Brandon Howell
CSC 587

Static Neural Compiler Optimization via Deep Reinforcement Learning - Cons

While the approach shows promise, outperforming LLVM's O3 sequence on the training set and achieving competitive performance on the validation set with up to 1.32x speedup on unseen programs, the sources highlight **several significant limitations** and aspects that were less convincing in the evaluation, including their dataset size, the chosen encoding method, hardware specificity, scalability challenges, and their overall results.

The limited dataset size and diversity used for training (109 single-source benchmarks) is a big major weakness. The authors themselves attribute the agent's inferior performance on the validation set compared to O3, and a lack of generalization, partly to this small and non-diverse dataset. Furthermore, the intermediate representation (IR) **encoding method** used (embeddings by Ben-Nun et al.) has limitations, resulting in a loss of information, such as immediate values of instructions. The authors suggest this encoding limitation also contributes to the performance gap on the validation set. Additionally, the learned agents exhibit **hardware specificity**, being fine-tuned for the particular hardware configuration used for training and benchmarking. This means an agent trained on one system might not perform optimally on another. MLCO [1] explicitly highlights the challenge of hardware dependency, reinforcing the acknowledged limitation and suggesting potential solutions like training agents per hardware unit or including hardware information in the state representation. The approach also faces **scalability challenges with larger action spaces**. The number of possible optimization sequences grows exponentially, and although strategies were employed to manage this, experiments in the largest action space (L) resulted in the loss function diverging, making further training difficult and requiring prohibitively increased time for stability. Relatedly, experiments in the middle action space (M) did not show performance improvement on the validation set and were terminated due to limited resources. These issues point to challenges in effectively training the RL agent as the complexity of the search space increases. Finally, the paper uses step-by-step action selection, which may not be effective if optimizations' effectiveness depends on which optimizations are already applied. Instead of just learning a step-by-step action selection as this paper does, TCT [2] proposes framing the problem as predicting the effectiveness of individual passes to skip them (compile-time reduction) or clustering programs to apply tailored pass sequences.

Beyond the fundamental limitations, the experimental results showed several less convincing aspects. Most notably, while the agent outperforms O3 on the training set (2.24x vs 2.17x speedup), its performance on the validation set is simply worse (2.38x vs 2.67x speedup). This significant performance gap on unseen data strongly indicates a lack of generalization beyond the training programs. The agent also lacks robustness, very rarely significantly outperforming O3 and failing to be consistently effective across all source codes. Overall, the paper has a very interesting premise and does show promising potential, but in the way it's currently implemented, it is nowhere near effective enough to pursue implementation.

1. Machine Learning in Compiler Optimization
2. Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning