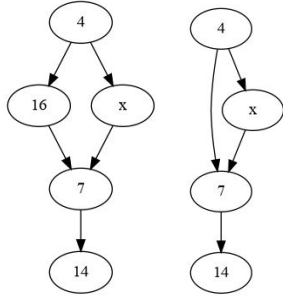


# Basic Block Trace Modelling

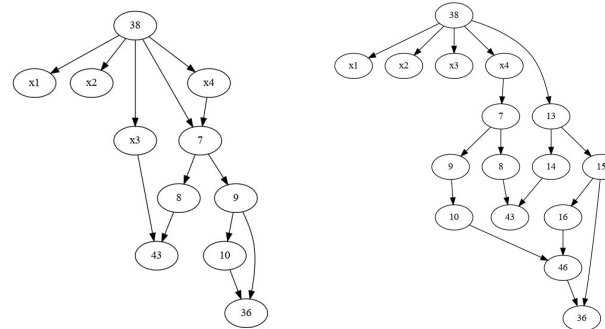
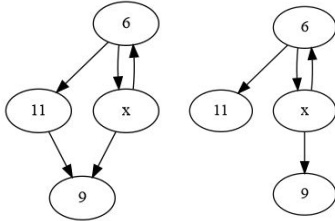
Becomes as simple as loading BBs from the  
binary/compiled corpus.



# Reconciling CFG Differences - The Problem



- Most cases are trivially reconcilable.
- Some cases are impossible to reconcile without additional information.



# Reconciling CFG Differences - The Side Step



## Disabling 3.5 Passes eliminates all CFG Differences

### Findings

1. Disabling three (believed to be) non-regalloc-coupled passes eliminates all CFG differences.
2. Disable one option on another pass (the half a pass).
3. Simple trick allows for much simpler BB trace modelling design.

# BB Traces are Reasonably Close to Source Traces



**0.5% missing  
instructions**

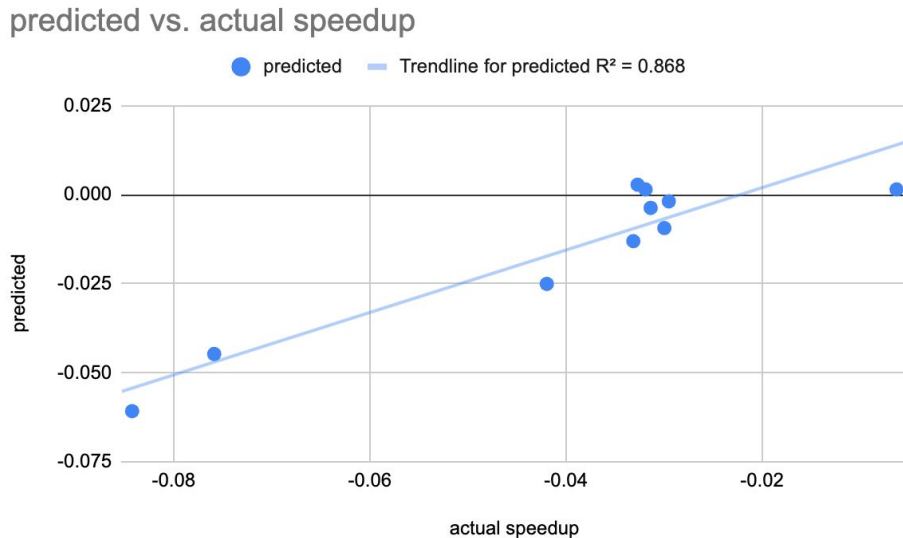
On traces upwards of 10M instructions.

Some cases we are not handling currently like interrupted restartable sequences and symbols from assembly files. No theoretical obstacles to completely fixing the gap.

# It Even Works With PGO+CSPGO+ThinLTO!

For skylake, measuring runtime in cycles. Albeit with a large constant offset.

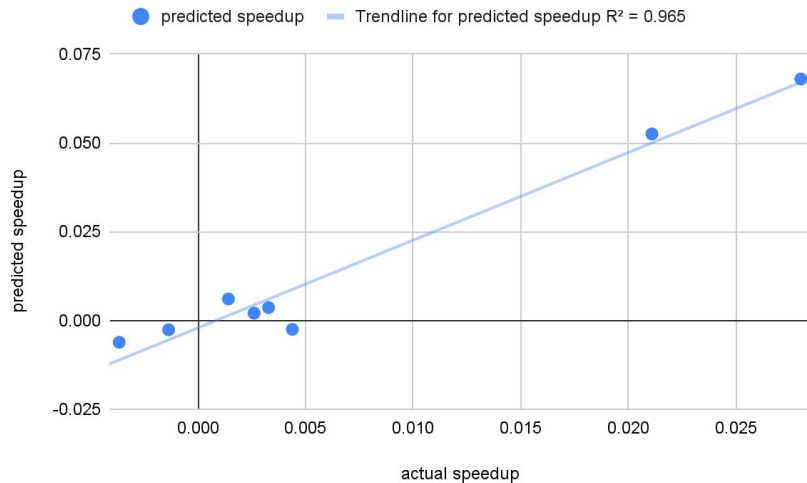
## Predicted Speedup vs. Actual Speedup

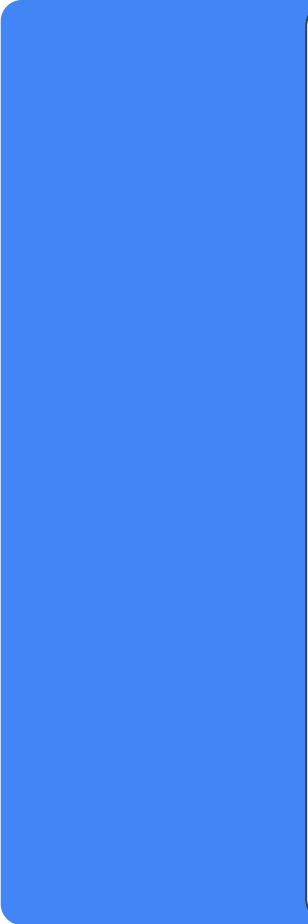


# The Non-PGO case works as well:

`opt -passes="default<03>" -disable-output` on `StructuralHash.cpp` from LLVM. ~10M retired instructions.

**Predicted Speedup vs. Actual Speedup**



A solid blue vertical bar is positioned on the left side of the slide, extending from the top to the bottom.

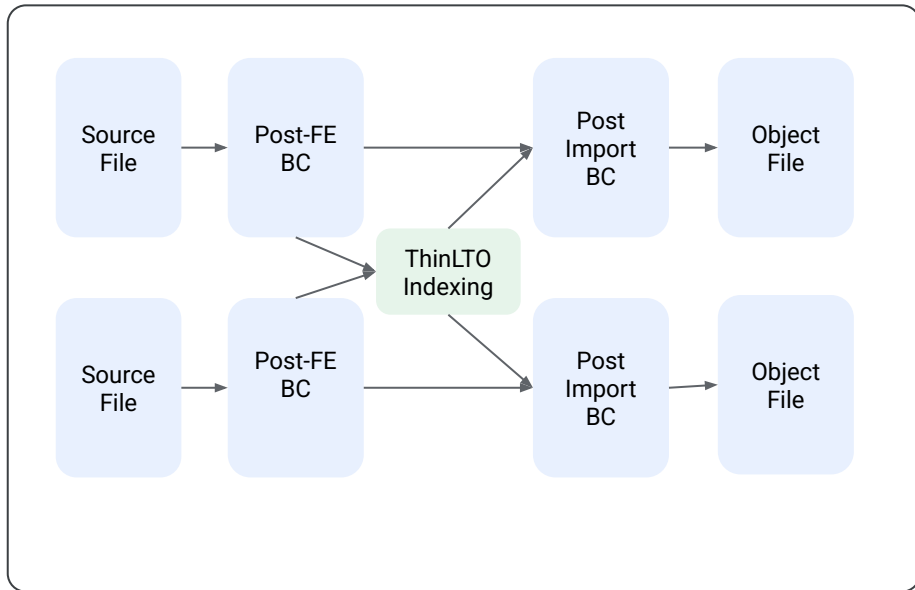
# **Training with Reinforcement Learning**



# Background - The Corpus

How should we efficiently collect training data?

