Opti-ML: An End-to-End Architecture for ML Guided Inlining for Performance

Sam Keyser [1]

Abstract

Optimizing compilers are very complicated pieces of software, relying on heuristics which have been handcrafted and fine tuned by domain experts. In recent years machine learning (ML) techniques have shown promise as a way to learn these heuristics. We introduce work done by two previous authors in the space of ML compiler optimization, specifically with respect to *function inlining*. We also create a full end-to-end pipeline for the LLVM C/C++ compiler capable of (1) creating a dataset, (2) training a reward model, and (3) training an RL agent to resolve inlining decisions during compile time.

1. Introduction

LLVM is a modern framework for creating compilers. It decouples the parsing of source code from the generation of valid machine code by introducing an intermediate representation (IR) that captures program semantics in a language agnostic format (Lattner & Adve, 2004). LLVM is also an optimizing compiler. It performs several passes on program IR in an attempt to improve program size, runtime, or both. A ”pass” is a single traversal through either the complete code, or a segment. These optimizations can be performed at several granularities such as Function, Call Graph, Loop, or Module level (Lattner & Adve, 2004). Passes can generally either gather information or perform a transformation on the module of code. Some passes may also be run several times, either in sequence or after other passes have been performed to take advantage of new changes introduced by prior passes. Passes have been tenatively organized into pipelines such as -O{1, 2, 3, s, z} by compiler authors which optimize for either runtime speed or binary size, at the expense of increased compile time. These pipelines can be quite involved; LLVM's -O3 pass involves approximately 160 optimizations passes which involves optimizations performed at multiple different levels of granularity.

Function inlining is one of the fundamental optimizations implemented by modern compilers. Function inlining passes are applied at the function level and examine each call site within the function. The decision is then made to either perform inlining or not. Inlining is the replacement of the callsite with the entire function definition. It can be thought of as ”copy-pasting” the function definition wherever the function would have been called. Function inlining can improve runtime performance by removing the overhead of jumping to the caller body from the callsite. Additionally, it can improve the performance of downstream optimization passes such as escape analysis by adding more information the current function context (Theodoridis et al., 2022).

Function inlining can have an impact on both the final binary size and the runtime of the final binary. Inlining small functions or functions which are called very infrequently is usually innocuous. The benefit derived from not having to jump to the function definition outweighs the increase in size from duplicating the function definition at each callsite. It becomes more fuzzy when functions are both large and called frequently. Inlining such functions would likely improve the runtime, but would also increase the final binary size. The answer in such a case is not clear. In fact, the inlining problem can be shown as equivalent to a 0-1 knapsack problem, making it NP-complete (Theodoridis et al., 2022).

Function inlining is traditionally solved by heuristics developed by compiler designers. Designing good heuristics is a non-trivial task, but recent work has shown promise in machine-learned policies as an alternative to manually defined policies (Trofin et al., 2021). To our knowledge, two prior works have approached function inlining using machine learning methods: MLGO (Trofin et al., 2021) and MLGOPerf (Ashouri et al., 2022). MLGO chose to focus on creating a heuristic for inlining for size due to several challenges with training an inlining for speed heuristic. MLGOPerf built on the work done by Trofin et al. and proposed a framework for training inlining for speed agents.

Opti-ML builds on top of the work of Ashouri et al. with the following additions[1]:

[1]Electrical Engineering & Computer Science, Milwaukee School of Engineering, Milwaukee, WI, United States. Correspondence to: Sam Keyser <keysers@msoe.edu>.

MSOE Masters in Machine Learning Capstone, Spring 2024.

[1]While not a novel addition, we also produced an open

1. The collection of a dataset based on CppPerfBenchmarks (Cox, 2018) for training inlining for speed agents. We collect this dataset because the dataset used by Ashouri et al. is not openly available. We also make available the infrastructure for collecting datasets of this nature.

2. The replacement of the feature extraction layer within MLGOPerf's IR2Perf model with an embedding layer, IR2Vec (VenkataKeerthy et al., 2020). MLGOPerf relies on extracting a set of manually chosen features from the compiler, which are fed to their model. IR2Vec introduces the ability to embed the entire representation of a module of code, potentially providing a much richer representation of the IR.

3. The use of a xgboost random forest as an alternative to deep neural networks for IR2Perf. We found empirically that the xgboost random forest outperformed the neural network, at least when using the IR2Vec embedding layer. We speculate on why this might be later.

