LEARNING TO SUPEROPTIMIZE REAL-WORLD PROGRAMS

Alex Shypula
Massachusetts Institute of Technology *
CSAIL
shypula@mit.edu

Pengcheng Yin
Carnegie Mellon University †
Language Technologies Institute
pengchey@alumni.cmu.edu

Jeremy Lacomis
Carnegie Mellon University
Institute for Software Research
jlacomis@cs.cmu.edu

Claire Le Goues
Carnegie Mellon University
Institute for Software Research
clegoues@cs.cmu.edu

Edward Schwartz
Carnegie Mellon University
Software Engineering Institute
eschwartz@cert.org

Graham Neubig
Carnegie Mellon University
Language Technologies Institute
gneubig@cs.cmu.edu

ABSTRACT

Program optimization is the process of modifying software to execute more efficiently. *Superoptimizers* attempt to automatically outperform optimizing compilers by employing significantly more expensive search and constraint solving techniques to generate more efficient code. Generally, these methods do not scale well to programs in real development scenarios, and as a result superoptimization has largely been confined to small-scale, domain-specific, and/or synthetic program benchmarks. In this paper, we propose a framework to learn to superoptimize real-world programs by using neural sequence-to-sequence models. We created a dataset consisting of over 25K real-world x86-64 assembly functions mined from open-source projects and propose an approach, **S**elf **I**mitation **L**earning for **O**ptimization (**SILO**) that is easy to implement and outperforms a standard policy gradient learning approach on our dataset. Our method, SILO, superoptimizes 5.9% of our test set when compared with the `gcc` version 10.3 compiler's aggressive optimization level `-O3`. We also report that SILO's rate of superoptimization on our test set is over five times that of a standard policy gradient approach and a model pre-trained on compiler optimization demonstration.

1 INTRODUCTION

Program optimization is a classical problem in computer science that has existed for over 50 years McKeeman (1965); Allen & Cocke (1971). The standard tool for generating efficient programs is an *optimizing compiler* that not only converts human-written programs into executable machine code, but also performs a number of semantics-preserving code transformations to increase speed, reduce energy consumption, or improve memory footprint (Aho et al., 2006). Most optimizing compilers use heuristic-based optimizations. These optimizing transformations generally need to be written by experts for an individual compiler, and are applied to an intermediate representation of the code

*Work completed while at Carnegie Mellon University.
†Now at Google Research.

produced while transforming high-level code into executable machine code. In an effort to automatically create optimized programs that surpass human-defined heuristics, the research community has pioneered automated optimization methods, or "superoptimizers." These superoptimizers may outperform compiler-based optimizations, but they can be difficult to build for new intermediate representations or languages and difficult to employ in practice, especially on longer programs. Research in machine learning-based program optimization remains relatively under-explored, especially in light of the progress deep learning models have made in reasoning about and generating code for a variety of tasks Allamanis et al. (2018); Chen et al. (2021).

In this work, we investigate the ability of deep neural networks to optimize real-world programs mined from Github. For this, we created **Big Assembly**, a dataset consisting of over 25K functions in x86-64 assembly mined from online open-source projects from Github, which enabled our experiments on the data-driven and learning-based optimization of real-world programs. We also propose an easy to implement algorithm **S**elf **I**mitation **L**earning for **O**ptimization (**SILO**) that progressively improves its superoptimization ability with training. Our results indicate that it superoptimizes 5.9% of our test set beyond `gcc -O3`, over five times the rate of a model pre-trained on the outputs of an optimizing compiler as well as a model fine-tuned with policy gradient methods. Instead of focusing on a customized search method unique to a language's implementation and semantics, our methodology relies on a dataset of demonstrations for pre-training as well as a test case generator, a sandbox for executing programs, and a method for verifying program equivalence.

2 Problem Formulation

The superoptimization task is, when given a specification for a program S and a reference program F_{ref} (which meets the specification), to generate an optimized program F_o that runs more efficiently and is equivalent to the reference program F_{ref}. In this paper we focus on program optimization at the assembly level, so S, F_{ref}, and F_o are all programs written in X86-64 assembly code. S is a program with no optimizations applied at all and its purpose is to demonstrate desired program semantics. F_{ref} is an assembly program produced by the optimizing compiler `gcc` at its aggressive `-O3` optimization level.

Specifically, our goal is to learn a model $f_\theta : \mathrm{S} \mapsto \mathrm{F}_o$ such that a model-generated (optimized) program $\hat{\mathrm{F}}_o$ attains lower cost, and ideally minimal cost, under a cost function $\mathcal{C}(\cdot)$ evaluated on a suite of K input-output test cases $\{IO\}_{k=1}^{K}$ (for example energy consumption or runtime). Here, I represents the hardware state prior to executing the program (i.e., input) and O represents the hardware state after executing the program (i.e., output). This model-generated program must meet the specification, which is determined by a verification function $\mathcal{V}(\cdot) \in \{0, 1\}$. More details about $\mathcal{C}$ and $\mathcal{V}$ are located in Section 5. The learned model's objective is then to produce rewrites that meet the condition:

$$\mathcal{C}\Big(\hat{\mathrm{F}}_o; \{IO_k\}_{k=1}^{k}\Big) \; < \; \mathcal{C}\Big(\mathrm{F}_{ref}; \{IO_k\}_{k=1}^{K}\Big) \;\; s.t. \;\; \mathcal{V}(\,\hat{\mathrm{F}}_o, \mathrm{S}) \; = 0 \tag{1}$$