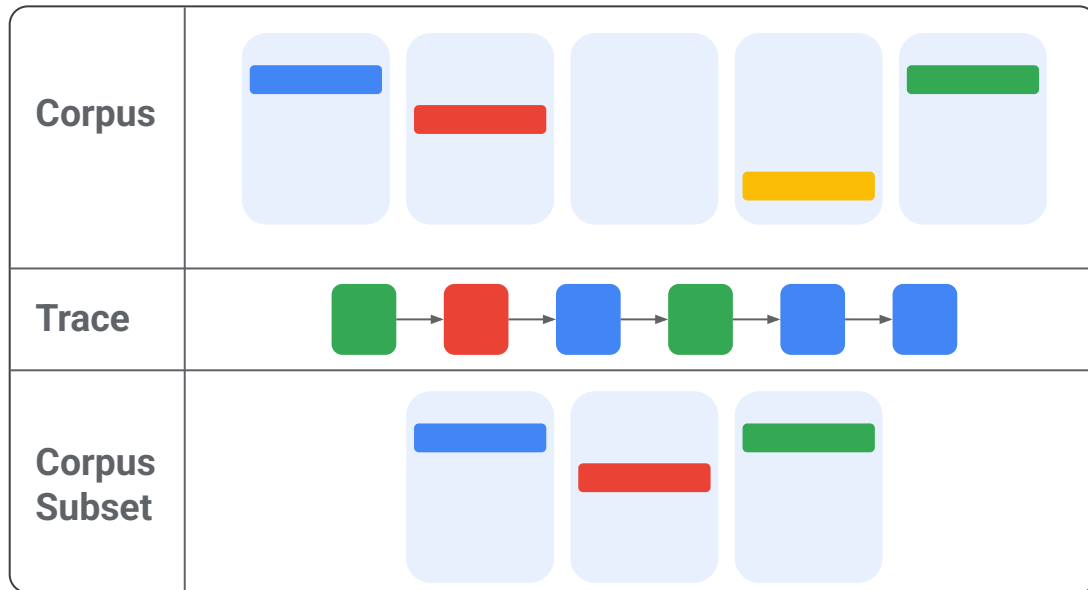
We collect **LLVM Bitcode** for all translation units involved in the final link.



# Corpus Subsetting

How do we efficiently evaluate models?

- Find the minimum set of translation units covering the entire set of functions in the trace.
- Pull them to the side.



# Background - Training Setup

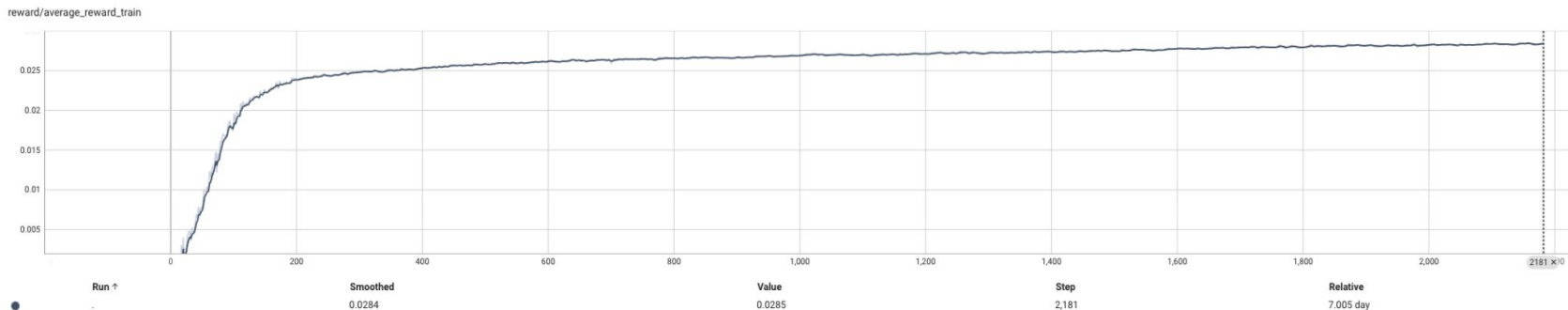
How do the requirements on the ML side interface with the cost model/compiler side?

- We use ES (Evolutionary Strategies) as our training algorithm.
  - Simple math, relatively easy to understand.
  - Enables long trajectories - We can give feedback on many individual decisions and the algorithm will still adjust the policies appropriately.
  - Has a set of perturbations for each iteration.
- Utilize existing training infrastructure
  - But scaled given now we need to compile an entire corpus subset to get a signal rather than a handful of modules.
- First experiments were performed with the same opt invocation from earlier. ~10M retired instructions.

# Training Results

It trains! Somewhat slowly... (for LLVM opt)

## Reward (Predicted speedup over baseline)



- 100 Machine Slices (1600 Threads)
- 100 Perturbations per iteration
- ~800 Modules
- ~7 Days of training time
- 0.5% Real World Performance Improvement over an already peak optimized (PGO+CSPGO+ThinLTO) binary.

## So Why Does This Matter?



We have a **validated, predictive** cost model for real applications.

# Distributed Training

# Distributing the Training Process

Using more machines will at least help.

01

## Parallelize individual workers

We use a threadpool within each worker to enable parallel compilation with the modelling component already being parallelized.

02

## Spawn a bunch of distributed workers.

We use XM to manage experiments, with each worker being given about 32CPU cores, giving us reasonable scalability.