The rest of the paper is organized as follows. Section 2 discusses the architectures of MLGO and MLGOPerf in more detail, as well as IR2Vec and the default inlining heuristic for LLVM. Section 3 provides detail on how we collected our dataset and our proposed inlining-for-speed training scheme as well as the hardware & software used for training our agent. Section 4 presents the results of our function inlining heuristic against the default inlining heurstic and the MLGO v1.0 agent. Finally, section 5 discusses current shortcomings, challenges, and proposes future work. Section 6 concludes the paper.

2. Background

2.1. LLVM Inlining Heuristic

The function inlining pass used by the LLVM C/C++ compiler operates on strongly connected components (SCCs) within the call graph of the module under compilation. Every call in every function is considered for inlining, and a yes/no decision is resolved at each point. Finally, several optimizations are applied on the processed SCC after all inlining decisions have been resolved as "cleanup" (Trofin et al., 2021). These optimizations can have downstream effects on later inlining decisions.

The inlining decision itself consists of several heuristics. A static cost is estimated for inlining the callee into the caller by simulating post-inlining cleanup passes on the callee. A threshold, estimated based on call site hotness and inlining hints, is applied to the cost to determine the go / no-go decision on inlining the callee (Trofin et al., 2021). The estimated cost can also be influenced by the number of SIMD instructions within the callee or the number of basic blocks[2]. The compiler can also choose to defer an inlining decision, returning the callee back to the queue of call sites to consider, if inlining the caller may result in better savings.

source implementation of MLGOPerf's training architecture: https://github.com/TrainingAfternoon/opti-ml

2.2. MLGO

2.2.1. Overview

MLGO, or the "Machine Learning Guided Compiler Optimization Framework" (Trofin et al., 2021), was an effort to bridge the gap between academic investigation of machine learning methods for compiler optimization and real world compilers. Trofin et al. worked with the LLVM community to introduce a drop-in, machine-learned replacement for the inlining decision heuristic. Trofin et al. chose the inlining decision heuristic for both the importance of the heuristic, and the ease of measuring final binary size compared to a more noisy statistic like program runtime.

Trofin et al. used both reinforcement learning (RL) and evolutionary strategies (ES) to train candidate models for the inlining-for-size heuristic. While Trofin et al. found that the ES method performed slightly better than the RL method, we focus on the RL method in this paper as later work builds on it.

2.2.2. Reinforcement Learning

Reinforcement learning is a method for training an agent to understand, and interact with, an environment. The agent, or interchangably the model, learns via trial and error by interacting with its environment. At each time step the agent receives some state s_t which is a collection of features that represent the environment. It takes some action a_t according to its policy, $\pi(s_t)$, which is a distribution over all actions the agent could take conditioned on its current state. After taking an action it receives both its next state and a reward based on the new state, $s_{t+1}, r_{t+1} = \mathcal{R}(s_t, s_{t+1})$. Note that $\mathcal{R}$ is the reward function and is a user designated function. $\mathcal{R}$ is what drives the agent towards a particular goal. In deep reinforcement learning the policy π is learned via a deep network which we will represent as π_θ where θ is the parameters defining the model.

Reinforcement learning can also be thought of as solving a Markov Decision Process (MDP), which is a mathematical framework for repesenting sequential decision making problems such as inlining-for-size. The MDP is represented with the 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$. Respectively, $\mathcal{S}$ the set of all states, $\mathcal{A}$ the set of all actions, $\mathcal{P}$ the probability distri-

[2] A sequence of instructions ending with a yield of control flow. A function can potentially be composed of several basic blocks

bution of state transitions $\mathcal{P}(s_{t+1}|s_t, a_t)$, and $\mathcal{R}$ the reward function. The goal of RL algorithms is to find some π^* that maximizes the total reward[3] $R = \Sigma_{t=0}^{T} \gamma^t r_t$.

Trofin et al modeled inlining-for-size as

- **state** $\mathcal{S}$: the current call graph and the current call site being visited
- **action** $\mathcal{A}$: $\{0, 1\}$ where 1 = inline, and 0 = do not inline
- **state transition probability** $\mathcal{P}$: $\mathcal{P}$ is captured by the compiler. Inlining is a fully deterministic system in that the same compiler will behave the same way each time when resolving an inlining decision on the exact same state.
- **reward** $\mathcal{R}$: the native size reduction after inlining is taken (or not taken).

