RL-Phase: Compiler Pass Ordering for Mobile Systems using Reinforcement Learning

Charles Cal, Anthony Li, Andrew Scheffer, Nikhil Patel, and Evan Zimmerman

***Abstract*— The order of optimization passes in modern compilers can have a significant influence on the efficiency of generated code, impacting metrics such as runtime, code size, and energy efficiency. Conventional systems use fixed, heuristically determined static optimization passes created by domain experts, which may overlook potential opportunities to improve code efficiency. This work introduces a novel reinforcement learning (RL) based approach to address the pass-ordering problem for mobile and embedded systems. Our method jointly optimizes code size and statically estimated energy efficiency, achieving noticeable improvements over baseline LLVM optimization levels like `-O3` and `-Oz` on various benchmarks. Our open-source implementation and additional documentation can be found on our GitHub: https://github.com/schefferac2020/EnergyAwareLLVM.**

I. Introduction

One of the critical tasks of compilers, aside from directly translating source code to machine code, is to improve code efficiency. Various optimizations involve transforming some input program to an output program that is provably semantically equivalent but may improve metrics like runtime, code size, or energy efficiency. Though we can imagine that there is some correlation between these metrics, optimization passes may sacrifice the performance in some metrics in order to produce better results in others [1]. These trade-offs become more pronounced when compiling for systems with limited memory or battery power such as mobile or embedded devices, where energy efficiency becomes a more relevant concern than typical optimization scenarios.

Modern compilers, such as LLVM, can implement upwards of 80 independent optimization passes (i.e. loop-unrolling, vectorization, dead-code elimination, etc). In most common use cases, the order of these passes are fixed and packaged into standard optimization flags such as `-O3` and `-Oz`, containing up to 300 optimization passes with repetition. These standard flags order optimization passes for different objectives. For example, the `-O3` optimization level prioritizes aggressively speeding up the runtime of a program, even if its passes potentially increase the code size in memory. Conversely, `-Oz` optimizes specifically to reduce code size, potentially increasing the runtime of the program.

The long-standing problem of finding an optimal ordering of optimization passes for a particular program is a difficult one. The optimization search space is incredibly large given the number of potential optimization passes. Even so, the phase ordering problem is still recognized as an important problem to study for numerous reasons: 1) A pass ordering does not guarantee improvement for all programs and 2) Different permutations of optimization passes can yield different results. For these reasons, we focus on the problem of identifying unique pass orderings for specific programs.

In this work, we investigate the potential of using reinforcement learning techniques to create an agent capable of iteratively applying optimization passes to arbitrary LLVM IR programs. We narrow our focus to the use case of mobile and embedded devices where the need for minimal code size and energy usage is exacerbated. We train a model that simultaneously optimizes for both code size and statically estimated energy efficiency. We train and evaluate on various benchmarks from the *cbench* dataset [2] with various ablations. Overall, we find that our framework is capable of outperforming statically determined pass sequences on numerous benchmarks while simultaneously optimizing differing objectives.

II. Related Work

A. Reinforcement Learning for Phase Ordering

The authors of POSET-RL observe that the traditional fixed optimization sequences may not yield optimal results across all programs [1]. Their solution employs an RL-based approach that uses Deep Q-Networks (DQN) to tailor optimization pass sequences for individual programs for runtime efficiency. It uses IR2Vec [3] to create vector embeddings from LLVM IR, and uses an action space primarily built on the dependence graph of the optimization passes used in the `-Oz` optimization flag.

"Autophase" examines optimizing the order of compiler optimization passes in High Level Synthesis (HLS) by using a deep reinforcement learning approach [4]. They extracted static features from LLVM IR that they could use to define the observation space for their RL agent. They evaluated the Policy Gradient and DQN RL algorithms compared to various baselines. Our work uses the observation space from Autophase for our RL agent.

A later paper by the same authors focuses on Proximal Policy Optimization and Asynchronous Advantage Actor-Critic RL methods [5]. This work explores the use of random forests to understand the relationship between program features and the effectiveness of optimization passes. Their use of random forests helps decrease the search space.