03

## Profit (Somewhat)

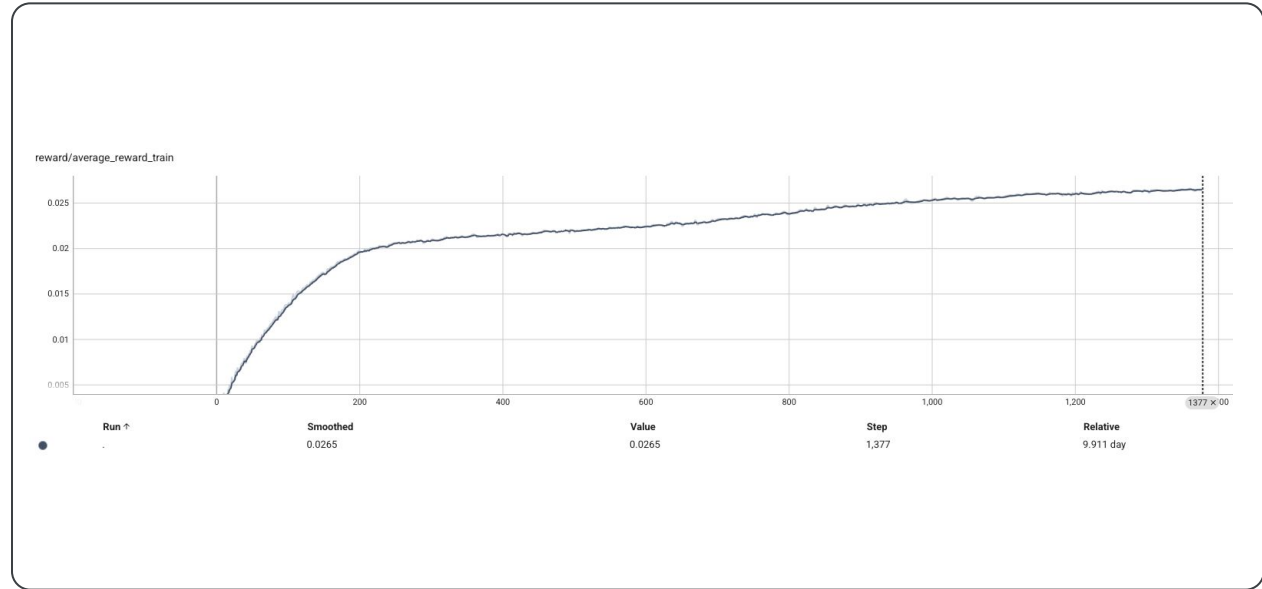
This dropped iterations times to about five minutes. The latency of an individual model evaluation precludes us from going faster. We can evaluate many perturbations in parallel.

# Shipping a Model



# Training - Training Hyperparameter Tuning/RL

RL training started to go significantly faster when we realized the learning rate could be increased 10x with no ill-effect. More experimentation is still needed.



# Some Reasonable Initial Performance Improvements

This is from a new [model](#) trained on a single workload. It ends up generalizing reasonably well.



## Internal Server App 1

- 0.34ms latency on action 1
- 0.24ms latency on action 2



## Internal Search App 1

```
Benchmark Setup 1:
---- total:qps ----
diff: +0.32% ±0.088%
---- <workload1 cpu kcycles> ----
diff: -0.75% ±0.198%
Benchmark Setup 2:
---- <workload2 cpu kcycles> ----
diff: -0.46% ±0.268%
```



## Internal Search App 2

```
-- <round trip latency> --
diff: -0.64% ±0.372%
```

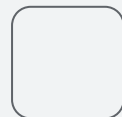


# **Current and Future Work**



**During training, only compiling functions with regalloc decisions causes compile time drops from 3-4 minutes to 5-10s.**

Modeling time remains about the same as it is bottlenecked by MCA. We have some ideas on how to fix that...



# Reducing Modeling Costs

Now that compile times have been drastically reduced, modeling costs dominate. It would be good to reduce them too.

- Only modeling changing functions might provide significant benefits.
  - Benefits depends upon how many functions they call.
  - Needs empirical validation.
  - Natural extensions (like excluding blocks from functions that get called) also need separate validation.
- Trace subsetting - Only evaluate a subset of the traces on each invocation.

# Future Work

# Future Work

## Understanding Constant Offsets

- Understand why our model is producing large constant offsets.
- Hopefully leads to better models.
- Experiment with other modelling techniques (ML based, etc.)

## Ship Better Models

- Utilize more efficient training techniques, train better models and ship them.

## What do we need to do to Generalize for other Optimizations?

- Control flow graph reconciliation?
- Keep track of data/inputs to interpret (M)IR?
- Something else?
- Better cost modelling techniques?

# Thank You!





# Thank You!

(Questions?)

