O4: Learning a Dynamic Phase Ordering Policy

Ajay Uppili Arasanipalai

***Abstract*—Phase ordering, the task of choosing and ordering code transformations from a set of semantics-preserving passes, is an important problem for optimizing compilers. The ordering of passes can have a significant impact on key performance metrics for the generated code, such as instruction count, run time, and build time.**

In this work, we propose a dynamic phase ordering policy based on a learned cost model for LLVM passes. Our learned cost model is able to predict code size reduction with a 9.5% error rate. We show that our cost model can be used to generate phase ordering policy that outperforms LLVM's built-in `-Oz` policy.

Code and pretrained models can be found at https://github.com/iyaja/O4.

I. Introduction

The LLVM [8] compiler framework has become a widely successful and popular tool for building optimizing compilers by abtracting away the front-end and back-end to enable powerful language-agnostic tools to be built over LLVM intermediate representation (IR).

Among the most useful tools in LLVM is `opt`, an optimization engine that transforms LLVM IR code by applying a set of transformations called passes, such as dead code elimination, constant propagation, and loop unrolling. These passes can, in general, be ordered arbitrarily, and the order of the passes can have a significant impact on the performance of the generated code. There are many phase ordering a classically challenging problem.

Complex Complier Internals. LLVM passes can be complex. Some involve many thousands of lines of C++ code designed by human experts with complex heuristics. Designing a good phase ordering policy involves not only understanding how each of these passes work, but also how specifically on the IR to be optimized.

Large Search Space. The search space of all possible optimization pipelines is very large (arguably infinite). For a pipeline with n phases and m passes, the search space is $O(n^m)$ - exponential in the number of phases. Furthermore, there is no hard limit to the total number of phases that can be used in a pipeline, but there is a point of diminishing returns. Applying more passes does not nessecarily improve performance, and can sometimes even hurt it. Furthermore, using more passes also increases build time, meaning a complex pipeline that improves performance but has too many passes may be unprofitable. A good policy must be able to balance the trade-off between performance metrics and build time.

Interdependancies Between Passes While phase ordering can be modeled as a sequential decision making process, passes in LLVM serve various purposes and can have complex interdependencies. For example, while some passes may not have an immediate or appreciable effect on the generated code, they may enable other passes to be applied or work more effectively.

Currently, LLVM implements a collection of "good default" optimization policies, such as `-O0` (no optimization), `-O1` (some level of runtime reduction with quick builds), `-Os` (codesize reduction), `-Oz` (aggresive codesize reduction), and `-O3` (aggresive runtime reduction). This is by far the most common approach for optimizing compilers.

Additionally, LLVM's built-in optimization pipelines are static - they apply the same set of passes to all IR. This is potentially suboptimal, as certain optimizations may be more or less effective on certain code patterns. However, designing a dynamic phase ordering policy is sufficiently complex that it is not feasible write manually. LLVM's default pipelines have been emperically shown to work well across a wide range of code patterns, and we argue that this reliance on experimental data naturally suggests that a learning-based search method might be more effective.

II. Method

In this work, we focus on the problem of choosing a dynamic phase ordering that minimizes codesize. Here, we present our proposed method for building and training O4 (collectively referring to both a cost model and the phase ordering policy). Our approach can be broken down into three steps:

1) Train a cost model to predict instruction count.
2) Train a classifier to predict the optimal pass to apply to current IR, using the cost model as a loss function.
3) Run the trained policy recursively on the desired IR, applying passes sequentially after each forward step.

The following sections decribe each of these steps in more detail.

A. Training a Cost Model

The first component of O4 is a cost model that predicts the effect of applying a given optimization from LLVM's predefined set of 124 passes exposed in Compiler Gym [4].

We use a transformer-based architecure for the cost model. More details are provided in III-A.

We use the LLVM IR instruction count as a target metric. While it is not a measure of true codesize reduction, as it does not take into account the effects of lowering, it is fast to evaluate, deterministic, and platform-independent. Additionally, IR instruction count is the default reward metric used in Compiler Gym [4], enabling us to draw a fair comparison with similar methods that have been evaluated on the same dataset and benchmarks.