In order to train our model on some of the optimizations that are present in modern compilers in a supervised manner and to improve it by learning from experience, a dataset is necessary. Our training set D_o therefore consists of N tuples of (1) an I/O test suite $\{IO_k\}_{k=1}^{K}$, (2) a compiled and unoptimized program specification S, (3) and a compiled and aggressively optimized program F_{ref}:

$$D_o = \left\{ \Big(\; \{IO_k^i\}_{k=1\ldots K}, \; \mathrm{S}^i, \; \mathrm{F}_{ref}^i, \; \Big) \right\}_{i=1\ldots N} \tag{2}$$

In Section 4 we explain our methodology for learning to superoptimize programs; before this, however, we first introduce our dataset for optimizing real-world programs in Section 3.

3 Big Assembly, a Dataset Mined from from Real World Code

There is no standard benchmark for evaluating superoptimization research. Some researchers have evaluated on randomly-generated programs in a simplified domain specific language Chen & Tian

Table 1: A list of program optimization benchmarks from machine learning and programming languages / systems works. The three criteria for evaluating listed are the number of individual examples in the benchmark (Sz.), are the programs written by humans (H.), are the programs found "in the wild" (i.e. in open source projects) (R.W.), and does the benchmark contain either branching or control flow (CTL.)

DATASET	SZ	H.	R.W.	CTL.
SHI ET AL. (2020)	12,000	✗	✗	✗
GULWANI ET AL. (2011)	25	✓	✗	✗
CHURCHILL ET AL. (2017)	13	✓	✓	✓
OURS	25,141	✓	✓	✓

(2019); Shi et al. (2020), while others have tested on small and hand-picked programs Joshi et al. (2002); Gulwani et al. (2011); Churchill et al. (2017). While these datasets are sufficient for demonstrating methodological capabilities, they do not necessarily reflect the properties of real code, and thus do not predict their performance relative to modern optimizing compilers on real-world programs. Lastly, small-scale benchmarks are insufficient for data-hungry modern deep learning.

We created a dataset, which we will refer to as **Big Assembly**, consisting of of 25,141 functions in x86-64 assembly code collected by using `gcc` to compile programs both with (`-O3`) and without (`-O0`) aggressive optimizations. We started by collecting 1.61 million functions from open source projects on Github that were written in C. Of these 1.61M, we were able to mine testcases for a dataset of over 100,000 functions. Given this suite of executable functions with testcases, for each function, we then needed to determine the live out registers: the subset of CPU registers we rely on for testing program equivalence. We performed two stages of analysis and sanity checks to compute a conservative approximation of the live out registers. The first involved using SMT solvers to mine the live out set by finding the set of CPU registers under which the `gcc -O0` function used as the specification S and the `gcc -O3` function used as F_{ref} were equivalent. To avoid an overly-permissive live out set, we also filtered out trivial examples, such as functions equivalent to `return 0` under the live out set. This first pass reduced our dataset to 77,813 functions. We then performed a similar pass, but instead using the test case suite of randomly generated inputs and their corresponding outputs $\{IO\}_{k=1}^{K}$, reducing our total dataset to 25,141 functions (19,819 train, 2,193 dev, 3129 test). For verification, test case generation, and program instrumentation, we used artifacts from the STOKE[1] project with additional modifications.

Table 1 compares the basic properties of our dataset to those from existing work; beyond its size, our dataset is notable because it consists entirely of functions mined from real world codebases. It also contains examples with more complex operations such as SIMD instructions, branching, and loops. Additional details on how the dataset was collected are available in the supplementary materials section.

4 LEARNING PROGRAM OPTIMIZATIONS

NEURAL PROGRAM OPTIMIZER

Our program optimization model f_θ is a neural sequence-to-sequence network, where the input specification (unoptimized program) and output (optimized program) are represented as sequences of tokens. Specifically, f_θ is parameterized with a standard Transformer-based encoder-decoder model (Vaswani et al., 2017).

LEARNING ALGORITHMS

For learning to optimize, we develop a two-stage learning approach. First, in a *pre-training* stage, to capture commonly-used optimization heuristics adopted by existing optimizing compilers, we use supervised learning to train the model on the mined corpus D_o of `gcc`-optimized programs (E.q. 2)

[1] https://github.com/StanfordPL/stoke

described in Section 3. Next, to discover more efficient optimization strategies, we investigate fine-tuning using policy-gradient methods and a propose an iterative learning approach, SILO.