Trofin et al. used Proximal Policy Optimization (PPO) (Schulman et al., 2017) which is a member of the Policy Gradient (Sutton et al., 1999) family of RL algorithms. Policy Gradient methods directly learn some π_θ which is updated via $\nabla J(\theta)$ where $J(\theta)$ is the expected reward under π_θ.

In the interest of simplicity we explain a policy gradient algorithm called REINFORCE (Williams, 1992) here. PPO is an enhancement on REINFORCE and the full details can be found in (Schulman et al., 2017).

$\nabla_\theta J(\theta)$ is computed as:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\Sigma_{t=0}^{T} R \nabla_\theta log \pi_\theta(a_t|s_t)\right] \quad (1)$$

In practice this expectation is approximated with Monte Carlo methods using n trajectories[4]. θ is updated as:

$$\theta \leftarrow \theta + \alpha \frac{1}{n} \Sigma_{i=1}^{n} \left\{\Sigma_{t=0}^{T} R_i \nabla_\theta log \pi_{theta}(a_{i,t}|s_{i,t})\right\} \quad (2)$$

Note α is the learning rate.

Trofin et al. made one change on top of PPO. Because of difficulties in getting native size rewards at each step of the inlining process, they replaced the partial reward with the total reward R in equation 1 (Trofin et al., 2021).

In addition, due to the complexity of the state space, Trofin et al. simplified $\mathcal{S}$ to a set of 11 numerical features.

Finally, Trofin et al. used imitation learning to "warmstart" the RL agent to imitate the behavior of the default inlining heuristic. This is speed the agent's training by giving it a good baseline policy.

[3] $\gamma \in [0, 1]$ is the discount factor. MLGO takes $\gamma = 1$, hence we do not discuss it in depth here

[4] A trajectory, or an episode, is a sequence of $(s_0, a_0, r_1, s_1, a_1, ..., a_{T-1}, s_T, r_T)$ and the total reward R

2.3. MLGOPerf

MLGOPerf (Ashouri et al., 2022) builds on top of the work of Trofin et al. in order to create an inlining-for-speed agent. One challenge identified in the MLGO paper was the difficulty in training for speed as rewards related to program runtime would be collected, and progam runtime is both very noisy and very expensive to collect (Trofin et al., 2021). Ashouri et al. proposed the addition of a second model, IR2Perf, to estimate program runtime from the IR level which enables the generation of program runtime rewards during MLGO's training process.

Ashouri et al. collected a corpus of IRs using an autotuner, which uses algorithmic search to assist the compiler's optimization passes, to generate different inlining decisions. Ashouri et al. used the SPEC CPU2006 benchmarking dataset (Henning, 2006) as the basis for their corpus.

Each IR example was compiled with instrumentation[5] and ran under the Linux perf tool[6] in order to collect runtime data. In particular, perf was used to collect T_{Func} and T_{Total} which were the percentage of time spent in an individual function and the total program runtime respectively. The llvm-profdata tool was used to report N_{Func} which was the number of time each function was called. Using these statistics Ashouri et al. computed the runtime of each function:

$$Func_{runtime} = \frac{T_{Total} * T_{Func}}{N_{Func}} \quad (3)$$

Using the results of equation 3 the total speedup of a function compared to the function compiled with no tuning and default compiler settings was computed as:

$$Func_{speedup} = \frac{Func_{runtime(Base)}}{Func_{runtime(X)}} \quad (4)$$

where X denotes an example from the corpus.

The IR, $Func_{speedup}$ pairs were used to train IR2Perf in a supervised fashion. Prior to training, a collection of 20 features were extracted from the IR to be used as input to the model. Redundant data points were also pruned from the dataset, and the data was normalized and scaled. Finally, PCA was applied to the dataset with $PC = 7$ components. The motivation for using PCA was the reduce the sparsity of the data. IR2Perf was trained using Mean Squared Error (MSE) loss function for 5000 epochs.

Other than modifying MLGO to use the IR2Perf network to generate runtime rewards, the MLGO training architecture was left unchanged.

[5] https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation

[6] https://perf.wiki.kernel.org/index.php/Main_Page

2.4. IR2Vec

IR2Vec is a general purpose embedding model for LLVM IR (VenkataKeerthy et al., 2020). Embedding models are models which try to learn a transformation over some distribution that preserves the information within the original domain. They are frequently used in the context of NLP to convert natural language into a numeric format within There are two broad classes of embedding models: context window based, like word2vec (Mikolov et al., 2013), and knowledge graph based, like TransE (Bordes et al., 2013).

Context windows try to infer token meaning using proximal tokens. Knowledge graphs, which use an entity-relationship model, can capture longer range relationships by observing the relationships participated in by each entity. A knowledge graph is composed of triplets $\langle h, r, t\rangle$ where $h, t \in Entities$ and $r \in Relations$. The representations of h, t are learned as translations from the head entity h to the tail entity t via the relation r in a high dimensional embedding space (VenkataKeerthy et al., 2020). Given any pair of the triplet, the third member of the triplet should be recoverable.

IR2Vec uses a knowledge graph based embedding model. Each instruction within the IR file is represented as an entity. Each instruction is considered to participate in the following relations: (1) opcode & type of the instruction, (2) the opcode & type of the subsequent instruction, and (3) the relation between the opcode and each of the instructions arguments. More formally, each instruction l can be represented $\langle O, T, A\rangle$ where O is the opcode, T is the type, and A is the arguments of the function $A_1, A_2, ..., A_n$. IR2Vec computes the instruction vector as:

$$W_o.O + W_t.T + W_a.(A_1 + A_2 + ... + A_n) \quad (5)$$

where . is the scalar multiplication operator and W_o, W_t, W_a are scalars $\in [0, 1]$. W_o, W_t, W_a are chosen in accordance with a heuristic such that $W_o > W_t > W_a$.

IR2Vec refers to instruction vectors such as these as "symbolic vectors". The instructions are used in conjunction with TransE to learn a "seed embedding vocabulary" which is used to lookup instruction vectors based on instruction at inference time. There are a second class of instruction vectors which IR2Vec refers to as "flow aware" which make use of the other information within the IR such as $use - def$ (UD) information. UD information encodes the different lifetimes of a variable, or its uses. Variables hich are currently in use are said to be "live". Flow aware embeddings are described in more detail in the original paper (VenkataKeerthy et al., 2020).

Live instruction vectors are grouped by the basic block that they belong to and summed to create basic block level embedding vector. Functions are composed of potentially several basic blocks and can be created in the same way. Similarly, function level embedding vectors can be summed to create a representative vector of the entire module.

3. Methods

3.1. Computing Environment

All work for Opti-ML was performed in a Debian 12 Bookworm VM with a 16-core ARM Neoverse-N1 and 256GiB RAM. A complete list of software packages necessary for setting up the compute environment can be found in the project repository[7].

3.2. Data Sources

3.2.1. CppPerformanceBenchmarks

CppPerformanceBenchmarks is a collection of C++ files intended to evaluate C++ compiler performance (Cox, 2018). CppPerformanceBenchmarks consists of 69 distinct benchmarks. We chose a subset of examples which ran in under two minutes: (1) machine.cpp, (2) functionobjects.cpp, (3) loop_removal.cpp, (4) rotate_bits.cpp, and (5) scalar_replacement_structs.cpp.

We used CppPerformanceBenchmarks as the basis for our training and validation datasets. We chose CppPerformanceBenchmarks because of its (1) free availability, (2) being compatible with the Clang compiler, and (3) it's ease of use.

3.2.2. cBench

The Collective Benchmark (cBench) is a collection of open-source programs with an emphasis on portability and support for several compilers such as GCC, LLVM, etc. The cBench collection has been used for evaluating compiler optimizations within MILEPOST GCC before (Fursin, 2010).

We used cBench as our testing corpus. It has similar positives to CppPerformanceBenchmarks, as well as being the testing corpus used by MLGOPerf (Ashouri et al., 2022).

3.3. IR2Perf

3.3.1. Data Collection

Unlike MLGOPerf we chose to forgo the use of an autotuner for a few reasons: 1) there were no good open source tuners available for our case, 2) we thought that by instead generating samples under varying inlining conditions we might more uniformly explore the space of inlining configurations, hopefully helping the IR2Perf performance. We hope that providing the infrastructure for collecting corpuses of IR files with different inlining configurations will help efforts

[7] https://github.com/TrainingAfternoon/opti-ml

in this area.