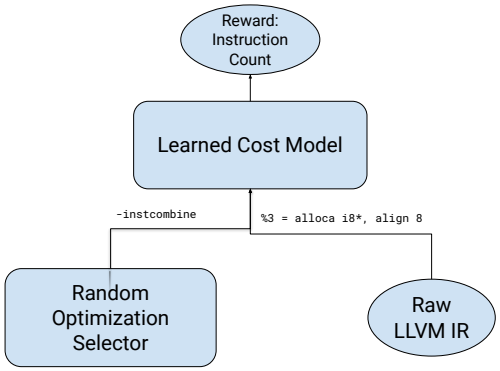

Fig. 1. **Training the cost model.**

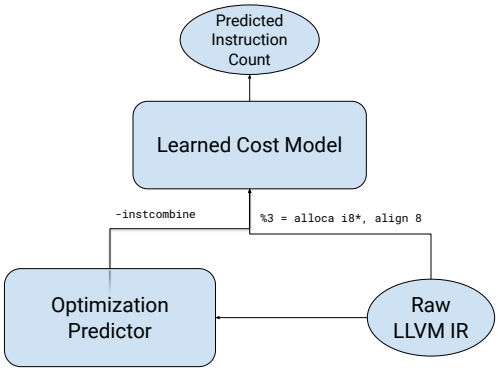

Fig. 2. **Training the policy network.**

The approach is summarized in II-A.

The cost model is trained using a supervised learning. Since the true reward (instruction count after applying an optimization pass to IR) is known at training time, we simply train the cost model to minimize the mean squared error between the true reward and the predicted reward.

The differentiable cost model learns predict the result of applying a given transformation to a given IR. The cost model takes the raw LLVM IR as input and predicts the desired reward metric, as shown in II-A.

The groud truth reward signal, which the cost model attempts to approximate, is the reduction in instruction count relative to LLVM's built-in `-Oz` pipeline:

$$R = \frac{IC_{o0} - IC_{o \sim \mathcal{U}(A)}}{IC_{o0} - IC_{oz}}$$

Where IC_p is the LLVM IR instruction count of the program after applyng optimization pipeline p (obtained using the `IrInstructionCount` observation space in Compiler Gym).

B. Training a Policy Network

Once the cost model has been trained, we can use it to train a classifier that predicts which optimization pass to apply to a given IR. The optimization prediction network takes the raw LLVM IR as input and predicts which one among the 124 passes should be applied. Then, the cost model is used as a supervision signal to train the network to improve it's predictions.

Since the cost model is a differentiable function, we can backpropagate through it and treat it as a loss function, with the learning objecttive of the policy network being to minimize the estimated instruction count from the cost model.

This second training step is summarized in II-B.

C. Generating a Phase Ordering

With both models trained, the final step is to generate a dynamic phase ordering. This is done by applying the policy network to the raw LLVM IR recurrently - apply the optimization predicted by the policy network, and then pass the transformed IR back into the network to predict the next optimization. This process repeated recurrently until the desired target performace is reached.

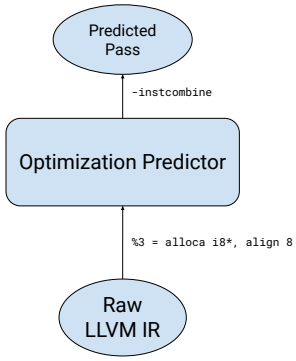

Fig. 3. **Generating a phase ordering.**

As shown in II-C, the only model required at compile time is a single copy of the optimization predictor. The cost model can be discarded once optimization predictor as been trained and does not need to be integrated into the compiler, unlike in other machine learning methods. This simple functional interface makes it easy to integrate O4 into existing compiler frameworks without rewriting pass managers to use the cost model directly.

III. Implementation

Next, we describe the implementation and training details of O4. Code is available in at https://github.com/iyaja/O4.