Policy Gradient Approach As in Eq. (1), our goal is to synthesize a correct program $\hat{\mathrm{F}}_o$ verified by $\mathcal{V}$ that outperforms a reference program F_{ref} on the cost function $\mathcal{C}$. An intuitive choice is to use policy gradient methods to learn a policy that directly minimizes our cost function $\mathcal{C}$ and produces correct programs under $\mathcal{V}$ in expectation. Specifically, we express this dual objective via the Lagrangian relaxation:

$$\mathbf{J}(\hat{\mathrm{F}}_o) = \mathcal{C}\Big(\hat{\mathrm{F}}_o; \{IO_k\}_{k=1}^{k}\Big) + \lambda\, \mathcal{V}(\,\hat{\mathrm{F}}_o, \mathrm{S}). \tag{3}$$

A commonly used policy gradient approach is REINFORCE with baseline Williams (1992). Based on our minimization objective, we can express the loss using the following equation, where $b(\mathrm{S})$ is a baseline value for the given specification and $p(a_t|a_{<t}; \mathrm{S})$ is the model-given probability for generating a token at time step t for sequence $\hat{\mathrm{F}}_o$. In a traditional reinforcement learning context one might seek to perform gradient ascent on the following term; however, because we are trying to minimize the objective function, we perform gradient descent instead.

$$\mathcal{L} = \sum_{t=1}^{T} \log p(a_t|a_{<t}; \mathrm{S})\, \big(\mathbf{J}(\hat{\mathrm{F}}_o) - b(\mathrm{S})\big) \tag{4}$$

Self Imitation Learning for Optimization (SILO) Algorithm 1 illustrates our SILO learning approach. It consists of two steps, an exploration step (lines 4-11) where the model seeks to discover alternative optimizations that are more efficient than the compiler generated targets used in pre-training, and a learning step (lines 12-13), where the model parameters are updated using newly discovered optimized programs. First, in the exploration step, an exploration batch B_{ex} is sampled from the dataset D initialized with program specifications (in our case, unoptimized assembly programs) S^i and their compiler-optimized outputs F^i_{ref}. For each input specification S^i in B_{ex}, we sample a model-predicted optimization $\hat{\mathrm{F}}^i_o$, execute $\hat{\mathrm{F}}^i_o$ on the I/O test suite $\{IO\}_{k=1}^{K}$ to compute the cost function $\mathcal{C}$, and assess the verification function $\mathcal{V}$ on $\hat{\mathrm{F}}^i_o$ and S. If any of the new samples are both functionally equivalent by our verification function $\mathcal{V}$ and also achieve a lower cost under $\mathcal{C}$ compared to the compiler-optimized targets in the original dataset, the compiler-optimized target in the dataset is then replaced with the model's newly-discovered optimal rewrite. After the exploration step is taken, in the learning step, a separate training batch B_{tr} is sampled for maximum-likelihood training from the dataset which may now contain model-optimized targets.

Self-imitation learning Oh et al. (2018) is an off-policy reinforcement learning algorithm intended to help agents solve challenging exploration problems by learning from good past actions. Besides the ordinary on-policy reinforcement learning Monte-Carlo model-sampled actions, the model is also trained on historical states s and actions a that achieve high rewards R using the following off-policy imitation learning loss:

$$\mathcal{L}_{\mathrm{sil}} = -\log p(a|s)\ \max\big((R - V_\theta(s)), 0\big) \tag{5}$$

where each sample $\langle a, s\rangle$ is weighted by how much better the off-policy return was compared to the learned baseline $V_\theta(s)$. Our algorithm, SILO, has a few differences with standard self-imitation learning. First, for sequences that outperform F_{ref} we omit a learned value function and train on the entire sequence using cross-entropy loss, as opposed to individual actions. Additionally, we do not interpolate our loss with an on-policy reinforcement learning algorithm. Rather, we avoid the policy gradient altogether, and instead train only on the best sequence found so far in our dataset, be it the compiler-optimized outputs F_{ref} or a sequence discovered that outperforms it F_o. This is also broadly related to the "hard" EM algorithm that uses the best model-predicted results as optimization targets Kearns et al. (1998).

Actor-Learner Architecture, Model Configuration, and Training

One hurdle to performing program optimization at scale is that the time required to evaluate $\mathcal{C}$ and $\mathcal{V}$ is costly, limiting the model's throughput of learning examples. To alleviate this bottleneck, we

Algorithm 1: SILO for Program Optimization

Initialize model f parameters θ from pre-trained model
Initialize dataset of program function pairs and test cases:
$D = D_o = \left\{ \left(\{IO^i\}_{k=1}^{K},\ \mathrm{S}^i,\ \mathrm{F}^i_{ref},\ \right) \right\}_{i=1...N}$
while budget not exhausted **do**
  Sample a batch B_{ex} from D_o
  for $\left(\{IO^i\}_{k=1}^{K},\ \mathrm{S}^i,\ \mathrm{F}^i_{ref},\ \right)$ in B_{ex} **do**
    sample $\hat{\mathrm{F}}^i_o \sim f_\theta(\mathrm{S}^i)$
    calculate $\mathcal{C}(\hat{\mathrm{F}}^i_o)$, and $\mathcal{V}(\hat{\mathrm{F}}^i_o, \mathrm{S}^i)$
    if $\mathcal{V}(\hat{\mathrm{F}}^i_o, \mathrm{S}^i) = 0$ and $\mathcal{C}(\hat{\mathrm{F}}^i_o) < \mathcal{C}(\mathrm{F}^i_{ref})$ **then**
      Replace F^i_{ref} with sample $\hat{\mathrm{F}}^i_o$ in D
    end if
  end for
  Sample a batch B_{tr} from D
  Update θ via supervised learning on B_{tr} from D
end while

utilize an actor-learner set up (Liang et al., 2018; Espeholt et al., 2018). Additional details on our actor-learner set up are provided in the appendix.

Our neural superoptimizer f_θ uses a 3-layer transformer encoder-decoder with embedding dimension of 512 with 8 attention-heads. We utilize the Adam optimizer Kingma & Ba (2014) with the inverse square root schedule from Vaswani et al. (2017). We pre-trained the model for 88K steps and subsequently performed our fine-tuning algorithms for an additional 5K steps. We additionally use `SentencePiece`[2] to pre-process the assembly with byte-pair encoding (Sennrich et al., 2015). Additional details on training hyperparameters and data preprocessing are located in our supplementary materials section.

5 Evaluation

With our final model, we test our methods by generating 10 model-optimized candidates through beam search. We then calculate $\mathcal{C}$ and $\mathcal{V}$ for each and report results based on the best result of all 10 candidate programs.

A primary concern in constructing the verification function $\mathcal{V}$ in Eq. (1) is the undecidability of program equivalence for programs with control flow such as loops. If using testcases for equivalence $\mathcal{V}$ as well as the cost function $\mathcal{C}$ challenges also lie in utilizing a testcase suite $\{IO\}_{k=1}^{K}$ for either benchmarking a program's performance or coming up with an approximate estimate for performance.

Measuring Program Correctness

A key claim of this work is that our superoptimizer outputs correct programs — that is, programs that are more efficient, but still semantically equivalent, to the input programs. Program equivalence is undecidable in the general case, motivating a complementary set of mechanisms for verifying output program correctness. In our experiments, we confirm output program correctness for our verification function $\mathcal{V}$ in two ways. First, we run synthesized programs on the provided test cases. Second, we formally verify correctness using two solver based verifiers: the bounded SMT solver based verifier from the standard STOKE project artifacts, and an additional verifier available from the artifacts in the program verification work in Churchill et al. (2019). These program verifiers are based on the state-of-the-art Z3 SMT solver De Moura & Bjørner (2008); SMT ("Satisfiability Modulo Theories") solvers decide the satisfiability of boolean formulas with respect to a set of underlying theories (such as the theory of integer arithmetic).

[2] https://github.com/google/sentencepiece

We use both test cases and the two verifiers for several reasons. High coverage test suites are informative in terms of program correctness, but intrinsically incomplete. Meanwhile, verifiers do not always scale, especially to programs with arbitrary numbers of loop iterations to ensure termination. As is standard we configured a maximum bound of b (set to 4) loop iterations, and the set a maximum timeout for verifications taking over T seconds (set to 150). Verification is thus also incomplete past those bounds, and limited by the correctness of the verifiers' underlying models.

Indeed, we manually observed cases where our fine-tuning methods exploited either gaps in test suite coverage, or bugs in the verifiers' models of x86-64 semantics. Motivated by this, we use both testcases and the two verifiers for additional robustness. While this would intuitively help mitigate spuriousness during evaluation, it could still remain an issue when evaluating optimization of open-domain programs. As we later explain in Section 7, we also resort to human verification to get reliable results when reporting model performance on test sets.

MEASURING PROGRAM PERFORMANCE

We follow previous work on superoptimization of x86-64 assembly and primarily calculate the cost function $\mathcal{C}$ (E.q. 1) as a static heuristic approximation of expected CPU-clock cycles. We compute the sum of both performance cost functions from Schkufza et al. (2013) and Churchill et al. (2017). The former is a sum of all expected latencies for all instructions in the assembly function (denoted as $\mathcal{C}_{\text{all}}$), while the latter computes expected latencies only using executed instructions ($\mathcal{C}_{\text{exe}}$) from the randomly generated test suite $\{IO\}_{k=1}^{K}$. $\mathcal{C}_{\text{exe}}$ is a better approximation, especially for functions that contain loop constructs, while $\mathcal{C}_{\text{all}}$ may additionally penalize redundant instructions that are not executed. Expected latencies were calculated by the authors of STOKE for the Intel Haswell architecture by benchmarking and measuring CPU-clock cycles over a suite of x86-64 instructions.

6 APPLICATION TO HACKER'S DELIGHT

We apply our methods to the 25 functions chosen from the HACKER'S DELIGHT benchmark Warren (2002), first used in Gulwani et al. (2011) for program synthesis and later in Schkufza et al. (2013) for x86-64 program superoptimization. In the latter work, authors express they were able to either match or outperform `gcc -O3` when provided programs compiled with `LLVM -O0`. The superoptimization benchmark consists of bit-vector manipulation challenges such as "take the absolute value of x." Before evaluating on our large scale Big Assembly dataset in Section 7, we perform controlled experiments on HACKER'S DELIGHT allowing for interpretable optimizations and consistency with prior work.

```
.absolute_value:
  movl %edi, %eax
  sarl $0x1f, %eax
  xorl %eax, %edi
  subl %eax, %edi
  movl %edi, %eax
  retq
```

(a) -O3 code

```
.absolute_value:
  movl %edi, %eax
  sarl $0x1f, %edi
  xorl %edi, %eax
  subl %edi, %eax
  retq
```

(b) Model-optimized code

Figure 1: An example of the absolute value function from HACKER'S DELIGHT optimized by `gcc -O3` on the left and the trained models on the right. For the right example, the same output was witnessed in the results of both fine-tuning experiments. The model-optimized code demonstrates superior register allocation.

Results We examined the results of REINFORCE and SILO with respect to two quantities: (1) the number of programs where a superoptimized version was found at least once during the training process, and (2) the number of programs for which a superoptimized version was found within the top-10 hypotheses generated by beam search from the last model at the end of training. Regarding the former metric, REINFORCE and SILO respectively found 3 and 2 superoptimized programs during the training. For the latter metric, the final models produced by REINFORCE and SILO output 1 and 2 superoptimized programs respectively. These results indicate that while the policy

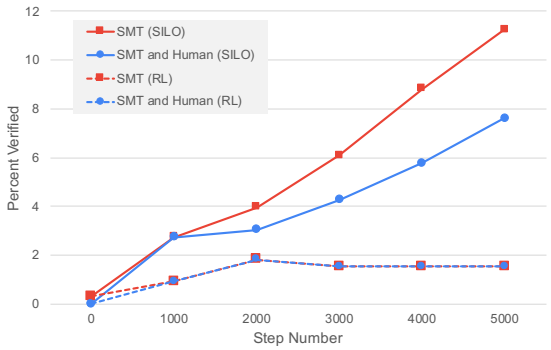

Figure 2: A plot reflecting the proportion of the validation set sub-sample population (329 programs) superoptimized every 1000 steps of training.

gradient methods may have a wider breadth for exploration during the training process compared to SILO, policy gradient methods may also be less stable in their final solutions.

7 Application to Big Assembly

In applying our methods to the Big Assembly dataset, we followed the same general experimental setup as HACKER'S DELIGHT; however, we evaluated our results on held-out sets. As mentioned in Section 5, we observed cases where our models exploited bugs in the verifiers' models of x86-64 semantics, thus we also incorporated manual human evaluation into our reporting methodology for the dataset.

For both learning methods, we chose the best model of the ones checkpointed every 1K steps during fine-tuning. We did this by choosing the model with the highest proportion of programs that were superoptimized on a randomly-sampled subset of 329 functions from the validation set according to our cost function $\mathcal{C}$, correct according to our verification methodology $\mathcal{V}$, and correct again by manual human evaluation.

Using the chosen model, we then evaluated performance on the test set by manually checking all reported superoptimizations for correctness [3]. In Table 2 we report (1) the proportion of the entire test set that was superoptimzied according to our automatic methods discussed in Section 5 as well as (2) the actual proportion that we manually verified to be correct.

Results We witness that our proposed algorithm SILO far outperformed REINFORCE with baseline on this task: on the test set, SILO superoptimized 5.9% of programs and REINFORCE with baseline superoptimized 0.9%. We also note that despite the fact we used two separate SMT based verifiers to prove correctness, our SILO approach was capable of finding and learning generalizable exploits in a manner that REINFORCE did not.

In our study of manually verifying assembly programs, we witnessed that across all earlier and later stages of training, the REINFORCE model consistently superoptimized the same 5 to 6 programs in the validation set; in other words, it did not seem to apply superoptimization patterns to any new programs or learn any new superoptimization patterns. In contrast, the SILO model consistently increased the number of programs it superoptimized in the held-out set over time: it seemed to broaden its capacity to apply patterns to new programs and simultaneously learn different strategies to superoptimize and even exploit the verifier; this trend is reflected in the Fig. 2 plot.

We hypothesize the large difference in performance can be attributed to a reward space that is both very sparse and noisy. It is sparse, because program superoptimziations are hard to find: while train-

[3] Two authors manually reviewed over 80% of all reported superoptimizations across all 10 beams. The additional ∼20% of optimizations were manually checked by only one author. The authors checked only for false-positives; therefore, no outputs that were automatically determined to be either incorrect or not optimal to `-O3` were reviewed.

Table 2: Test set results on the Big Assembly dataset comparing the pre-trained model, SILO and REINFORCE with baseline. The first column (SMT Ver.) reports the proportion of programs that beat the `gcc -O3` baseline and verify using our automated evaluation methods, while the second column (SMT + Human Ver.) reports our expected proportion of programs that additionally pass a human evaluation step.

Model	SMT Ver.	SMT + Human Ver.
PRE-TRAIN	1.2%	1.0%
REINFORCE	0.9%	0.9%
SILO	8.3%	5.9%

```
.popEntry.s:
  movl 0x4(%rdi), %eax
  subl $0x1, %eax
  movl %eax, 0x4(%rdi)
  cltq
  leaq (%rax,%rax,2), %rdx
  movq 0x8(%rdi), %rax
  leaq (%rax,%rdx,4), %rax
  retq
```

(a) -O3 code

```
.popEntry.s:
  subl $0x1, 0x4(%rdi)
  movq 0x8(%rdi), %rax
  movslq 0x4(%rdi), %rdx
  leaq (%rdx,%rdx,2), %rdx
  leaq (%rax,%rdx,4), %rax
  retq
```

(b) Model-optimized code

Figure 3: An example of a program from the Big Assembly dataset superoptimized with the SILO-trained model. Here a subtraction is performed in memory to eliminate the instructions performing the subtraction in `%rax` (in red) and storing it back in memory. This approach is followed by `movslq` instead of `cltq` along with modified register allocation to accommodate the changes.

ing, we saw that less than 1 in every 1,000 samples the REINFORCE model made were program superoptimizations. The reward space in this task is noisy, because a minuscule change in the output text can have an extreme impact on program semantics and syntactic correctness. We believe, without a method to re-learn from past experience, the on-policy REINFORCE algorithm struggles to find the signal in the noise.

