Machine Learning in Compiler Optimization

Ameer Haj Ali

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2021-2
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-2.html

February 17, 2021

Machine Learning in Compiler Optimization

by

Ameer Haj-Ali

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Fall 2020

Machine Learning in Compiler Optimization

Abstract

Machine Learning in Compiler Optimization

by

Ameer Haj-Ali

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Krste Asanović, Co-chair

Professor Ion Stoica, Co-chair

The end of Moore's law is driving the search for new techniques to improve system performance as applications continue to evolve rapidly and computing power demands continue to rise. One promising technique is to build more intelligent compilers. Compilers map high-level programs to lower-level primitives that run on hardware. During this process, compilers perform many complex optimizations to boost the performance of the generated code. These optimizations often require solving NP-Hard problems and dealing with an enormous search space. To overcome these challenges, compilers currently use hand-engineered heuristics that can achieve good but often far-from-optimal performance. Alternatively, software engineers resort to manually writing the optimizations for every section in the code, a burdensome process that requires prior experience and significantly increases the development time.

In this thesis, novel approaches for automatically handling complex compiler optimization tasks are explored. End-to-end solutions using deep reinforcement learning and other machine learning algorithms are proposed. These solutions dramatically reduce the search time while capturing the code structure, different instructions, dependencies, and data structures to enable learning a sophisticated model that can better predict the actual performance cost and determine superior compiler optimizations. The proposed techniques can outperform existing state-of-the-art solutions while requiring shorter search time. Furthermore, unlike existing solutions, the deep reinforcement learning solutions are shown to generalize well to real benchmarks.

To my parents, *Aida* and *Yosef*,
my sisters, *Siwar*, *Reem*, *Faten*, and *Rouba*, and
my nieces and nephews, *Bana*, *Khalid*, *Rani*, *Anan*, *Aamer*, and *Nai*,
for their constant love and support!

Contents

List of Figures

List of Tables

Acknowledgments

Working towards a PhD has been a deeply enriching experience. The things I learned and the skills I gained are exceptional. While sometimes it was depressing, it was always rewarding. Looking back, this journey would have not been successful without many people to whom I would like to extend my thanks.

I am very grateful to my thesis co-advisor Ion Stoica for his great guidance throughout my graduate studies. From the very beginning Ion had an open door for me, was always available for me, and treated me like a friend. He always maintained my focus, kept me motivated, and reminded me to start my research by identifying the right problems that matter and by focusing on real-world impact. He made me realize that sometimes it is possible learn without making mistakes. Ion always gave me direct and honest feedback, which made me a better researcher and helped me become a better person. I am also very thankful to my thesis co-advisor Krste Asanovic. Krste always reminded me to follow my passion and my dreams. His feedback was very instrumental and helped identify the right problems to tackle. For all of this and much more, Ion and Krste, thank you, my work would not have been possible without you. I hope and look forward to continued collaboration with you in the future.

I would also like to thank Qijing (Jenny) Huang who influenced my PhD more than anyone else and collaborated with me on multiple projects from the beginning of my PhD. She was always ready to help me, as were Hasan Genc and William Moses. I have benefited greatly from working with them. My collaboration with them has been one of the most fruitful and fun engagements I have experienced. I learned a lot writing multiple papers with them.

I have been very fortunate to work closely with some wonderful colleagues and peers in the BWRC, ADEPT and RISE labs at UC Berkeley including Alon Amid, Lianmin Zheng, Keertana Settaluri, Arya Reais-Parsi, Sagar Karandikar, David Kohlbrenner, Eyal Sela, Chloe Liu, Seah Kim, David Biancolin, Abraham Gonzalez, Adam Izraelevitz, Dayeol Lee, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Colin Schmidt, John Wright, Zongheng Yang, Eric Liang, Richard Liaw, Giulia Guidi, and Paras Jain. I am also thankful to the administrators, and lab engineers in these labs for their high availability and readiness to help.

I am thankful to Professor Jonathan Ragan-Kelley for being on my prelim committee and guiding me in my research, Jacob Steinhardt for being on my qualification exam and thesis committee, Yakun Sophia Shao for guiding me and being on my qualification exam committee, and Borivoje Nikolic and Joseph Gonzalez for guiding me. I am also grateful to Shirley Salanio for her availability and having her door always open for me, and the international office for assisting me with my nonimmigrant student status in the US.

My internship at Intel Labs would not have been so successful without the help of Ted Willke, Nesreen K. Ahmed, Bin Li, and Nicole Beckage, as well as a large group of engineers who assisted me and worked with me during my internship.

I am grateful to Professor Shahar Kvatinsky for recommending me to UC Berkeley and Professor Elad Alon for accepting me. I am also grateful to the International House for granting me the gateway fellowship, which allowed me to live and meet with a great and diverse group of students at UC Berkeley, and more importantly, provided me a welcoming home outside my country.

Last, but not least, I would like to thank my family: my parents, Aida and Yosef, my sisters, Rouba, Faten, Siwar, and Reem, and my nieces and nephews, Bana, Khalid, Rani, Anan, Aamer, and Nai. I am very fortunate and blessed to have you in my life. Despite all, you always loved me, motivated me, have been there for me, believed in me, and gave me all I needed. Words cannot express how grateful I am for your selfless love and the joy you bring to my life. This is dedicated to you.

Chapter 1

Introduction

In the era of big data, artificial intelligence, cloud computing, and the Internet of Things, our thirst for computing power has never been greater. At the same time, the end of Moore's law is plainly in sight. This calls for alternative, novel techniques to continue scaling performance. One efficient, cost-effective, and quick-to-deploy technique is to leverage existing hardware by improving the mapping of program code to hardware primitives. This mapping is generally done by a compiler. The compiler translates the high level code to low level instructions that run on the hardware. This translation has to maintain the correctness of the code while still achieving good performance, a very challenging task that often requires solving complex NP-hard optimization problems with enormous search space. Compilers rely on hand-engineered heuristics and greedy algorithms, which can achieve good performance but often fall short in finding optimal solutions. Despite decades of research on developing sophisticated optimization algorithms, there is still a performance gap between the code generated by a compiler and the hand-optimized code produced by experts [1–18]. This often leads software engineers to manually optimize the code by rewriting it, writing the low level instructions, and giving optimization hints to the compiler, which is an error-prone task that requires prior experience and significantly increases the development time.

Good solutions to compiler optimization problems are hindered by the large space of options available and the need to compile the entire program in reasonable time. The characteristics of the underlying hardware must also be considered—whether the optimization decisions are made by the compiler or the software engineer. With the diminishing returns of Moore's law, the recent decade has seen a plethora of new, customized hardware that increased the challenge of writing optimized code to different hardware targets. Overcoming these challenges requires a new technique that can efficiently search the space of optimizations efficiently, and similarly to a compiler expert, examine the computation graph, functionality, data dependencies, and code characteristics to find the best optimizations.

Many compiler optimization problems can be formulated as a Markov Decision Process (MDP) [19], where the next optimization to apply is the action, the state is the current program's intermediate representation, and the reward is proportional to the decrease in execution time. When an action is applied, the generated program changes, as does its performance. The solution would be the actions that lead to the optimal program state; hence, solving the MDP can guarantee the optimal performance. Recent advancements in machine learning, specifically, reinforcement learning (RL) [20, 21], show promising results in efficiently solving MDPs. Among the most successful techniques is to use a neural network in conjunction with RL, also known as deep RL. Deep models allow RL algorithms to solve complex problems in an end-to-end fashion, handle unstructured environments, learn complex functions, or predict actions in states that have not been visited in the past. Deep RL is gaining wide interest due to its success in robotics and Atari games, and for its superhuman performance capabilities [22–25]. Deep RL was the key technique behind defeating the human European champion in the game of Go, long viewed as the most challenging of classic games for artificial intelligence [26]. In the deep RL setting, a software agent continuously interacts with the environment by taking actions. Each action can change the state of the environment and generate a reward. The goal of the RL agent is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward.

Deep RL is also gaining wide interest in the field of system optimization [27]. Many system optimization problems are characterized by delayed, sparse, aggregated or sequential rewards, where improving the long-term sum of rewards is more important than a single immediate reward. This behavior is inherent in deep RL. In the compiler optimization case, the observation from the environment is the program state and/or the optimization applied so far. The action is the optimization pass to apply next, and the reward is the performance improvement.

In this thesis, deep RL and other machine learning methods are applied to solve complex compiler optimization tasks: phase ordering, vectorization, scheduling, and cache allocation. With collaborators from Berkeley, other universities, and companies, the AutoPhase [13, 28], NeuroVectorizer [18, 29], RLDRM [30], ProTuner [31], and Ansor [32] systems were built, open sourced, and used by the community to solve key compiler optimization problems. Unlike prior works that focus on hand-engineered heuristics and program features, or other machine learning methods, the deep RL methods proposed in this thesis address this problem with end-to-end solutions, which learn from prior experience, efficiently explore the search space, and quickly find better optimizations.

The thesis is organized as follows:

- Chapter 2 gives an overview of deep RL algorithms and other machine learning methods used in this thesis, background on compiler optimization, and related work.

- Chapter 3 surveys the recent trends in deep RL algorithms for system optimization. Despite the growing adoption of these solutions, a review of prior work shows the clear lack of standardized metrics for assessing their performance. Therefore, quintessential metrics to guide future work in evaluating the use of deep RL in system optimization are also introduced in this chapter, and the multiple challenges faced when integrating deep RL into systems are discussed. These metrics guided the application of deep

RL in compiler optimization throughout the thesis. RLDRM, which uses deep RL to automatically optimize last-level cache allocation, is also introduced in this chapter.

- Chapter 4 introduces AutoPhase: a system for solving the NP-Hard compiler phase ordering challenge with deep RL. AutoPhase is implemented, evaluated, and open sourced.

- Chapter 5 introduces NeuroVectorizer: an open source system that uses deep RL to automatically and directly vectorize code from text, achieving 97% of the optimal performance and running more than 14$\times$ faster than a state-of-the-art algorithm that uses supervised learning.

- Chapter 6 introduces ProTuner: a system for program scheduling that combines both phase ordering and vectorization, in addition to multithreading, tiling, and loop unrolling. ProTuner uses Monte Carlo tree search (MCTS). Unlike the current state-of-the-art, MCTS makes decisions by looking ahead, evaluating complete schedules, avoiding greediness, and considering the expected long-term reward of scheduling decisions, which also makes it more resilient to noise in the cost model. Ansor, which is similar to ProTuner but uses a scheduler and fine-tuner in conjunction with evolutionary algorithms, is also discussed.

- Chapter 7 concludes the thesis and discusses future research directions.

Chapter 2

Background

Markov Decision Processes

A Markov decision process (MDP) is a discrete stochastic control process that models sequential decision-making in fully observable environments. It assumes the Markov property that the impact of one decision made in a state depends only on that state and not on the prior decision history. An MDP model consists of:

- S: A set of possible states, with s_0 representing the initial state.
- A: A set of possible actions.
- $R(s, a)$: A reward function.
- $T(s'|s, a)$: A transition function that models the probability of getting to state s' given an action a in state s.

Solving an MDP means finding a policy $\pi(s)$ that chooses an action to apply based on the current state and optimizes for the overall expected reward. The decision-making process is modeled as a sequence of state and action pairs (s, a). The next state s' can either be decided deterministically by the pair (s, a) or stochastically by a probability distribution $p(s'|s, a)$.

Deep Reinforcement Learning Algorithms

One promising machine learning approach for solving MDPs is reinforcement learning (RL), in which an agent learns by continually interacting with an environment [20]. In RL, the agent observes the state of the environment and performs an action based on this state/observation. The ultimate goal is to compute a policy—a mapping between the environment states and actions—that maximizes the expected reward. RL can be viewed as a stochastic optimization solution for Markov Decision Processes (MDPs) [19], when the MDP is not known. An MDP is defined by a tuple with four elements: $S, A, P(s, a), r(s, a)$, where S is the set of states of the environment, A describes the set of actions or transitions between states, $s' \sim P(s, a)$

describes the probability distribution of next states given the current state and action, and $r(s,a) : S \times A \rightarrow R$ is the reward for taking action a in state s. Given an MDP, the goal of the agent is to gain the largest possible cumulative reward. The objective of an RL algorithm associated with an MDP is to find a decision policy $\pi^*(a|s) : s \rightarrow A$ that achieves this goal for that MDP:

$$\begin{gathered}\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi(\tau)} \left[\tau\right] = \\ \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi(\tau)} \left[\sum_t r(s_t, a_t)\right],\end{gathered} \tag{2.1}$$

where τ is a sequence of states and actions that define a single episode, and T is the length of that episode. Deep RL leverages a neural network to learn the policy (and sometimes the reward function). In recent years, a plethora of new deep RL techniques have been proposed [33–37].

Policy Gradient (PG) [35], for example, uses a neural network to represent the policy. This policy is updated directly by differentiating the term in Equation 2.1 as follows:

$$\begin{aligned}\nabla_\theta J = \nabla_\theta \mathbb{E}_{\tau\sim\pi(\tau)} &\left[\sum_t r(s_t, a_t)\right] \\ &= \mathbb{E}_{\tau\sim\pi(\tau)} \left[\left(\sum_t \nabla_\theta log\pi_\theta(a_t|s_t)\right)\left(\sum_t r(s_t, a_t)\right)\right] \\ &\approx \frac{1}{N}\sum_{i=1}^{N} \left[\left(\sum_t \nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_t r(s_{i,t}, a_{i,t})\right)\right]\end{aligned} \tag{2.2}$$

and updating the network parameters (weights) in the direction of the gradient:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J. \tag{2.3}$$

Asynchronous Advantage Actor-critic (A3C) [33] uses an actor (usually a neural network) that interacts with the critic, which is another network that evaluates the action by computing the value function. The critic tells the actor how good its action was and how it should adjust. The update performed by the algorithm can be expressed as $\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})\hat{A}_t$.

Proximal Policy Optimization (PPO) [36] is a variant of PG that enables multiple epochs of minibatch updates to improve the sample complexity. PPO obtains more deterministic, stable, and robust behavior over PG. Vanilla PG performs one gradient update per data sample while PPO uses a novel surrogate objective function to enable multiple epochs of minibatch updates. It alternates between sampling data through interaction with the environment and optimizing the surrogate objective function using stochastic gradient ascent. It performs updates that maximize the reward function using the surrogate objective function to ensure that the deviation from the previous policy is small. The loss function of PPO is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{2.4}$$

where $r_t(\theta)$ is defined as a probability ratio $\frac{\pi_\theta(\mathbf{a_t}|\mathbf{s_t})}{\pi_{\theta_{old}}(\mathbf{a_t}|\mathbf{s_t})}$ so that $r(\theta_{old}) = 1$. This term penalizes policy updates that move $r_t(\theta)$ from $r(\theta_{old})$. $\hat{A}_t$ denotes the estimated advantage that approximates how good $\mathbf{a_t}$ is compared to the average. The second term in the *min* function acts as a disincentive for moving r_t outside of $[1-\varepsilon, 1+\varepsilon]$, where ε is a hyperparameter.

In contrast, Q-Learning [38], state-action-reward-state-action (SARSA) [39], and deep deterministic policy gradient (DDPG) [37] are temporal difference methods, *i.e.*, they update the policy on every timestep (action) rather than on every episode. Furthermore, these algorithms bootstrap and, instead of using a neural network for the policy itself, they learn a Q-function, which estimates the long-term reward from taking an action. The policy is then defined using this Q-function. In Q-Learning the Q-function is updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + r(s_t, a_t) + \gamma max_{a'_t}[Q(s'_t, a'_t)]. \tag{2.5}$$

In other words, the Q-function updates are based on the action that maximizes the value of that Q-function. On the other hand, in SARSA, the Q-function is updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}). \tag{2.6}$$

In this case, the Q-function updates are based on the action that the policy would select given state s_t. DDPG fits multiple neural networks to the policy, including the Q-function and target time-delayed copies that slowly track the learned networks and greatly improve stability in learning. Upper confidence bound and greedy algorithms can then be used to determine the policy based on the Q-function [21, 40]. The reviewed works in this thesis focus on the epsilon greedy method, where the policy is defined as follows:

$$\pi^*(a_t|s_t) = \begin{cases} \arg\max_{a_t} Q(s_t, a_t), & \text{w.p. } 1-\epsilon \\ \textit{random action}, & \text{w.p. } \epsilon. \end{cases} \tag{2.7}$$

A method is considered to be on-policy if the new policy is computed directly from the decisions made by the current policy. PG, PPO, A3C, and SARSA are thus on-policy while DDPG and Q-Learning are off-policy. All the mentioned methods are model-free: they learn directly from the environment by trial and error. If a model is nonetheless available or can be learned, it could be used for planning and enable more robust training as less interaction with the environment would be required.

Multi-armed Bandits

Multi-armed bandits [41, 42] simplify RL by removing the learning dependency on the state and thus providing evaluative feedback that depends entirely on the action taken (1-step RL problems). The actions usually are decided upon in a greedy manner by independently updating the benefit estimates of performing each action. To also consider the state, contextual bandits may be used [43]. Bandit solutions often perform as well as more complicated RL solutions or even better. Many bandit algorithms enjoy stronger theoretical guarantees on their performance even under adversarial settings. These bounds would likely be of great value to the systems world as they imply in the limit that the proposed algorithm would be no worse than using the best fixed system configuration in hindsight.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) solves MDPs by combining tree search with random sampling to find the optimal decisions in the MDP. The tree is built incrementally using selection, expansion, simulation, and backpropagation. A *tree policy* is used to *select* a node to expand at each iteration of the algorithm. This policy should balance the exploration and exploitation of the search algorithm. The node is *expanded* and a *simulation* is then run from the selected node to collect the rewards of the terminal state. The decisions made during the simulation are determined by a *default policy*, which can be uniform random sampling in its simplest form. This is the policy used in this thesis. Finally, MCTS backpropagates the reward and updates the statistics of the ancestor nodes. In this thesis, the tree selection policy used is the UCB [42]:

$$UCB = \bar{X}_j + 2C_p\sqrt{\frac{2ln(n)}{n_j}} \tag{2.8}$$

where n is the number of times the current parent node has been visited, n_j is the number of times child j has been visited, and $C_p > 0$ is a constant. $\bar{X}_j$ is the average reward of the simulations. The left term ($\bar{X}_j$) tracks exploitation while the right term tracks exploration. Increasing C_p will add more exploration, and decreasing it will reduce exploration.

According to [44], MCTS offers significant advantages over alpha-beta pruning, which minimizes the search space when there is no good evaluation function. The evaluation of moves in MCTS has proven it to converge to Minimax.

Evolutionary Algorithms

Evolutionary algorithms are another technique that can be used to search for the best compiler pass ordering. They include a family of population-based meta-heuristic optimization algorithms inspired by natural selection. The main idea of these algorithms is to sample a population of solutions and use the good ones to direct the distribution of future generations. Two commonly used evolutionary algorithms are genetic algorithms (GA) [45] and evolution strategies (ES) [46].

GA generally requires a genetic representation of the search space where the solutions are coded as integer vectors. The algorithm starts with a pool of candidates, then iteratively evolves the pool to include solutions with better fitness using the strategies of selection, crossover, and mutation. Selection keeps a subset of solutions with the highest fitness values. These selected solutions act as parents for the next generation. Crossover merges pairs from the parent solutions to produce new offspring. Mutation perturbs the offspring solutions with a low probability. The process repeats until a solution that reaches the goal fitness is found or after a certain number of generations.

ES works similarly to GA. However, the solutions are coded as real numbers in ES. In addition, ES is self-adapting. The hyperparameters, such as the step size or the mutation probability, are different for different solutions. They are encoded in each solution, so good settings get to the next generation with good solutions. Recent work [47] has used ES to update policy weights for RL and showed it is a good alternative for gradient-based methods.

Beam Search

Beam search [48] is a heuristic algorithm that explores a decision tree and searches for the optimal decisions by expanding a limited number of children with the highest intermediate rewards. It is widely used for the sequential decision-making process, for example in speech recognition [48] and software scheduling [12]. It builds its search tree with a breadth-first search. At each step of the algorithm, it exhaustively evaluates all the direct children, sorts the children based on the intermediate rewards, and keeps the top-k children as the parent nodes for the next iteration. k is the beam size that determines the total number of top children to keep at every iteration. It is essentially a greedy algorithm and thus can get stuck in local optima.

Compiler Phase-ordering

Compilers execute optimization passes such as inlining, dead code elimination, loop unrolling, loop invariant code motion, vectorization, and loop fusion to transform programs into more efficient forms to run on various hardware targets. This is done by applying a sequence of analysis and optimization phases, where each phase in this sequence consumes the output of the previous phase and generates a modified version of the program for the next phase. The performance of the code a compiler generates depends on the order in which it applies the optimization passes. Furthermore, these phases are not commutative, which makes the order in which they are applied critical to the performance of the generated program. Choosing a good order—often referred to as the *phase-ordering* problem—is an NP-hard problem.

While these optimization levels offer a simple set of choices for developers, they are handpicked by the compiler-designers and most often benefit certain groups of benchmark programs. The compiler community has attempted to address the issue by selecting a particular set of compiler optimizations on a per-program or per-target basis for software [1–4].

Compiler Auto-Vectorzation

Vectorization is critical to enhancing the performance of compute-intensive workloads in modern computers. All the dedicated vector machines and modern CPUs that support vector instructions rely on vectorization to enhance the performance of such workloads.

Loops are among the most commonly vectorized parts of the code. Loop vectorization is done by setting the VF and the IF, which respectively determine the number of instructions to pack together and the stride. Appropriately setting the values of VF and IF for loops is cumbersome as it depends on many parameters, such as the instructions in the loop body, the stride, the underlying hardware architecture, the computations graph, and the functionality.

Most C and C++ compilers allow the users to manually determine the VF and the IF in their code. This, however, is time-consuming and error-prone. Thus, many works have tried to address the automatic vectorization challenge. For example, Polly [15] uses an abstract mathematical representation based on integer polyhedra to analyze and optimize the memory access pattern of a program. Polly performs classical loop transformations, especially tiling and loop fusion, to improve data locality. These transformations also simplify vectorization decisions for the compiler.

Prior work [49] represented the code characteristics using hand-engineered heuristics extracted from the assembly code, such as arithmetic intensity, used in conjunction with supervised learning to predict the vectorization factors. Unfortunately, these features are typically not sufficient to fully capture the code functionality [50]. To overcome this challenge, the authors of [51] proposed an end-to-end solution that relies on deep supervised learning. However, supervised learning methods require labels to train and finding these labels can be time-consuming. Furthermore, optimizing for multiple objectives with large search spaces can be challenging for supervised learning methods.

Automatic Cache Allocation

Modern CPUs use set associative cache, which comprises multiple sets, each comprising multiple ways. Useful data in one application can be evicted by another application whose data falls into the same set, following a certain replacement policy. For example, one very popular replacement policy is the least-recently-used (LRU) policy, which means that the least recently used cache line is evicted in the case of replacement. In a modern CPU chip, multiple cores share a single logical last level cache (LLC). Applications running simultaneously on different cores will compete for the limited LLC capacity. In other words, one application may evict the useful data of another application from the same cache set, thus interfering with its performance.

To alleviate the resource contention problem in multicore systems, numerous software and hardware solutions have been proposed [52–63]. One of the most mature technologies available today is Intel® Resource Director Technology or Intel® RDT [64]. Intel® RDT provides the capability to partition LLC to restrict the usage of each co-running application.

Halide Scheduling

Halide [65] is a domain-specific language for image processing and deep learning tasks. Halide's language abstraction decouples the algorithmic descriptions of the target image-processing workloads from a specific mapping of the workload on hardware, which we refer to as a "schedule". This abstraction provides the user with a clearly defined scheduling space and makes it easier to explore different schedules automatically. Many decisions need to be made in a Halide schedule, including the execution order of different functions, vectorization factors, tiling factors, inlining, memory allocation strategies, etc. The overall scheduling space is intractable and expert scheduling can be hard to develop. Therefore, automatic generation of high-performance Halide schedulings has been implemented and studied in several prior works [9–12].

TVM

TVM [66] is an open-source automated end-to-end optimizing compiler for deep learning workloads running on CPUs, GPUs, and machine learning accelerators. TVM enables machine learning engineers to optimize and run computations efficiently on multiple hardware backends. TVM provides performance portability across diverse hardware back-ends by relying on operator-level and graph-level optimizations, and it solves optimization challenges specific to deep learning, such as operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. It also automates optimization of low-level programs to hardware characteristics by using a learning-based cost modeling method for rapid exploration of code optimizations.

Chapter 3

Deep Reinforcement Learning in System Optimization

3.1 Introduction

Reinforcement learning (RL) is a class of learning problems framed in the context of planning on a Markov Decision Process (MDP) [19], when the MDP is not known. In RL, an agent continually interacts with the environment [20, 21]. In particular, the agent observes the state of the environment and performs an action in accordance with the observed state. The goal of the RL agent is then to compute a policy—a mapping between the environment states and actions—that maximizes a long term reward. There are multiple ways to extrapolate the policy. Non-approximation methods usually fail to predict good actions in states that were not visited in the past, and require storing all the action-reward pairs for every visited state, a task that incurs a huge memory overhead and complex computation. Approximation techniques are a promising alternative, and a top contender among these is Deep RL: by using a neural network in conjunction with RL, unstructured environments can be successfully handled, complex functions learned, and actions predicted in states not visited in the past. Deep RL can thus provide end-to-end solutions for complex problems that non-approximation techniques cannot solve. Deep RL is gaining wide interest due to its success in robotics and Atari games, and for its superhuman performance capabilities [22–26].

Many system optimization problems are characterized by delayed, sparse, aggregated or sequential rewards, where improving the long-term sum of rewards is more important than a single immediate reward. For example, an RL environment can be a computer cluster. The state could be defined as a combination of the current resource utilization, available resources, time of the day, duration of jobs waiting to run, *etc.* The action could be to determine on which resources to schedule each job. The reward could be the total revenue, jobs served in a time window, wait time, energy efficiency, *etc.*, depending on the objective. In this example, if the objective is to minimize the waiting time of all jobs, then a good solution must interact with the computer cluster and monitor the overall wait time of the jobs to determine good schedules. This behavior is inherent in RL. The RL agent has the advantage of not requiring expert labels or knowledge, being able instead to learn directly from its own interaction with the world. RL can also learn sophisticated system characteristics that cannot be learned by a straightforward solution such as a first-come first-served allocation scheme. For instance, it could be better to put earlier long-running arrivals on hold if a shorter job requiring fewer resources is expected shortly.

In this chapter, different attempts to overcome system optimization challenges with the use of deep RL are reviewed. Unlike previous reviews [16, 17, 67–70] that focus on machine learning methods without discussing deep RL models or applying them beyond a specific system problem, we focus on deep RL in system optimization in general. Reviewing previous work clearly shows that standardized metrics for assessing deep RL solutions in system optimization problems are lacking. Thus, quintessential metrics to guide future work in evaluating the use of deep RL in system optimization are proposed. Different deep RL methods and neural network models are suggested for different use cases. Multiple challenges faced when integrating deep RL into systems are discussed and addressed. We also introduce and briefly discuss the RLDRM system, in which we used deep RL to address the challenge of dynamically adjusting cache allocation to optimize network function virtualization (NFV). Training the deep RL model on a real machine allows us to overcome stability issues and maximize the long-term reward, thus successfully overcoming some of the key challenges that will be detailed in Section 3.7.

3.2 Background

Most RL methods considered in this review are structured around value function estimation (*e.g.*, Q-values) and using gradients to update the policy. However, this is not always the case. For example, genetic algorithms, simulated annealing, genetic programming, and other gradient-free optimization methods – often called *evolutionary* methods [21] – can also solve RL problems in a manner analogous to the way biological evolution produces organisms with skilled behavior. Evolutionary methods can be effective if the space of policies is sufficiently small, the policies are common and easy to find, and the state of the environment is not fully observable. This review considers only the deep versions of these methods, *i.e.*, using a neural network in conjunction with evolutionary methods typically used to evolve and update the neural network parameters.

Prior RL Works With Alternative Approximation Methods

Multiple prior works have proposed to use non-deep neural network approximation methods for RL in system optimization. These works include reliability and monitoring [71–73], memory management [74–77] in multicore systems, congestion control [78, 79], packet routing [80–82], algorithm selection [83], cloud caching [84], energy efficiency [85], and performance [86–90]. Instead of using a neural network to approximate the policy, these works used tables, linear approximations, and other approximation methods to train and represent the policy. Tables were generally used to store the Q-values, *i.e.*, one value for each action/state pair used in training, and this table becomes the ultimate policy. In general, deep neural networks allow for more complex forms of policies and Q functions [91], and can better approximate good actions in new states.

3.3 Deep Reinforcement Learning in System Optimization

In this section, the different system challenges tackled using RL are discussed and categorized as (1) *episodic tasks*, in which the agent-environment interaction naturally breaks down into a sequence of separate terminating episodes, and (2) *continuing tasks*, in which it does not. For example, when optimizing resources in the cloud, the jobs arrive continuously and there is not a clear termination state. But when optimizing the order of SQL joins, the query has a finite number of joins, and thus after enough steps the agent arrives at a terminating state.

Continuing Tasks

Important features of RL are that it can learn from sparse reward signals, it does not need expert labels, and it has the ability to learn direction from its own interaction with the world. Jobs in the cloud arrive continuously and unpredictably. This might explain why many system optimization challenges tackled with RL are in the cloud [92–99]. A good job scheduler in the cloud should make decisions that are good in the long term. Such a scheduler should sometimes forgo short-term gains in an effort to realize greater long-term benefits. For example, it might be better to delay a long running job if a short running job is expected to arrive soon. The scheduler should also adapt to variations in the underlying resource performance and scale in the presence of new or unseen workloads combined with large numbers of resources.

These schedulers have a variety of objectives, including minimizing the average execution time of jobs and optimizing the resource allocation of virtual machines [92, 95, 98, 99], optimizing data caching on edge devices and base stations [93, 94], and maximizing energy efficiency [96, 97]. The RL algorithms used for addressing each system problem are listed in Table 3.1.

Interestingly, for cloud challenges most works are driven by Q-learning (or the very similar SARSA). In the absence of a complete environmental model, model-free Q-Learning can be used to generate optimal policies. It is able to make predictions incrementally by bootstrapping the current estimate with previous estimates and it provides good sample efficiency [100]. Q-Learning is also characterized by inherent continuous temporal difference behavior, where the policy can be updated immediately after each step (not at the end of a trajectory), something that might be very useful for online adaptation.

Episodic Tasks

Due to the sequential nature of decision-making in RL, the order of the actions has a major impact on the rewards the RL agent collects. The agent can thus learn these patterns and select more rewarding actions. Previous works took advantage of this behavior in RL to optimize congestion control [101, 102], decision trees for packet classification [103], sequence to SQL/program translation [104–106], ordering of SQL joins [107–110], compiler phase ordering [13, 28, 111], and device placement [112, 113].

After enough steps in these problems, the agent will always arrive at a clear terminating step. For example, in query join order optimization, the number of joins is finite and known from the query. In congestion control – where the routers need to adapt the sending rates to provide high throughput without compromising fairness – the updates are performed on a fixed number of senders/receivers known in advance. These updates combined define one episode. This may explain why there is a trend towards using PG methods for these types of problems, as they do not require continuous temporal difference behavior and can often operate in batches of multiple queries. Nevertheless, in some cases, Q-learning is still used, mainly for sample efficiency, as the environment step might take a long time.

The performance of PG methods can be improved by taking advantage of the way the gradient computation is performed. If the environment is not needed to generate the observation, many environment steps can be saved by rolling out the entire episode by interacting only with the policy and performing one environment step at the very end. The sum of rewards will be the same as the reward received from this environment step. For example, in query optimization, since the observations are encoded directly from the actions, and the environment is mainly used to generate the rewards, it will be possible to repeatedly perform an action, form the observation directly from this action, and feed it to the policy network. After the end of the episode, the environment can be triggered to get the final reward, which would be the sum of the intermediate rewards. This can significantly reduce the training time.

Discussion: Continuous vs. Episodic

Continuous policies can handle both continuous and episodic tasks, while episodic policies cannot. So, for example, Q-Learning can handle all the tasks mentioned in this work, while PG based methods cannot directly handle them without modification. For example, in [92], the authors limited the scheduler window of jobs to M, allowing the agent in every time step to schedule up to M jobs out of all arrived jobs. The authors also discussed the "bounded time horizon" problem and hoped to overcome it by replacing the time-dependent baseline with a value network. It is interesting to note that prior work on continuous system optimization tasks using non-deep RL approaches [80–82, 84–89] used Q-Learning.

Continuous problems without episode boundaries can be handled with PG based methods by defining performance in terms of the average rate of reward per time step [21] (Chapter

13.6). Such approaches can help better fit the continuous problems to episodic RL algorithms.

3.4 Formulating the RL environment

Table 3.1 lists all the works we reviewed and their problem formulations in the context of RL, *i.e.*, the model, observations, actions, and reward definitions. Among the major challenges when formulating the problem in the RL environment is properly defining the system problem as an RL problem, with all of the required inputs and outputs, *i.e.*, state, action spaces, and rewards. The rewards are generally sparse and behave similarly for different actions, making the RL training ineffective due to bad gradients. The states are generally defined using hand-engineered features in an attempt to encode the state of the system. This results in a large state space with some features that are less helpful than others and rarely captures the actual system state. Using model-based RL can alleviate this bottleneck and provide better sample efficiency. In [97], auto-encoders were used to help reduce the state dimensionality. The action space is also large but generally represents actions that are directly related to the objective. Another challenge is the environment step. Some tasks require a long time for the environment to perform one step, significantly slowing the learning process of the RL agent.

Interestingly, most works focus on using simple out-of-the-box FCNNs, while some works that targeted parsing and translation ([104–106]) used RNNs [115] due to their ability to parse strings and natural language. While FCNNs are simple and can be easily trained to learn linear and non-linear function policy mappings, a more complicated network structure tailored to the problem can sometimes further improve the results.

Evaluation Results

Table 3.2 lists training data and the evaluation results of the reviewed works. The evaluation considers the time it takes to perform a step in the environment, the number of steps needed in each iteration of training, number of training iterations, total number of steps needed, and whether the prior work improves the state of the art and provides a comparison against random search/bandit solution.

The total number of steps and the cost of each environment step is important to understand the sample efficiency and practicality of the solution, especially when considering RL's inherent sample inefficiency [116, 117]. For different workloads, the number of samples needed varies from thousands to millions. The environment step time also varies from milliseconds to minutes. In multiple cases, the interaction with the environment is very slow. Note that in most cases when the environment step time was a few milliseconds, it was because it was a simulated environment, not a real one. For faster environment steps, more training samples can be collected to further improve the performance. This excludes [97], where a cluster was used and thus more samples could be processed in parallel.

Table 3.1: Deep RL problem formulation. Model abbreviations: FCNN, fully connected neural network; CNN, convolutional neural network; RNN, recurrent neural network; GNN, graph neural network; GRU, gated recurrent unit; LSTM, long short-term memory.

Description	Work	State/Observation	Action	Reward	Objective	Algorithm
congestion control	[101]^1 [102]^2	histories of sending rates and resulting statistics (e.g., loss rate)	changes to sending rate	throughput and negative of latency or loss rate	maximize throughput while maintaining fairness	PPO^{1,2}/PG^2/ DDPG^2 + FCNN
packet classification	[103]	encoding of the tree node, *e.g.*, split rules	cutting a classification tree node or partitioning a set of rules	classification time /memory footprint	build optimal decision tree for packet classification	PPO + FCNN
SQL join order optimization	[107]^1 [108]^2 [110]^3 [109]^4	encoding of current join plan	next relation to join	negative cost$^{1-3}$, $1/cost^4$	minimize execution time	Q-Learning$^{1-3}$/ PPO^4 + tree conv.^{3}, FCNN$^{1-4}$
sequence to SQL	[104]	SQL vocabulary, question, column names	query corresponding to the token	-2 invalid query, -1 valid but wrong, +1 valid and right	tokens in the WHERE clause	PG + LSTM
language to program translation	[105]	natural language utterances	a sequence of program tokens	1 if correct result 0 otherwise	generate equivalent program	PG + LSTM, FCNN
semantic parsing	[106]	embedding of the words	a sequence of program tokens	positive if correct 0 otherwise	generate equivalent program	PG + RNN, GRU
resource allocation	[92]	current allocation of cluster resources & resource profiles of waiting jobs	next job to schedule	$\Sigma_i(\frac{-1}{T_i})$ for all jobs in the system (T_i is the duration of job i)	minimize average job slowdown	PG + FCNN
	[93, 94]	status of edge devices/base stations /content caches	which station, to offload/cache or not	total revenue	maximize total revenue	Q-Learning + CNN
	[95]	current allocation & demand	next resource to allocate	payments	maximize revenue	Q-Learning + FCNN
resource allocation for radio access networks	[96]	active remote radio heads & user demands	which remote radio heads to activate	negative power consumption	power efficiency	Q-Learning + FCNN
resource allocation & power management	[97]	current allocation & demand	next resource to allocate	linear combination of total power , VM latency, & reliability metrics	power efficiency	Q-Learning + autoencoder, weight sharing & LSTM
virtual machine configuration	[99] [98]	current resource allocations	increase/decrease CPU/time/ memory	throughput -response time	maximize performance	Q-Learning + FCNN, model-based
compiler phase ordering	[111]^1 [13, 28]^2	program features	next optimization pass	performance improvement	minimize execution time	Evolutionary Methods^1/ Q-Learning^2/PG^2 + FCNN
device placement	[113]^1 [112]^2	computation graph	placement of graph node	speedup	maximize performance & minimize peak memory	PG^{1,2}/Evolutionary Methods^1 + GNN/FCNN
distributed instruction placement	[114]	instruction features	instruction placement location	speedup	maximize performance	Evolutionary Methods + FCNN

Table 3.2: Evaluation results.

Problem	Work	Environment Step Time	# of Steps Per Iteration	# of Training Iterations	Total # of Steps	Improves State of the Art	Compares Against Random Search
packet classification	[103]	20-600ms	$\leq$ 60K	$\leq$ 167	1,002,000	✓(18%)	×
congestion control	[101]	50-500ms	8192	1200	9,830,400	✓(similar)	✓
	[102]	0.5s	N/A	N/A	50K-100K	×	×
resource allocation	[92]	10-40ms	20K	1K	20,000,000	✓(10-63%)	✓
	[93] [94]	N/A	N/A	20K	N/A	N/A	×
	[95]	N/A	N/A	10K-20K	N/A	N/A	✓
	[96]	N/A	N/A	N/A	N/A	N/A	✓
	[97]	1-120 minutes	100K	20	2,000,000	N/A	×
	[99] [98]	N/A	N/A	N/A	N/A	N/A	×
SQL Joins	[107]	10ms	640	100	64K	✓(70%)	✓
	[108]	N/A	N/A	N/A	N/A	N/A	×
	[110]	250ms	100-8K	100	10K-80K	✓(10-66%)	✓
	[109]	1.08s	N/A	N/A	10K	✓(20%)	✓
sequence to SQL	[104]	N/A	80,654	300	24,196,200	✓(similar)	×
language to program translation	[105]	N/A	N/A	N/A	13K	✓(56%)	×
semantic parsing	[106]	N/A	3,098	200	619,600	✓(3.4%)	×
compiler phase ordering	[13, 28]	1 second	N/A	N/A	1K-10K	✓(similar)	✓
	[111]	13.2 days for all steps	N/A	N/A	N/A	×	×
device placement	[112]	N/A (seconds)	N/A	N/A	1.6K-94K	✓(3%)	✓
	[113]	N/A (seconds)	N/A	N/A	400K	✓(5%)	✓
instruction placement	[114]	N/A (minutes)	N/A	200	N/A (days)	×	×

As listed in Table 3.2, many works did not provide sufficient data to reproduce the results. Reproducing the results is necessary to further improve the solution and enable future evaluation and comparison against it.

Frameworks and Toolkits

A few recent RL benchmark toolkits assist in the development and comparison of reinforcement learning algorithms, and provide a faster simulated system environment. OpenAI Gym [118] supports an environment for teaching agents everything from walking to playing games such as Pong or Pinball. Iroko [102] provides a data center emulator to understand the requirements and limitations of applying RL in data center networks. It interfaces with the OpenAI Gym and offers a way to evaluate centralized and decentralized RL algorithms against conventional traffic control solutions.

Park [119] proposes an open platform that facilitates formulation of the RL environment for twelve real-world system optimization problems with one easy to use API. The platform provides a translation layer between the system and the RL environment, making it easier for RL researchers to work on systems problems. That being said, actions, state, and reward definitions cannot be changed in this framework, making it harder to improve the performance.

3.5 Considerations for Evaluating Deep RL in System Optimization

In this section, we propose a set of questions that can help system optimization researchers determine whether deep RL could be an effective tool in solving their systems optimization challenges. We also provide examples on how we applied them to AutoPhase in Chapter 4 and to NeuroVectorizer in Chapter 5.

Can the System Optimization Problem Be Modeled by an MDP?

MDPs are a classic formalization of sequential decision-making, where actions influence not only immediate rewards but also future states and rewards. This involves delayed rewards and the trade-off between delayed and immediate rewards. In MDPs, the new state and new reward are dependent only on the preceding state and action. Given a perfect model of the environment, an MDP can compute the optimal policy.

MDPs are typically a straightforward formulation of the system problem, as an agent learns by continually interacting with the system to achieve a particular goal, and the system responds to these interactions with a new state and reward. The agent's goal is to maximize the expected reward over time. If the system optimization problem can be modeled by an MDP, then RL is promising solution. For example, in AutoPhase and NeuroVectorizer, the problem is formulated as an MDP where the program features are the state, the action is the compiler optimization to apply, and the reward is the decrease in the execution time.

Can the Problem Be Best Solved by Reinforcement Learning?

RL is different from supervised learning. The latter is learning from a training set with labels provided by a knowledgeable external supervisor. For each example the label is the correct action the system should take. The objective of this kind of learning is to act correctly in new situations not encountered in the training set. However, supervised learning is not suitable for learning from interaction, as it is often impractical to obtain examples representative of all the cases in which the agent has to act. What distinguishes RL from other machine learning approaches is that it relies on self-exploration and exploitation, and thus must take into account the tradeoff between them.

In AutoPhase and NeuroVectorizer, we demonstrate that the problem can be best solved by RL as both systems require a tradeoff between exploitation and exploration. RL is also shown to converge with fewer compilations and shorter training time, while achieving promising performance.

Are the Rewards Delayed?

RL algorithms do not maximize the immediate reward of taking actions but, rather, the expected reward over time. For example, an RL agent can choose to take actions that give low immediate rewards but lead to higher rewards overall, as opposed to taking greedy actions that lead to high immediate rewards in every step but low rewards overall. If the objective is to maximize the immediate reward or the actions are not dependent, then simpler approaches, such as bandits and greedy algorithms, will perform better than deep RL. In AutoPhase, where the episode length was set to 45, the execution time is only known after performing 45 compilations. In NeuroVectorizer, however, the episode length is one. Hence, the execution time is known after applying a single optimization.

What is Being Learned?

It is important to understand what is being learned by the agent. For example, what actions are taken in which states and why? Can the knowledge learned be applied to new states/tasks? Is there a structure to the problem being learned? If a brute-force solution is possible for simpler tasks, it will also be helpful to know how much better the performance of the RL agent is than the brute force solution. In some cases, not all hand-engineered features are useful. Using all of them can result in high variance and prolonged training. Feature analysis can help overcome this challenge. For example, in [114] significant performance gaps were shown for different selected features.

In AutoPhase, the deep RL learns how to correlate the static program features with the outcome of applying a pass. In NeuroVectorizer, the deep RL agent learns the relationship between the learned vector representation of the program and the necessary action that would lead to the shortest execution time.

Does It Outperform the State of the Art?

The most important metric in system optimization in general is outperforming the state of the art. Improving the state of the art includes different objectives, such as efficiency, performance, throughput, bandwidth, fault tolerance, security, utilization, reliability, robustness, complexity, and energy. If the deep RL approach does not perform better than the state of the art in some metric, then its use is hard to justify. Moreover, the state of the art solution is frequently more stable, practical, and reliable than deep RL. In many prior works listed in Table 3.2, a comparison against the state of the art is not provided or deep RL performs worse. In some cases deep RL can perform as well as the state of the art or slightly worse, but it can still be a useful solution as it improves on other metrics such as the time it takes to navigate the search space and find the solution. Our AutoPhase and NeuroVectorizer systems are shown to outperform the state of the art.

Does It Outperform Random Search and a Bandit Solution?

In some cases, the RL solution is just another form of improved random search, and good RL results have sometimes been achieved by mere chance. If the features used to represent the state are not good or do not have a pattern that could be learned, then random search might perform as well as RL, or even better, as it is less complicated. For example, in [28], the authors showed 10% improvement over the baseline by using random search. In some cases the actions are independent and a greedy or bandit solution can achieve optimal or near-optimal results. Using a bandit method is equivalent to using a 1-step RL solution, in which the objective is to maximize the immediate reward. Maximizing the immediate reward could deliver the overall maximum reward and, thus, a comparison will reveal whether a bandit solution is preferable in this case. In our case, both AutoPhase and NeuroVectorizer are shown to outperform random search.

Are the Expert Actions Accessible?

In some cases the *expert actions*, *i.e.*, optimal actions, are accessible, for example by running a brute-force search to collect the labels if it is plausible and practical. In that case, using imitation learning [120], which is a supervised learning approach that learns by imitating expert actions, will outperform deep RL. In AutoPhase the expert actions are not accessible. This is because knowing the expert actions requires running a brute-force search on an extremely large space (2^{247} permutations), which requires years to complete. In NeuroVectorizer, on the other hand, the search space is smaller, and a brute-force search is feasible. However, the long search time of brute-force search required for supervised learning is shown to result in $14\times$ slower search time compared to deep RL.

Is It Possible to Reproduce/Generalize Good Results?

The results are reproducible and generalizable when they span a wide spectrum of the search space and a third party can reproduce them. The learning process in deep RL is stochastic; thus, good results are sometimes achieved due to local maxima, simple tasks, and chance. In [121] different results were generated by simply changing the random seeds. In many cases, good results cannot be reproduced by retraining, training on new tasks, or generalizing to new tasks. In that case, it is possible that deep RL cannot achieve good results overall.

In AutoPhase and NeuroVectorizer, the results are reproducible and the code is open sourced with scripts. An evaluation of both systems shows that the results can generalize to new unseen programs.

3.6 Deep RL Methods and Neural Network Models

Multiple RL methods and neural network models can be used to solve system optimization problems. RL frameworks such as RLlib [122], Intel's Coach [123], TensorForce [124], Facebook Horizon [125], and Google's Dopamine [126] can help the users pick the right RL model, as they provide implementations of many policies and models for which a convenient interface is available.

As a rule of thumb, we rank RL algorithms in accordance with their sample efficiency, as follows: model-based approaches (most efficient), temporal difference methods, PG methods, and evolutionary algorithms (least efficient). In general, many RL environments run in a simulator. For example, [92, 113, 119] run in a simulator as the real environment's step would take minutes or hours, which significantly slows down the training. If this simulator is fast enough or training time is not constrained, then PG methods can perform well. If the simulator is not fast enough or training time is constrained, then temporal difference methods can do better than PG methods as they are more sample efficient.

If the environment is the real one, then temporal difference can do well, as long as interaction with the environment is not slow. Model-based RL performs better if the environment is slow. Model-based methods require a model of the environment (that often can be learned) and rely mainly on planning rather than learning [127, 128]. Since planning is not done in the actual environment, but in much faster simulation steps within the model, it requires fewer samples from the real environment to learn. Many real-world system problems have well-established and often highly accurate models, which model-based methods can leverage. That being said, model-free methods are often used as they are simpler to deploy and have the potential to generalize better from exploration in a real environment.

If good policies are easy to find, and if either the space of policies is small enough or time is not a bottleneck for the search, then evolutionary methods can be effective. Evolutionary methods also have advantages when the learning agent cannot observe the complete state of the environment. As mentioned earlier, bandit solutions are good if the problem can be viewed as a one-step RL problem.

PG methods are in general more stable than methods such as Q-Learning that do not directly use and derive a neural network to represent the agent's policy. Because directly deriving the policy and moving the gradient in the direction of the objective are intuitively greedy strategies, PG methods are easier to reason about and often more reliable. However, Q-Learning can be applied to data collected from a running system more readily than PG, which must interact with the system during training.

The RL methods may be implemented using any number of deep neural network architectures. The preferred architecture depends on the nature of the observation and action spaces. CNNs that efficiently capture spatially organized observation spaces lend themselves to visual data (*e.g.*, images or video). Networks designed for sequential learning, such as RNNs, are appropriate for observation spaces involving sequence data (*e.g.*, code, queries, temporal event streams). Otherwise, FCNNs are preferred for their general applicability and ease of use, although they tend to be the most computationally intensive choice. Finally, GNNs or other networks that capture structure within observations can be used in the less frequent case that the designer has a priori knowledge of the representational structure. In this case, the model can even generate structured action spaces (*e.g.*, a query plan tree or computational graph).

3.7 Challenges

In this section, the primary challenges of applying deep RL in system optimization are discussed.

Interactions with Real Systems Can Be Slow. Generalizing from Faster Simulated Environments Can Be Restrictive. Unlike the case in simulated environments, when an action is performed in a real environment, the reward may be triggered only after a lengthy delay. For example, when scheduling jobs on a cluster of nodes – where the reward is proportional to the decrease in execution time – some jobs might require hours to run, and thus collecting the rewards will be slow. To speed up this process, some works use simulators as cost models instead of the actual system. These simulators often do not fully capture the actual behavior of the real system and thus the RL agent may not work as well in practice. More comprehensive environment models can aid generalization from simulated environments. RL methods that are more sample efficient will speed up training in real system environments.

Instability and High Variance. These are common problems that lead to bad policies when performing system optimization with deep RL. Such policies can generate a large performance gap when trained multiple times and behave in an unpredictable manner. This is mainly due to poor formulation of the problem as an RL problem, limited observation of the state, *i.e.,* the use of embeddings and input features that are not sufficient/meaningful, and sparse or similar rewards. Sparse rewards can be due to bad reward definition or to the fact that some rewards cannot be computed directly and are known only at the end of the episode. For example, in [103], where deep RL is used to optimize decision trees for packet classification, the reward (the performance of the tree) is known only when the entire tree is built, or after approximately 15,000 steps. In some cases using more robust and stable policies can help. For example, Q-learning is known to have good sample efficiency but unstable behavior. SARSA, double Q-learning [129] and policy gradient methods, on the other hand, are more stable. Subtracting a bias in PG can also help reduce variance [130].

Lack of Reproducibility. Reproducibility is a frequent challenge with many recent works in system optimization that rely on deep RL. Restricted access to the resources, code, and workloads used, lack of a detailed list of the used network hyperparameters, and lack of stable, predictable, and scalable behavior of the different RL algorithms all make the results difficult to reproduce. This challenge prevents future deployment, incremental improvements, and proper evaluation.

Defining Appropriate Rewards, Actions and States. The proper definition of states, actions, and rewards is the key to a useful RL solution. In the general use case of deep RL, defining the states, actions and rewards is much more straightforward than in the case of system optimization. For example, in Atari games, the state is an image representing the current status of the game, the rewards are the points collected while playing, and the actions are moves in the game. In system optimization, however, it is often unclear what the appropriate definitions are. Furthermore, in many cases the rewards are sparse or similar, and the states are not fully observable and have limited features that capture only a small portion of the system state. This results in unstable and inadequate policies. The action and state spaces are usually large, requiring a lot of samples to learn and resulting in instability and large variance in the learned network. Therefore, retraining often fails to generate the same results.

Lack of Generalization. The lack of generalization is also endemic to deep RL solutions. This might be beneficial when learning a particular structure. For example, in NeuroCuts [103], the target is to build the best decision tree for fixed set of predefined rules, and the objective of the RL agent is thus to find the optimal fit for these rules. However, lack of generalization sometimes results in a solution that works for a particular workload or setting but is not very good overall. This problem manifests when the RL agent has to deal with new states that it did not visit in the past. The ability to generalize is crucial in such a case. For example, in [112, 113], the RL agent has to learn good resource placements for different computation graphs. To improve generalization the authors trained the RL agent and tested it on a wide range of graphs.