We varied the default inlining heuristic by modifying the '-inline-threshold' flag. We used thresholds τ uniformly randomly sampled from $\{x \mid 0 \leq x \leq 1000,\ 0 \equiv x(mod\ 25)\}$. The inlining threshold controls the cost which a function must stay under to be considered for inlining. Additionally, in order to collect the data for training IR2Perf we compiled each program with '-fno-omit-frame-pointer'[8] and '-fprofile-generate'. We then ran each binary to record runtime information. We limited each run to 1 CPU core via numactl. The system page cache was cleared after each run to avoid influencing subsequent runs. During execution the binaries were monitored by the Linux perf tool and run with LLVM profiling enabled.

Linux perf is a event-oriented observability developed for use by Linux kernel developers. It is capable of observing hardware events like CPU performance counters, as well as taking snapshots of current CPU usage using custom timed interrupt events (Gregg). We sample with a frequency of 500Hz. We record the number of CPU cycles observed during progam execution, as well as snapshots of the call stack at each sample. This is used to compute an "overhead" value for each function, or how much time is spent within each function during the lifetime of the program.

LLVM profiling injects hooks into the compiled code which report function usage counters. Profiling does have an impact on code runtime as additional instructions are added to the final binary to be executed.

Traces for each function for benchmark file were collected, except when profiling information was not able to be collected. Because of its nature as a sampling profiler, perf was capable of "missing" a function, especially in the faster benchmark files.

For each function that we had N_{func} and T_{func} information for, we computed $Func_{speedup}$ using equation 4. We computed $Func_{runtime}(Base)$ for each function by taking the average runtime of all instances of the function compiled under the default heuristic settings[9] in the dataset.

Using this method we collected 111,844 total records which we used to train our own IR2Perf network.

3.3.2. Training

We trained an tree ensemble (Random Forest) via the XGBoost algorithm in lieu of a deep neural network. We empirically found that the Random Forest model fit the data better than a deep regression network (MLP), which ended up having numerical issues. The tree ensemble utilizes multiple decision trees which vote together on the final prediction[10]. XGBoost is a method for using gradient boosting to train random forests[11]. We speculate on why tree ensembles may have performed better in section 5.

During training we trained with a leave-one-out validation strategy to assess the generalization of the model. During final deployment of the model to the RL agent training pipeline, we trained it on all available data.

Our training pipeline was as follows: (1) embed IR file using IR2Vec, (2) drop redundant data points, (3) normalize and scale the embedding vectors, and (4) fit the model with MSE loss using the precomputed $Func_{speedup}$ as a target variable. We skip the PCA step performed by Ashouri et al. because we empirically did not find it beneficial. We speculate this is because, even if the data is more sparse without PCA, the main benefit we derive from our tree ensemble is its ability to overfit. The additional features likely help individual trees learn subsets of the overall dataset, which helps the overall fit of the model to the data.

3.4. Reinforcement Learning Agent

We used the same training scheme proposed by Ashouri et al., which modified the PPO algorithm used by MLGO to use the IR2Perf network to generate rewards during execution (Ashouri et al., 2022). The MLGO training pipeline was otherwise left unmodified.

We trained the RL agent for 3000 iterations. Each iteration the model collects several pre-inlining IR samples from the corpus. The current policy is used to make inlining decisions on the IR and then a reward is generated using IR2Perf after embedding the IR immediately before it would be lowered to machine code using IR2Vec. The policy is trained on this subset for $num\_policy = 3000$ iterations using the current policy π_θ [12] according to algorithm 1.

4. Results

4.1. IR2Perf

The model failed to adequately fit the dataset when trained on the entire collected CppPerformanceBenchmarks corpus, as can be seen in figure 1. The random forest appears to predict around the mean of the distribution[13]. We take this as an indication that the network has failed to generalize and is predicting the mean as a means to reduce its MSE loss.