8 RELATED WORK

Program Optimization The general undecidability of program equivalence means that there may always be room for improvement in optimizing programs (Rice, 1953). This is especially true as hardware options and performance goals become more diverse: what transformations are best for a scenario may vary greatly on performance objectives such as such as energy consumption or runtime or other factors.

State-of-the-art methods for superoptimization either rely on search-based procedures (Schkufza et al., 2013), or constraint-based methods (Sasnauskas et al., 2017). However, these methods have difficulty scaling to larger problems, and as a result, typically do not meet the performance requirements of real development scenarios at compile time.

Machine-Learning-Based Program Optimization Perhaps the closest work to ours is Shi et al. (2020) which attempted to learn symbolic expression simplification on a dataset of synthetically generated symbolic expressions in Halide by re-writing sub-trees of the parsed expression with reinforcement learning. The domain differs from ours; however, as the domain specific language contains simple expressions and randomly generated programs may contain redundancies not seen in assembly optimized by a compiler like `gcc`.

Another work that addressed automatic program optimization is Bunel et al. (2016). Unlike the Halide-based experiments, the work used reinforcement learning to learn a proposal distribution for stochastic search process used in Schkufza et al. (2013). While the learned proposal distribution showed improvements over the baseline, the method ultimately still used stochastic search, except

with improved search parameters. Unlike our work, the model is unable to fully control program transformations end-to-end.

9 CONCLUSION

In our work, we explored the task of program superoptimization with neural sequence models. Towards this goal, we utilized 1.61 million programs mined from open source projects on Github for pre-training along with and a subset of over 25K functions with testcases that can additionally be passed off to the SMT based verifiers in the STOKE project artifacts. We proposed SILO, a learning approach with a two step process (1) an exploration step to search for program superoptimizations, and (2) a learning step on the best sequences found during training. Our experiments on the Big Assembly dataset demonstate that SILO is able to outperform REINFORCE with baseline. We believe that REINFORCE struggles, because program superoptimzziation is a highly-challenging exploration task with a very sparse reward space. By incorporating supervision on superoptimized sequences, SILO is able to learn optimizations more effectively from its exploration.

Recently, large neural sequence models have been proposed as an effective method for program synthesis in high-level programming languages such as Python or C++ from natural language specifications Yin & Neubig (2017); Chen et al. (2021); Li et al. (2022); however, to our knowledge, relatively little work has been done to refine these models to go beyond synthesizing correct programs that meet a specification, and make additional considerations for important metrics such as performance or readability. Given the increased availability of executable program synthesis datasets, tuning neural sequence models to go beyond program synthesis and optimize for additional metrics is a promising direction for future work.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc., 2 edition, 2006.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Frances E. Allen and John Cocke. A catalogue of optimizing transformations. Technical report, IBM Thomas J. Watson Research Center, 1971.

Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787*, 2016.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, pp. 6281–6292, 2019.

Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. *ACM SIGPLAN Notices*, 52(4):313–326, 2017.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1027–1040, 2019.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *ACM SIGPLAN Notices*, 46(6):62–73, 2011.

Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices*, 37(5):304–314, 2002.

Michael Kearns, Yishay Mansour, and Andrew Y Ng. An information-theoretic analysis of hard and soft assignment methods for clustering. In *Learning in graphical models*, pp. 495–520. Springer, 1998.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, Feb 2022.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, pp. 9994–10006, 2018.

William M McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

Junhyuk Oh, Yijie Guo, Satinder Singh, and Honglak Lee. Self-imitation learning. *arXiv preprint arXiv:1806.05635*, 2018.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 1953.

Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422*, 2017.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Hui Shi, Yang Zhang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Deep symbolic superoptimization without human knowledge. In *International Conference on Learning Representations*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Henry S Warren. Hacker's delight, 2002.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 440–450, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL https://www.aclweb.org/anthology/P17-1041.

A ACTOR-LEARNER ARCHITECTURE

The actor-learner architecture used for training is as follows: before the training process begins, multiple actor threads inheriting the parameters of the parent learner are created. For every iteration, each of the actor threads independently samples a batch of program re-writes from the distribution of the inherited model.

After sampling a batch of re-writes, an attempt is made the evaluate the rewrites by sending them to an evaluation server. Inside the evaluation server, the programs are assembled and tested for correctness and performance to calculate $\mathcal{C}$ and $\mathcal{V}$; more details on evaluation are located in Section 5. The actor then sends the samples with their related cost and correctness information to the learner module for learning. Then, the actor attempts to synchronize its parameters by inheriting a new copy, if available, from the learner module. Fig. 4 contains an overall diagram for our entire system.

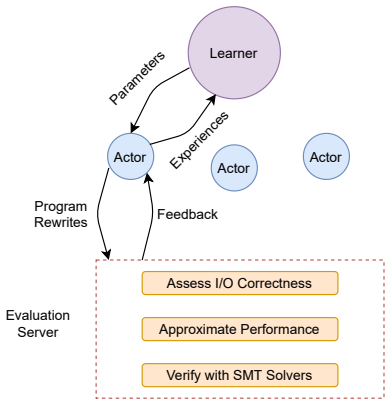

Figure 4: An overview of our actor-learner setup. It features interaction between a centralized learner module, numerous asynchronously-operating actors, as well as a server for evaluating program rewrites.

B SMT BASED VERIFIER EXPLOITS FOUND

Before the training process begins, multiple actor threads inheriting the parameters of the parent learner are created. For every iteration, each of the actor threads independently samples a batch of program re-writes from the distribution of the inherited model.

After sampling a batch of re-writes, an attempt is made the evaluate the rewrites by sending them to an evaluation server. Inside the evaluation server, the programs are assembled and tested for correctness and performance to calculate $\mathcal{C}$ and $\mathcal{V}$; more details on evaluation are located in Section 5. The actor then sends the samples with their related cost and correctness information to the learner module for learning. Then, the actor attempts to synchronize its parameters by inheriting a new copy, if available, from the learner module.

In our manual evaluation stage, we witnessed the primary pattern of exploiting the SMT solver based verifier was that of branch deleation. We present a concrete example of one such exploit paired with the verifier's output in Fig. 5. In the first subfigure we demonstrate the pattern exploiting the verifier from the original STOKE project artifacts[4], and in the second subfigure we show the output when running on the verifier included in the artifacts from a follow up work on program verification.[5] In this example, a comparison is done between the hex constant `0x2e` and the value located at the address in register `%rdi`; if the two are equal, the program jumps to location `.L1` executing a sequence of code to place a value in the `%eax` register conditional on multiple tests. If the constant is not equal to the value in memory, the jump is not taken, and the program returns with a 0 in the `%eax` register: this is because the very first line of the program `xorl %eax, %eax` zeros out the the `%eax` register.

[4]https://github.com/StanfordPL/stoke

[5]https://github.com/bchurchill/pldi19-equivalence-checker

```
Target                    Rewrite

.smtp_is_end.s:           .smtp_is_end.s:
xorl %eax, %eax           xorl %eax, %eax
cmpb $0x2e, (%rdi)        cmpb $0x2e, (%rdi)
je .L1                    je .L1
retq                      .L1:
.L1:                      retq
movzbl 0x1(%rdi), %edx
cmpb $0xd, %dl
sete %al
cmpb $0xa, %dl
sete %dl
orl %edx, %eax
movzbl %al, %eax
retq

Equivalent: yes
```

(a) Output from the original STOKE bounded verifier

```
Target                    Rewrite

.smtp_is_end.s:           .smtp_is_end.s:
xorl %eax, %eax           xorl %eax, %eax
cmpb $0x2e, (%rdi)        cmpb $0x2e, (%rdi)
je .L1                    je .L1
retq                      .L1:
.L1:                      retq
movzbl 0x1(%rdi), %edx
cmpb $0xd, %dl
sete %al
cmpb $0xa, %dl
sete %dl
orl %edx, %eax
movzbl %al, %eax
retq

[bv] Checking pair: 0 1 2 4; 0 1 2 3
Couldn't take short-circuit option without memory.
[bv] Paths 0 1 2 4 / 0 1 2 3
     verified: true
[bv] Checking pair: 0 1 2 4; 0 1 2 3
Couldn't take short-circuit option without memory.
[bv] Paths 0 1 2 4 / 0 1 2 3
     verified: true
[bv] Checking pair: 0 1 3 4; 0 1 2 3
We've finished early without modeling memory!
[bv] Paths 0 1 3 4 / 0 1 2 3
     verified: true
[bv] Checking pair: 0 1 3 4; 0 1 2 3
We've finished early without modeling memory!
[bv] Paths 0 1 3 4 / 0 1 2 3
     verified: true
Equivalent: yes
```

(b) Output from verifier in the artifacts from Churchill et al. (2019).

Figure 5: An example of the common exploit where the right hand side deletes the branch of code following `.L1`. If the third and fourth lines (`je .L1; .L1:`) are removed, then the verifier actually returns correctly.

Deletion of the branch following `.L1` is incorrect. We believe that the verifiers struggle with forms of branching and jump statements very often found in real-world programs. This is despite the fact that previous works the verifiers were used for included loops (which depend on jump statements) or branching. We found that when we removed the jump and location statement from the spurious rewrite, thereby preserving function semantics and eliminating all branching, both verifiers correctly identified that the two programs were not equivalent.

C ADDITIONAL INFORMATION ON THE BIG ASSEMBLY DATASET

Data Collection Our Big Assembly Dataset was mined from open source projects implemented in the C programming language on Github. Our programs were disassembled using `GNU` Binutils' objdump into x86-64 assembly, and split into individual functions. We performed the process twice on the same set of source code, so that we could create a parallel corpus of functions. We split our dataset into train, development, and test sets based at the level of entire individual github projects. We deduplicated by removing any overlapping binaries between the datasets. We also deduplicated at the individual function level by removing string matches after removing function names from assembly functions. For byte-pair encoding tokenization, we used a vocabulary size was 1029. Lastly, we removed programs pairs from the dataset if either the source or target program had length greater than 512 after byte-pair encoding tokenization.