CompilerDream uses a compiler world model to simulate the effects of optimization passes efficiently [6]. CompilerDream trains a model to simulate the state transitions and rewards of various optimization passes, which eliminates the need to compile between a RL agent's actions. CompilerDream's solution underscores the need for efficient training environments for RL-based compiler optimizations.

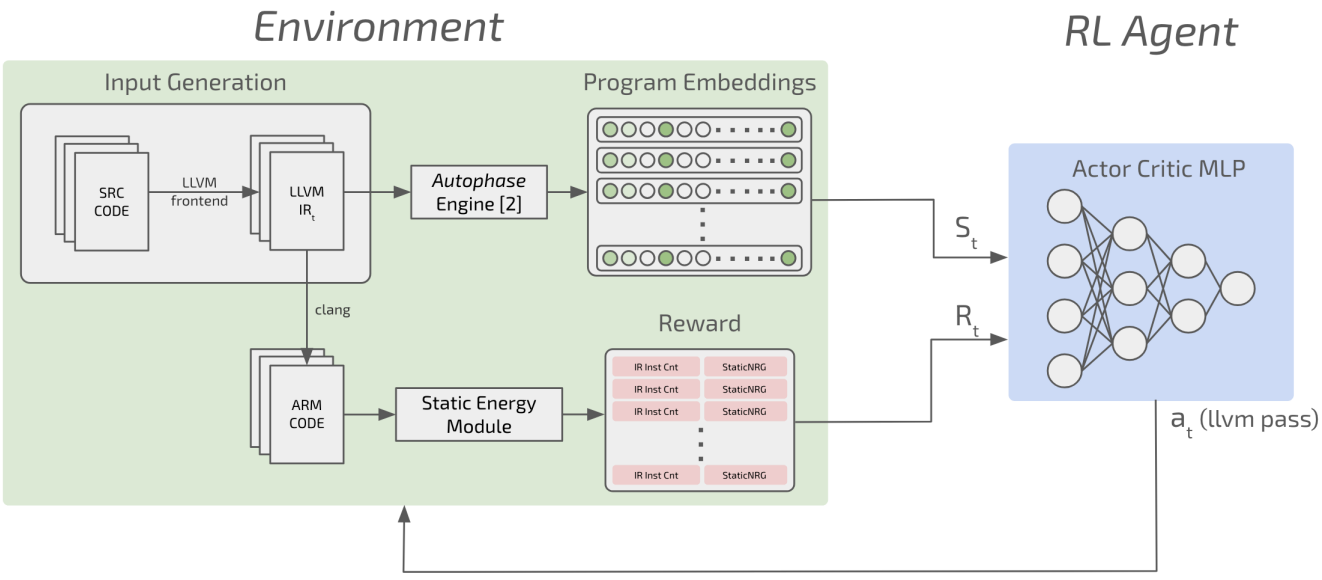

Fig. 1. An overview of the *RL-Phase* model architecture designed to jointly optimize LLVM IR programs for energy efficiency and code size by iteratively applying LLVM passes to a program.

While our approach used a simpler model, we avoided re-compilation by relying on a dynamic energy estimation model rooted in static analysis.

B. Energy Estimation and Optimization

Previous work has looked at optimizing IR code for energy consumption, although not through reinforcement learning. For example, Nobre, Reis, and Cardoso [7] use statistical techniques and a graph-based model to construct compiler sequences, then analyze them for energy efficiency.

Another work explored energy optimization via a fractional factorial design to explore the large action space [8]. They evaluated their energy results using hardware. Some of their key findings included noticing a correlation between energy and runtime efficiencies, and noting that sets of optimizations can outperform standard GCC optimization levels.

Grech et. al examine techniques to estimate energy consumption for embedded systems through static analysis of IR [9]. They map program segments at the LLVM IR level to program segments at the ISA level, and use this mapping to for an ISA-level energy model. They also directly estimate energy consumption from LLVM IR. Our work is inspired by these approaches for statically estimating dynamic energy consumption.

In Georgiou et. al's paper, *"Less is More"*, the authors investigated how reducing the number of optimizations in traditional optimization levels, while maintaining their order, can actually improve program efficiency in terms of runtime and energy efficiency for ARM processors [10]. In this work, the authors use physical hardware to measure energy consumption.

III. Methodology

In the case where a compiler has m unique optimization passes at its disposal, and can apply a sequence of N passes in total, there are a total of m^N different optimization pass orderings that a compiler could apply to optimize a given input program. Creating a dataset to learn the ideal pass ordering for an input program is infeasible as the number of unique optimization pass orderings is too large.

Reinforcement learning is a technique well suited for navigating the massive search space that is optimization pass orderings because it can train an agent to learn the best action (optimization pass) for a specified objective (reducing energy consumption and code size), given the current state of an input program. A reinforcement learning agent learns through strategic action selection and targeted feedback from its environment, effectively balancing exploring actions it is unsure of and exploiting actions it learns to be effective.

A. Reinforcement Learning

The formulation of our Reinforcement Learning System is shown in Fig. 1. There are two main components of this system: the environment and the agent. The environment is implemented through a wrapper that extends CompilerGym, a framework for training reinforcement learning agents for compiler optimization tasks [11]. The environment largely differs from the original LLVM CompilerGym environment in regard to the stepping process – it modifies the stepping process so that the agent receives statically estimated dynamic energy consumption as a reward for applying an optimization pass.

B. Environment

1) Input: The environment is generated from a source code file. Upon initialization, the source code is compiled

to LLVM IR, and is maintained as LLVM IR throughout the optimization passes applied by the agent.

2) Observation Space: The environment uses the Autophase [4] observation space from CompilerGym to provide the agent with a feature vector to represent the current state of the program. This feature vector summarizes the static LLVM-IR representation, containing features such as *number of basic blocks with 2 successors* and *number of branches*

3) Action Space: The action space, inherited from CompilerGym, is discrete, and contains 123 unique optimization passes. This action space includes the optimization passes found in common optimization levels such as `-O2`, `-O3`, and `-Oz`.

4) Rewards: In order for the agent to learn if an optimization pass was effective in reducing the energy consumption of a program, the reward function reflects a change in energy consumption from one state of LLVM IR to the following state after applying an optimization pass (1). In some cases, developers may not want to solely optimize their code for energy consumption. Instead, they may want to optimize for a dual objective – for example, both energy consumption and code size. To consider this situation, a separate reinforcement learning agent was trained with a reward function that was a weighted combination of both change in code size and change in energy consumption (3). The *IrInstructionCount-Norm* reward space from CompilerGym was used to implement the change in code size reward. As a baseline, another model was trained solely with change in code size as the reward (2). All terms in reward functions were normalized by the value of the respective metric described by the term in the initial state of the program.

$$R_{NRG}(s_t) = \frac{E(s_{t-1}) - E(s_t)}{E(s_{t=0})} \tag{1}$$

$$R_{InstCount}(s_t) = \frac{Size(s_{t-1}) - Size(s_t)}{Size(s_{t=0})} \tag{2}$$

$$R_{BOTH}(s_t) = 0.5 * R_{InstCount}(s_t) + 0.5 * R_{NRG}(s_t) \tag{3}$$

C. Statically Estimated Dynamic Energy Consumption

In order to create the energy reward function, we require a method to estimate the energy cost of running a given program that is both accurate enough to guide optimizations and efficient enough to be applied after every transformation. As such, we ruled out any approach requiring us to run the program to measure actual energy usage. Instead, we statically estimate dynamic energy usage by inspecting the LLVM IR representation of a program. We lack the necessary hardware to develop a mapping of LLVM IR instructions to energy costs ourselves. Thus, we build upon previous work by Vasilakis [12], which provides a model linking ARM instructions to estimated energy costs and includes a methodology for accounting for cache-level penalties and data hazards.

Each instruction type – such as integer arithmetic, floating-point operation, memory load/store, etc. – is associated with a baseline energy cost and cycle count stored in a lookup table. After compiling the LLVM IR to ARM Cortex-A7 Assembly, we parse the resulting code line-by-line, classifying each instruction and identifying whether it uses immediate or register operands. Using this classification, we then consult the lookup table to retrieve the appropriate energy and cycle values, incrementally accumulating an overall energy cost estimate as we process the code.

Energy consumption is notably influenced by memory access patterns, as fetching from lower-level caches or main memory increases energy use significantly. To approximate this effect, we perform a coarse-grained estimation of the program's memory footprint by counting unique memory references. We map this footprint to discrete categories, each associated with a different baseline energy cost for load/store operations. Although this method does not capture dynamic cache behavior or runtime data layouts, it serves as a practical heuristic.

We also model the effect that Read-After-Write (RAW) hazards may have on pipeline efficiency. Since we record the cycle count of all instructions, we can estimate the energy impact of potential stalls. To avoid underestimating overall usage, we conservatively assign the higher RAW-dependent cost to most instructions, acknowledging that a more precise, data-flow-based analysis could yield finer-grained estimates.

Our static estimator of dynamic energy simplifies many aspects of real-world execution. It does not account for loop iterations, cache behavior, complex pipeline behavior, branch prediction, prefetching heuristics, and many other intricacies that influence actual energy consumption. It is important to stress that this estimation method serves as a baseline – one that enables the RL agent to begin learning meaningful energy-related behaviors. Our RL pipeline is intentionally modular, allowing future researchers to integrate more accurate models tuned to their specific hardware. Overall, we have formed a flexible, foundational estimation strategy that, despite its simplifications, enables our RL system to iteratively refine compiler optimizations for improved energy efficiency.

D. Agent

The agent is trained through the Proximal Policy Optimization (PPO) algorithm, which falls under the Actor-Critic framework. This algorithm employs a multi-layer perceptron to capture the policy network, which learns to select optimal actions, and the value network, which estimates the advantage function to guide policy updates. The key advantage offered by the PPO algorithm over other reinforcement learning algorithms is a clipped objective function, which limits how much a new policy update can deviate from the old policy.

The PPO training process involves iteratively improving a policy by interacting with the environment and optimizing a clipped surrogate objective. First, the agent collects trajectories by following the current policy and records states, actions, rewards, and log probabilities of actions. Using these experiences, it computes rewards-to-go and advantage

estimates, typically with Generalized Advantage Estimation (GAE). The policy and value networks are then updated by minimizing a total loss that combines the clipped policy loss, value loss, and an entropy bonus for exploration. Updates are performed over multiple mini-batches of steps to stabilize learning. This process repeats during training, with the policy improving iteratively until convergence.

IV. Results and Discussion

In this section, we evaluate the effectiveness of our trained reinforcement learning agents at reducing code size and improving energy efficiency over a range of unseen test programs from *cbench* [2]. We also investigate what the model actually learned throughout training.

A. Optimizing to Reduce Code Size

We first evaluate the effectiveness of our trained reinforcement learning models on reducing code size over six *cbench* benchmarks. As seen in Fig. 2, we compare our trained networks to three baselines: **1)** the `-O3` LLVM compiler pass, **2)** the `-Oz` LLVM compiler pass, and **3)** a random baseline which applies a randomized optimization sequence of equal length as our model's output. For each benchmark, each of the six optimization sequences was run 10 times and the maximum code reduction value was plotted.

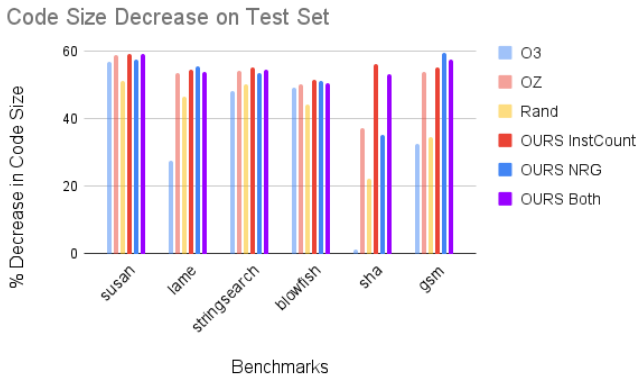

Fig. 2. Evaluation of code size decrease among *RL-Phase*$_{\text{InstCount}}$ (red), *RL-Phase*$_{\text{NRG}}$ (blue), and *RL-Phase*$_{\text{Both}}$ (purple) models applied to six unseen test programs.

One important takeaway from Fig. 2 is that *RL-Phase*$_{\text{InstCount}}$, which was trained specifically to reduce code size, reliably outperforms the `-Oz`, `-O3`, and randomized benchmarks at reducing code size. Another important takeaway is the stark difference in code size reduction achieved by `-O3` and `-Oz` optimizations. There seems to be a noticeable tradeoff between optimizing for code size and optimizing for runtime performance. Similarly, the performance of the *RL-Phase*$_{\text{NRG}}$ model can be found to dramatically underperform the `-Oz` baseline in the *sha* benchmark. This indicates that the optimization objective of statically estimated energy cost and code size may not be perfectly correlated. This intuition echos the ideas presented by Pallister, Hollis, and Bennett [8].

Another quite promising result from Fig. 2 can be seen with the evaluation of the *RL-Phase*$_{\text{Both}}$ model, trained on a weighted combination of the code size and energy reward. Interestingly, when compared to the *RL-Phase*$_{\text{NRG}}$ model, the underwhelming performance on the *sha* benchmark seems to be largely recovered. This gives confidence that our RL framework seems to be learning reasonable ways of exploiting both energy-based and code-size-based rewards through optimizing pass orderings.

B. Optimizing to Reduce Estimated Energy Cost

Next, in a very similar manner, we evaluate the effectiveness of our trained reinforcement learning models at reducing the statically estimated energy cost of a set of programs shown in Fig. 3. Due to time and resource constraints, we were unable to evaluate our model using real hardware energy measurements nor did we receive reliable measurements of energy decrease for the `-O3` and `-Oz` baselines. We consider these extensions as an avenue for future work.

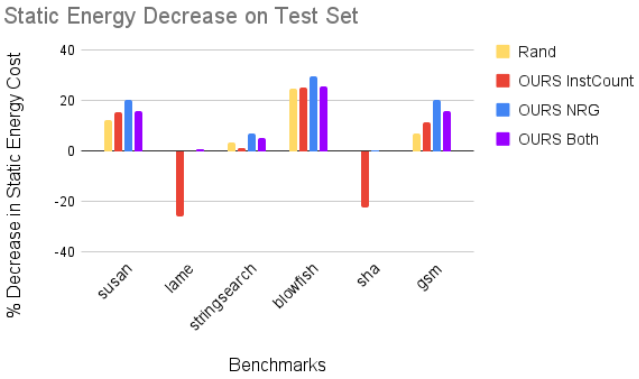

Fig. 3. Evaluation of the decrease in statically estimated energy consumption among *RL-Phase*$_{\text{InstCount}}$ (red), *RL-Phase*$_{\text{NRG}}$ (blue), and *RL-Phase*$_{\text{Both}}$ (purple) models applied to six unseen test programs.

From Fig. 3, a similar conclusion can be drawn about the largely differing objectives of the *RL-Phase*$_{\text{InstCount}}$ and *RL-Phase*$_{\text{NRG}}$ models. As seen, an agent trained solely on optimizing for code size actually increases the estimation of dynamic energy cost in the *lame* and *sha* benchmarks. As expected, the model designed to only minimize energy consumption outperforms all other methods. More interestingly, *RL-Phase*$_{\text{Both}}$ model seems to have learned to avoid the large negative rewards while consistently outperforming both the random baseline and the *RL-Phase*$_{\text{InstCount}}$ model. This further indicates that our dual-objective learning strategy seems to be working reasonably well across most benchmark programs, even if the optimization objectives of code size reduction and energy efficiency are not perfectly aligned.

C. Evaluating the Reward Space

In an effort to further examine the learned behaviors of our RL agents, we examined the specific actions that yielded the highest and lowest rewards across a specific benchmark. We chose the *susan* benchmark from the *cbench* suite and evaluated the reward returned from each action while rolling out

the policies for both the *RL-Phase*$_{\text{InstCount}}$ and *RL-Phase*$_{\text{NRG}}$ models. The corresponding reward distributions are plotted in Fig. 4 and Fig. 5 respectively. Interestingly, there seems to be a finite number of actions that drastically affect the overall performance of either model. The actions with the highest and lowest rewards for each model are noted in Table I. For both models, the five actions with the highest rewards are largely similar. Ubiquitous actions such as `-sroa` and `-simplifycfg` seem to have an important influence on the reward for both models, even though they optimize for different metrics.

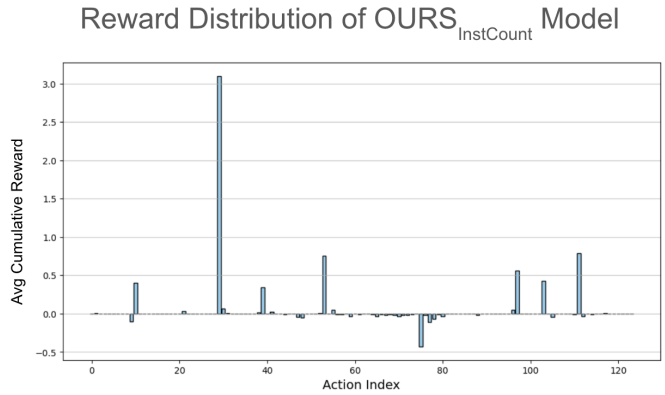

Fig. 4. Cumulative reward per LLVM pass action for *RL-Phase*$_{\text{InstCount}}$ model normalized over 10 episodes of the *cbench/Susan* benchmark

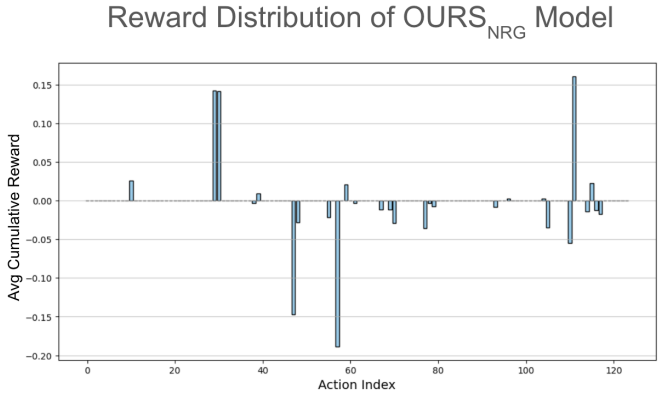

Fig. 5. Cumulative reward per LLVM pass action for *RL-Phase*$_{\text{InstCount}}$ model normalized over 10 episodes of the *cbench/Susan* benchmark

TABLE I
LLVM PASSES THAT YIELDED THE HIGHEST AND LOWEST REWARDS FOR *RL-Phase*$_{\text{INSTCOUNT}}$ AND *RL-Phase*$_{\text{NRG}}$ MODELS

Model	Highest Rewards	Lowest Rewards
RL-Phase$_{InstCount}$	**-early-cse-memssa -sroa** -instcombine -newgvn -mem2reg **-simplifycfg**	-loop-reduce -loop-unroll -break-crit-edges -loop-unswitch -irce
RL-Phase$_{NRG}$	**-sroa -early-cse-memssa** -early-cse **-simplifycfg** -sink -load-store-vectorizer	-licm -indvars -slp-vectorizer -loop-reroll -reassociate

Another important insight gained from this analysis is that in Table I, one can notice that the LLVM pass that consistently yields the lowest reward is in fact Loop Invariant Code Motion (`-licm`). This result is slightly counterintuitive to what we would expect our energy model to disincentivize as LICM typically reduces the number of memory accesses and thus likely reduces the dynamic energy cost of running the program. This exposes some potential shortcomings of our current static estimate of the energy cost of a program. Since our current energy estimate is a proof-of-concept, this leaves the door open for future work to implement their own more accurate energy models and hot-swap them into our custom dual-objective reinforcement learning system.

V. CONCLUSION

In this work, we developed *RL-Phase*, an RL-based approach to the phase-ordering problem, focused on optimizing for code size and energy efficiency. We trained three models, with rewards corresponding to only code size, only energy, or a weighted average of both. Evaluating on six randomly chosen benchmarks in the cbench test suite, our results showed that our models outperform `-O3`, `-Oz`, and a random-sequence policy, in the respective metrics that the models were trained on. These results show promise towards our goal of finding better pass orderings for program optimization in mobile and embedded domains.

A. Future Work

Concerning *RL-Phase*, further work can be done to profile dynamic energy consumption on real hardware systems in order to analyze the effectiveness of *RL-Phase* in practice. Additionally, it may be important to analyze runtime metrics in addition to code size and energy efficiency, in order to better understand the tradeoffs made between the three metrics in the *RL-Phase* agents. Since our training involves a modular component to statically estimate dynamic energy, it may be interesting to retrain with different models to see how this would impact optimization pass choices.

We imagine that future work in the realm of energy-aware optimization for mobile systems may explore other statically calculated estimations for dynamic energy in different architectures. There is also potential in developing other RL and ML algorithms, and examining how the pass sequences they produce compare to human-made heuristics on larger datasets.

B. Additional Project Deliverables

For further results, view our project presentation and GitHub which are linked below.
Project Presentation: https://docs.google.com/presentation/.
Project Github: github.com/schefferac2020/NRGLLVM.

References

[1] Shalini Jain et al. "POSET-RL: Phase ordering for Optimizing Size and Execution Time using Reinforcement Learning". In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, May 2022, pp. 121–131. DOI: 10.1109/ispass55109.2022.00012. URL: http://dx.doi.org/10.1109/ISPASS55109.2022.00012.

[2] Fursin, Grigori. "Collective Tuning Initiative: automating and accelerating development and optimization of computing systems". In: (June 2009). URL: https://arxiv.org/pdf/1407.3487.

[3] S. VenkataKeerthy et al. "IR2VEC: LLVM IR Based Scalable Program Embeddings". In: *ACM Transactions on Architecture and Code Optimization* 17.4 (Dec. 2020), pp. 1–27. ISSN: 1544-3973. DOI: 10.1145/3418463. URL: http://dx.doi.org/10.1145/3418463.

[4] Qijing Huang et al. "AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning". In: *ArXiv* abs/2003.00671 (2020). URL: https://arxiv.org/pdf/2003.00671.

[5] Qijing Huang et al. *AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning*. 2020. arXiv: 2003.00671 [cs.DC]. URL: https://arxiv.org/abs/2003.00671.

[6] Chaoyi Deng et al. *CompilerDream: Learning a Compiler World Model for General Code Optimization*. 2024. DOI: 10.48550/ARXIV.2404.16077. URL: https://arxiv.org/abs/2404.16077.

[7] Ricardo Nobre, Luís Reis, and João M. P. Cardoso. *Compiler Phase Ordering as an Orthogonal Approach for Reducing Energy Consumption*. 2018. arXiv: 1807.00638 [cs.PF]. URL: https://arxiv.org/abs/1807.00638.

[8] James Pallister, Simon J. Hollis, and Jeremy Bennett. "Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms". In: *The Computer Journal* 58.1 (2015), pp. 95–109. DOI: 10.1093/comjnl/bxt129.

[9] Neville Grech et al. "Static analysis of energy consumption for LLVM IR programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '15. ACM, June 2015, pp. 12–21. DOI: 10.1145/2764967.2764974. URL: http://dx.doi.org/10.1145/2764967.2764974.

[10] Kyriakos Georgiou et al. "Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption". In: *CoRR* abs/1802.09845 (2018). arXiv: 1802.09845. URL: http://arxiv.org/abs/1802.09845.

[11] Chris Cummins et al. "CompilerGym: robust, performant compiler optimization environments for AI research". In: *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '22. Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 92–105. ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741258. URL: https://doi.org/10.1109/CGO53902.2022.9741258.

[12] Evangelos Vasilakis. *An Instruction Level Energy Characterization of ARM Processors*. Tech. rep. FORTH-CS/TR-450. Heraklion, Greece: Foundation of Research and Technology Hellas - Institute of Computer Science, 2015. URL: https://projects.ics.forth.gr/carv/greenvm/files/tr450.pdf.