A. Inputs and Data

We train our models using a representative subset of datasets built into Compiler Gym [4]. Specifically, we use a combination of programs from the NASA parallel benchmark suite [1] and cBench [7]. Unfortunately, due to the use of this custom subset of small programs and limited compute resources, we

cannot make a fair comparison with current approaches on the Compiler Gym leaderboards.

The datasets are exposed as raw human-readable LLVM IR in Compiler Gym, and we use a custom tokenizer based on WordPiece [5] and Inst2vec [2] for our preprocessing pipeline. We train the WordPiece tokenizer on our dataset that is first normalized and preprocessed by Inst2vec [2]. We do **not** use the embeddings or tokens from Inst2vec, and instead build a custom fast tokenizer to work with the Huggingface Transformers library [10] and retrain the RoBERTa's embedding layer instead of using Inst2vec's fixed 200-dimensional vectors.

Where possible, the tokenizer learns to group entire instructions into a single token. This enables us to pack many more instructions into an input sequence than a conventionally trained Byte-Pair Encoding (BPE) tokenizer, which is especially important for models like RoBERTa, which have an input sequence length limit of 512 tokens. Since the model is trained to predict instruction count, not including all instructions can be problematic, since the model would have no sense of how long the full IR is. Future work could investiage augmenting the raw IR with additional metadata to circumvent this limitation.

For the cost model, the current pass to be applied on the is prepended as a special token to the input sequence, For the policy network, the tokenized instructions are passed directly to the transformer.

B. Neural Networks

We use RoBERTa [9], a popular variant of the BERT [5] transformer model as our base architecture for both the cost model and the policy network. We initialize it from the CodeBERT [6] checkpoint and leave architecure exploration for future work, acknowleding that there is potentially scope for more complex model that use , such as Programl embeddings [3] with graph neural networks to extract more semantic information from the IR.

CodeBERT [6] is a large language model pretrained on a large dataset of code snippets in high level languages, such as Java and Go. While this is not representative of our target domain (LLVM IR), we reasoned that it would be better than initializing our cost model from random weights and pretraining a new language model from scratch. While pretraining a language model on raw LLVM IR is bound to be a more effective, this approach was prohibitively expensive, and we leave it for future work.

Both the cost model and the policy network are implemented using the Huggingface Transformers library [10]. Pretrained models and are available on the Huggingface model hub [1].

IV. Results

Our current results are summarized in I. Of main interest is the error rate of the cost model - the best model, trained on 1000 samples of cBench [7], is able to predict the relative instruction count reduction with an error rate of 11.4% (root mean-squared error).

TABLE I
Summary of results.

Dataset	Unique Programs	Samples	Passes	MSE
cbench-v0	23	100	5	0.031
cbench-v0	23	500	10	0.019
cbench-v0	23	1000	20	**0.013**
npb-v0	122	64	32	-

Unfortunately, the policy network network, as trained on the best cost model, is unable to reach the performance of random search. We hope to iterate on this model an improve it's performance in future work.

V. Conclusion

Phase ordering is an import yet complex classic compiler problem that can have a significant impact on key performance metrics for the generated code. Our proposed method, O4, uses a learned cost model to generate a dynamic phase ordering policy. While our generated phase orderings do not outperform LLVM's built-in `-Oz` policy on average, they demonstrate that a learned policy for phase ordering, with more training data and time, can be effective.

Furthermore, our approach learns and adapts and scales with more data, which opens the door to phase ordering policies finetuned for specific domains, architecures, and applications.

References

[1] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[2] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[3] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *arXiv preprint arXiv:2003.10536*, 2020.

[4] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," *arXiv:2109.08267*, 2021.

[5] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[7] G. Fursin, "Collective tuning initiative: automating and accelerating development and optimization of computing systems," in *GCC Developers' Summit*, 2009.

[8] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[9] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[10] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[1] Pretrained models and tokenizers are availabe at https://huggingface.co/iyaja/O4