Setting Live out and Filtering Spurious Examples As mentioned in Section 3 for our each function in our dataset, we were required to determine the *live out* set, the portion of the CPU state required for determining the equivalence between programs. We also determine *heap out*, which is a boolean flag that determines whether or not we should also check the heap for equivalence as well. We perform this sanity check by determining what parts of the CPU state are equivalent between the `gcc -O0` function and the `gcc -O3`. Pseudocode for how live out is described in Algorithm 2, in this algorithm we refer to the `gcc -O0` function as F_u.

In line 1, the algorithm begins by initializing live_out with all possible CPU registers. In lines 2-5, until either the computation budget is exhausted or the live out set reaches a fixed point, we iteratively execute the function *get_live_out* which incrementally determines the candidate live out set. It works by either executing the testcase suite or runs the SMT solver based verifier, analyzing any difference in the resulting CPU state, pruning any part of the CPU state that differs, and then returning the subset of the CPU state that may be equivalent between $\mathrm{F}_u$and F_{ref}. This may need to be run iteratively, because after pruning live out with respect to one counter example, it is possible another counterexample may still trigger a difference in other parts of the CPU state. For most of the general purpose CPU registers, we also perform this pruning at the sub-register level, allowing the register size for equivalence to be pruned down to the lower 32, 16, and 8 bits. If the computation budget is exhausted, the program returns early, and the program is discarded from the dataset.

After determining live out, in lines 6-10 an additional check is done to see if both programs are equivalent when checking the heap. If they are, heap out is set to true, and this information is recorded to be used for the fine-tuning phases. Lastly in lines 12-15, we perform a final sanity check to ensure that none of our programs are equivalent to a set of spurious programs such as a null program, a `return 0`, and a `return 1`. If a program is equivalent to one of these highly simplistic programs, it is discarded from our dataset.

Algorithm 2: Set Live Out and Filter Examples

1: Initialize: $live\_out$ = ALL_LIVE_OUT
2: **repeat**
3: $\quad old\_live\_out = live\_out$
4: $\quad live\_out = get\_live\_out(\mathrm{F}_u, \mathrm{F}_{ref}, live\_out)$
5: **until** $old\_live\_out \neq live\_out$ or budget exhausted
6: **if** $diff(\mathrm{F}_u, \mathrm{F}_{ref}, live\_out, heap\_out = True)$ **then**
7: $\quad heap\_out = False$
8: **else**
9: $\quad heap\_out = True$
10: **end if**
11: $is\_spurious = False$
12: **for** F_{spur} in SPURIOUS **do**
13: $\quad$ **if** $\neg diff(\mathrm{F}_{spur}, \mathrm{F}_{ref}, live\_out, heap\_out)$ **then**
14: $\quad\quad is\_spurious = True$
15: $\quad$ **end if**
16: **end for**

Data Preprocessing for Training We perform additional processing on our programs that we feed into the model to remove noise; we do this for the `gcc -O0` function S as well as the `gcc -O3` functions used for pre-training F_{ref}. x86-64 assembly often uses GOTO-like instructions to jump to different parts of a binary: this is one way that control flow is implemented. The jump targets are often marked as offsets in the binary itself; however, at the individual function level these may be canonicalized with ordinal locations (i.e. .L1, .L2, and so on). This fully preserves function semantics while removing noise from the prediction task.

D HYPERPARAMETERS AND SETTINGS

In this section we report the hyperparameters used for fine-tuning our models.

General Hyperparameters We found that our SILO algorithm did not need hyperparameter fine-tuning; whereas, our REINFORCE with baseline experiments were more brittle. We witnessed that without a lower learning rate and a carefully tuned learning rate schedule, our REINFORCE experiments would very often diverge before the full fine-tuning budget was exhausted. For all fine-tuning, we used 2,000 steps warmup. For the SILO experiments, we utilized a constant factor of .50 applied to the "noam scheduler" from Vaswani et al. (2017). For the REINFORCE models, we used a factor of .01 for the Big Assembly dataset and a factor of .0025 for the HACKER'S DELIGHT dataset.

REINFORCE Hyperparameters For our baseline in Eq. (3), we used a mean of the objective function for the previous 256 samples for each unique program. After subtracting the mean from the return, we then subsequently normalized by the standard deviation of the objective function of the previous 256 samples. For the lagrangian multiplier λ in Eq. (3), we used a penalty of 50,000. Additionally, we follow Schkufza et al. (2013) in adding an additional penalty of 100 for every bit in the CPU state that differed between the reference implementations and the rewrite output such that functions with similar semantics would be penalized less than those with dramatically different semantics. To prevent our objective function from growing too great, we also clipped the maximum cost so it would not exceed 100,000. As is typical in many policy gradient algorithms, we also included an entropy bonus β to encourage additional exploration: we used a constant entropy bonus of $\beta = 0.01$ for both our HACKER'S DELIGHT and Big Assembly experiments.