[8] https://www.brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html

[9] $\tau = 225$

[10] https://builtin.com/data-science/random-forest-algorithm

[11] https://xgboost.readthedocs.io/en/stable/tutorials/rf.html

[12] A detailed set of hyperparameters can be seen at https://github.com/TrainingAfternoon/opti-ml/blob/main/compiler_opt/rl/inlining/gin_configs/ppo_nn_agent.gin

[13] In fact it does: $\bar{y} = 0.8755$, $\bar{\hat{y}} = 0.8772$

Algorithm 1 Training Inliner RL Model using IR2Perf

procedure FunctionSpeedup
for Function f in module **do**
$FTs \leftarrow getFunctionFeatures()$
$funcReward \leftarrow infer(FTs)$
$totalReward \leftarrow append(funcReward)$
end for
return $totalReward$
end procedure

procedure CallsiteInline
initialize policy π_γ randomly
for iteration i in Training **do**
$s_i \leftarrow Sample_{\mathcal{N}_{(0,1)}}(TrainingData)$
Compile and Get IR with policy $\pi_{\gamma+\sigma_i}$
$R \leftarrow FunctionSpeedup(Module)$
Update policy θ using equation 2
end for
end procedure

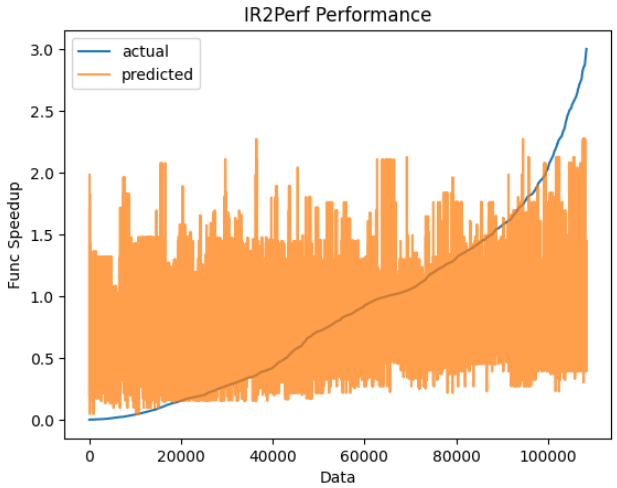

Figure 1. The fit of the random forest model to entire CppPerformanceBenchmarks corpus. $R^2 = 0.1367$

4.2. Evaluating RL Agent

We evaluated the default inlining heuristic, v1.0 of the MLGO inlining-for-size policy[14], and the inlining-for-speed policy we trained. The results of each policy on the training set are included in table 1 and the results of each policy on the testing set are included in table 2.

We observe that MLGO and our policy tend to do slightly worse, or the same as, the default inlining heuristic. In a few instances, it does significantly worse[15], though these examples have high variance potentially indicating a noisy test. We find this unsurprising for MLGO as MLGO was trained for inlining-for-size, not inlining-for-speed, so we would not expect a significant runtime difference with the default policy. However for our policy, we take the results as evidence that the agent was unable to learn and did not deviate from its learned baseline of imitating the default heuristic. We credit this failure to the failure of our IR2Perf network to adequately learn to predict runtime improvements. We discuss our thoughts more in the following section.

Table 1. cpp-perf-benchmarks results

file	agent	mean runtime (s)	variance
locales	base	103.6343	0.3870
	mlgo	104.1613	0.1510
	policy	104.3913	0.7208
loop removal	base	240.0767	0.0016
	mlgo	240.0443	0.0001
	policy	240.0720	0.0002
rotate bits	base	240.0780	0.0004
	mlgo	240.0523	0.0001
	policy	240.0767	0.0007
scalar replacement structs	base	240.0680	0.0001
	mlgo	240.0547	0.0002
	policy	240.0723	0.0001

[14] https://github.com/google/ml-compiler-opt/releases/tag/inlining-Oz-v1.0

[15] Such as security_rijndae1_e in table 2

Table 2. cBench results

file	agent	mean runtime (s)	variance
automotive_bitcount	base	2.8723	0.0004
	mlgo	1.4760	0.0001
	policy	1.4807	0.0002
automotive_qsort1	base	4.1610	0.0003
	mlgo	4.1087	0.0032
	policy	4.1010	0.0038
automotive_susan_c	base	4.0607	0.0019
	mlgo	4.7107	0.3834
	policy	4.6183	0.2083
automotive_susan_e	base	3.8733	0.0019
	mlgo	3.5863	0.0865
	policy	3.8360	0.2206
automotive_susan_s	base	2.9650	0.0001
	mlgo	3.3570	0.0605
	policy	3.3540	0.0290
bzip2d	base	4.8217	0.0231
	mlgo	10.4940	13.6295
	policy	11.0360	29.5467
bzip2e	base	3.8000	0.0018
	mlgo	5.2297	0.0222
	policy	5.0063	0.1599
consumer_jpeg_c	base	5.2367	0.0121
	mlgo	5.5857	0.0899
	policy	5.7590	0.5314
consumer_jpeg_d	base	10.9033	0.8649
	mlgo	14.2463	23.3346
	policy	13.7590	2.0387
consumer_lame	base	3.9433	0.0079
	mlgo	3.3437	0.5989
	policy	3.2883	0.4415
consumer_tiff2bw	base	6.9603	0.0708
	mlgo	9.3600	1.7578
	policy	8.0513	1.5820
consumer_tiff2rgba	base	10.8903	33.4148
	mlgo	13.3303	0.2384
	policy	12.0370	1.7586
consumer_tiffdither	base	6.6327	0.0582
	mlgo	7.5237	0.3659
	policy	7.2720	0.0085
consumer_tiffmedian	base	4.4033	0.0125
	mlgo	4.7023	0.0420
	policy	4.5587	0.0124
network_dijkstra	base	0.5150	0.0003
	mlgo	0.5207	0.0000
	policy	0.5160	0.0001
network_patricia	base	2.2740	0.0137
	mlgo	2.4783	0.0050
	policy	2.4427	0.0024
office_stringsearch1	base	3.0263	0.0002
	mlgo	2.9970	0.0000
	policy	2.9943	0.0000
security_blowfish_d	base	7.7403	0.0000
	mlgo	7.4630	0.0001
	policy	7.4567	0.0000
security_blowfish_e	base	7.7123	0.0001
	mlgo	7.5580	0.0001
	policy	7.5590	0.0000
security_rijndael_d	base	29.9463	0.7761
	mlgo	41.2183	1.0895
	policy	41.0523	1.3160
security_rijndael_e	base	29.1957	0.1716
	mlgo	41.5253	2.0253
	policy	41.0150	6.7436
security_sha	base	6.2483	0.0001
	mlgo	5.5697	0.0005
	policy	5.5600	0.0004
telecom_CRC32	base	2.4420	0.0001
	mlgo	2.4493	0.0002
	policy	2.4480	0.0027
telecom_adpcm_c	base	3.9727	0.0000
	mlgo	4.0133	0.0000
	policy	4.0170	0.0000
telecom_adpcm_d	base	3.8327	0.0000
	mlgo	3.6910	0.0000
	policy	3.6947	0.0003

5. Discussion

5.1. IR2Perf

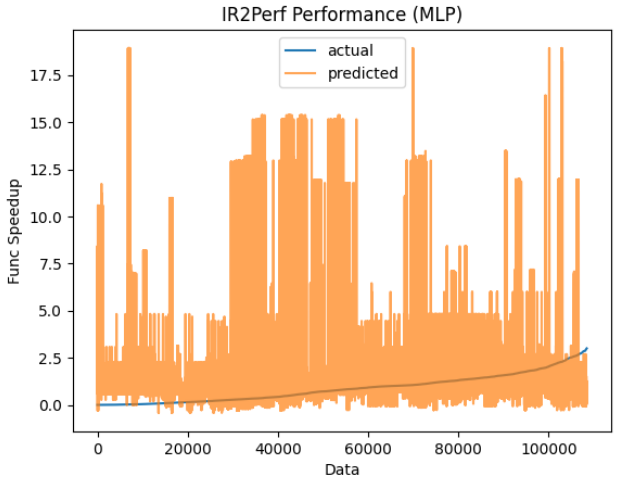

Figure 2. The fit of the best MLP model to entire CppPerformanceBenchmarks corpus. $R^2 = -6.7169$

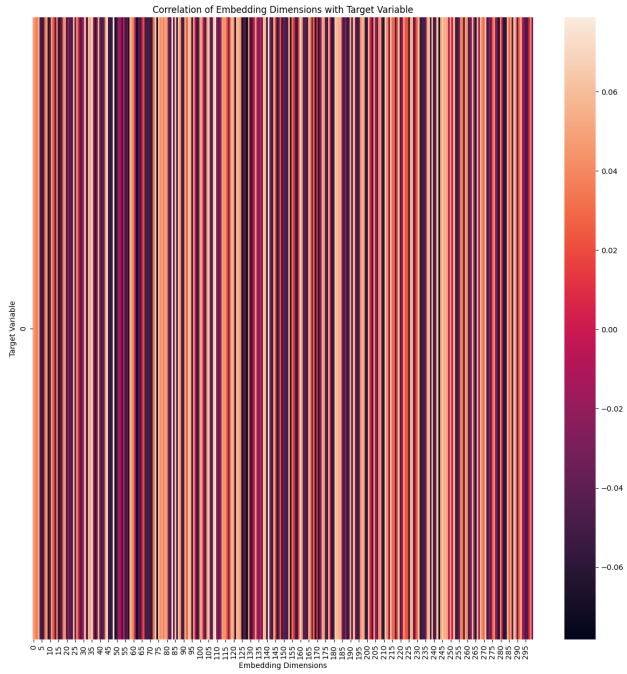

Figure 3. The Pearson's correlation of each embedding dimension to our target variable, computed over our entire dataset X

While the tree ensemble did not perform well on the data, it did outperform the best MLP model that we trained (see figure 2). We suspect that this is because MLPs try to create a linear combination of the features at each point in the network, and we found each feature of the embedding space has a very low correlation with our target variable of $Func_{speedup}$ (see fig 3). This implies that there is no "pattern" for the MLP to pick up on, hence its lackluster performance.

While the tree ensemble is given the same weakly correlated data, we theorize it is better able to overfit to the data because each tree in the ensemble is learning a set of decision boundaries over the features to place samples. With a large number of trees, we suspect subsets of the trees are learning subsets of the data which may exhibit more of a learnble pattern even if the over all collection is seemingly random.

5.2. Future Work

We have several directions we would like to take this work next. While we observed worse-than-before results from adding IR2Vec to the IR2Perf model, the use of embedding models to represent IR when trying to predict program runtime still seems promising. We could investigate the creation of a custom embedding model for IR2Perf. More work tuning and experimenting with the IR2Perf architecture could also be done.

Another improvement could be the introduction of partial rewards to the policy gradient algorithm described in equation 1. MLGO opted to approximate partial rewards with the total reward R because of the challenges with generating a partial reward for each inlining decision, but this is a deviation from the proper algorithm. Using IR2Perf we should be able to generate partial rewards by running the IR through the model after each inlining decision instead of after all decisions have been made.

6. Conclusion

While we failed to train a successful inlining-for-speed agent, we *did* succeed in creating an end-to-end, openly available architecture for collecting training data, training the IR2Perf network, and integrating it with the MLGO infrastructure. Hopefully this will provide a platform that future research in the space can take advantge of.

Additionally, while it is a negative result, we have found that there is most likely an issue with (1) not using an autotuner to generate data and/or (2) using IR2Vec embedding vectors are predictors of runtime performance, which can also hopefully guide future work in this space.

Acknowledgements

Thank you to the members of the Kedziora Research Lab for feedback and suggestions through out the process: Jonny Keane, Dr. Jeremy Kedziora, and Salvin Chowdhury.

Thank you to the members of the capstone committee for supervising this work: Dr. John Bukwoy, Dr. Rob Hasker,

and Dr. Jeremy Kedziora.

Finally, thank you to Dr. RJ Nowling for providing the computing platform used during this work.

References

Ashouri, A. H., Elhoushi, M., Hua, Y., Wang, X., Manzoor, M. A., Chan, B., and Gao, Y. Work-in-progress: Mlgoperf: An ml guided inliner to optimize performance. In *2022 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. IEEE, October 2022. doi: 10.1109/cases55004.2022.00008. URL http://dx.doi.org/10.1109/CASES55004.2022.00008.

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf.

Cox, C. Cppperformancebenchmarks. https://gitlab.com/chriscox/CppPerformanceBenchmarks, 2018.

Fursin, G. cbench. https://sourceforge.net/projects/cbenchmark, 2010.

Gregg, B. Perf examples. URL https://www.brendangregg.com/perf.html.

Henning, J. L. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, sep 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL https://doi.org/10.1145/1186736.1186737.

Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space, 2013.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In Solla, S., Leen, T., and Müller, K. (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

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

Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., and Li, D. MLGO: a machine learning guided compiler optimizations framework. *CoRR*, abs/2101.04808, 2021. URL https://arxiv.org/abs/2101.04808.

VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M. S., Upadrasta, R., and Srikant, Y. N. Ir2v ¡scp¿ec¡/scp¿: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization*, 17(4):1–27, December 2020. ISSN 1544-3973. doi: 10.1145/3418463. URL http://dx.doi.org/10.1145/3418463.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.