Lack of Standardized Benchmarks, Frameworks and Evaluation Metrics. The lack of standardized benchmarks, frameworks and evaluation metrics makes it very difficult to evaluate the effectiveness of the deep RL methods in the context of system optimization. Thus, it is crucial to have proper standardized frameworks and evaluation metrics that define success. Moreover, benchmarks are needed that enable proper training, evaluation of the results, measuring the generalization of the solution to new problems, and valid comparisons against baseline approaches.

3.8 An Illustrative Example

Using DeepRM [92] as an illustrative example, we put all the metrics from Section 3.5 to work in order to further highlight the challenges (from Section 3.7) of implementing deep RL solutions. The targeted system problem is resource management in the cloud. The objective is to avoid job slowdown, *i.e.*, the goal is to minimize the wait time for all jobs. DeepRM uses PG in conjunction with a simulated environment rather than a real cloud environment. This significantly improves the step time but can result in restricted generalization when used in a real environment. Nonetheless, since all the simulation parameters are known, the full state of the simulated environment can be captured. The actions are defined as selecting which job should be scheduled next. The state is defined as the current allocation of cluster resources, as well as the resource profiles of jobs waiting to be scheduled. The reward is defined as the sum of job slowdowns: $\Sigma_i(\frac{-1}{T_i})$, where T_i is the pure execution time of job i without considering the wait time. This reward basically gives a penalty of -1 for jobs that are waiting to be scheduled. The penalty is divided by T_i to give a higher priority to shorter jobs.

The state, actions and reward clearly define an MDP and *a reinforcement learning problem.* The agent interacts with the system by making sequential allocations, observing the state of the current allocation of resources, and receiving delayed long-term rewards as overall job slowdowns. *The rewards are delayed* because the agent cannot know the effect of the current allocation action on the overall slowdown at any particular time step; the agent would have to wait until all the other jobs are allocated to assess the full impact. The agent then *learns which jobs to allocate in the current time step to minimize the average job slowdown, given the current resource allocation in the cloud.* Note that DeepRM also learns to withhold larger jobs to make room for smaller ones to reduce the overall average job slowdown. DeepRM is shown to *outperform random search.*

Expert actions are not available in this problem as there are no methods to find the optimal allocation decision at any particular time step. During training in DeepRM, *multiple examples of job arrival sequences were considered in order to encourage policy generalization and robust decisions*[1]. DeepRM is also shown to *outperform the state-of-the-art by* 10–63*%*[1].

Clearly, most of the challenges mentioned in Section 3.7 are manifested in DeepRM. *The interaction with the real cloud environment is slow* and thus the authors opted for a simulated environment. This has the advantage of speeding up the training but may result in a policy that does not generalize to the real environment. Unfortunately, *generalization tests in the real environment were not provided. The instability and high variance* were addressed by subtracting a bias in the PG equation. The bias was defined as the average of job slowdowns taken at a single time step across all episodes. *The implementation of DeepRM was open sourced, allowing others to reproduce the results. The rewards, actions, and states defined allowed the agent to learn a policy that performed well in the simulated environment.* Note that defining the state of the system was easier because the environment was simulated. The solution also considered multiple reward definitions, for example, $-|J|$, where J is the number of unfinished jobs in the system. This reward definition optimizes the average job completion time. The jobs evaluated in DeepRM were considered to arrive online according to a Bernoulli process. In addition, the jobs were chosen randomly, and it is unclear whether they represent real workload scenarios. This emphasizes the *need for standardized benchmarks and frameworks* to evaluate the effectiveness of deep RL methods for job scheduling in the cloud.

[1]Results provided were only in the simulated system.

3.9 RLDRM: Applying Deep RL to Dynamic Resource Management Optimization

In RLDRM [30], we used deep RL to make dynamic cache allocation adjustments for network function virtualization (NFV). NFV technology is attracting tremendous interest from the telecommunication industry and from data center operators, as it allows service providers to assign resources for virtual network functions (VNFs) on demand, achieving better flexibility, programmability, and scalability. To improve server utilization, one popular practice is to deploy best effort (BE) workloads along with high priority (HP) VNFs when the resource usage of HP VNFs is detected to be low. To achieve that goal, RLDRM dynamically adjusts the last level cache (LLC) allocation between the high priority (HP) and the best effort (BE) workloads using deep RL. The deep RL agent's goal is to ensure that the HP workloads meet the packet loss or throughput target while maximizing the performance for the BE tasks, thus improving the server utilization.

Compared to traditional hardware black boxes, VNFs can be easily scaled and configured, enabling much shorter development to production time, and reducing the cost of upgrade and maintenance. However, hardware based networking functions still have better throughput and latency in many scenarios. Industry and academia have thus worked to develop various techniques to improve the performance of the NFV platform. Some examples include software algorithm improvements [131, 132], kernel bypass technologies [133], and in-core hardware accelerators [134, 135]. Unfortunately, running multiple applications on the same server platform could result in undesirable performance interference, as happens when HP and BE interfere on the allocated LLC cache resources. We use Intel Resource Director Technology (RDT) [64] to overcome performance interference. One of the most mature technologies, RDT provides cache partitioning capability, also known as CAT, or Cache Allocation Technology [58]. We use Intel RDT in conjunction with deep RL in RLDRM to improve the resource allocation of LLC for HP and BE workloads in servers.

RLDRM Framework Overview

Figure 3.1 shows the RLDRM framework of our closed-loop system for dynamic hardware resource allocation with deep RL. It works for a platform that runs both HP and BE workloads. The HP VNF workloads are usually the user facing, latency critical workloads. Meanwhile, system schedulers schedule the BE workloads on the same server to improve server utilization. The telemetry tool periodically collects telemetry data (platform performance counters, application throughput, *etc.*) The telemetry data are then stored and processed further by the analytic and dynamic resource allocation controller to make resource allocation decisions for the next time window.

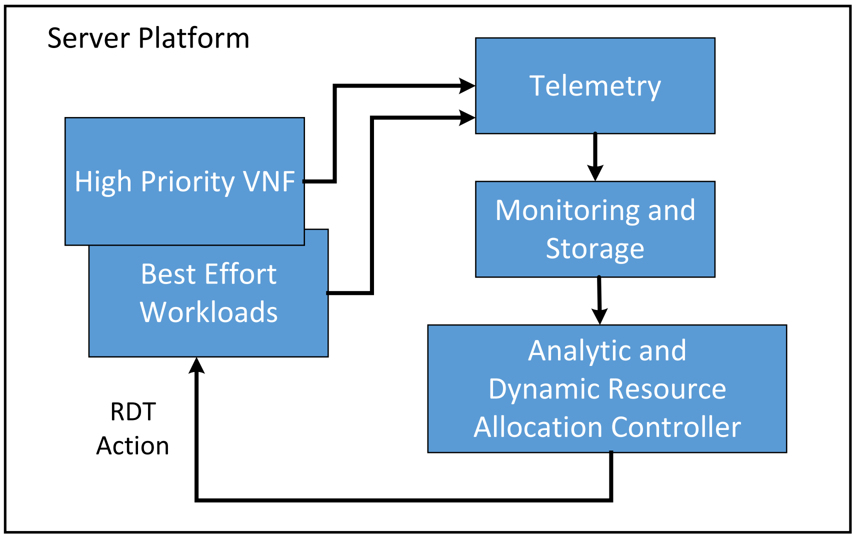


Figure 3.1: RLDRM: Closed-loop dynamic resource allocation framework.

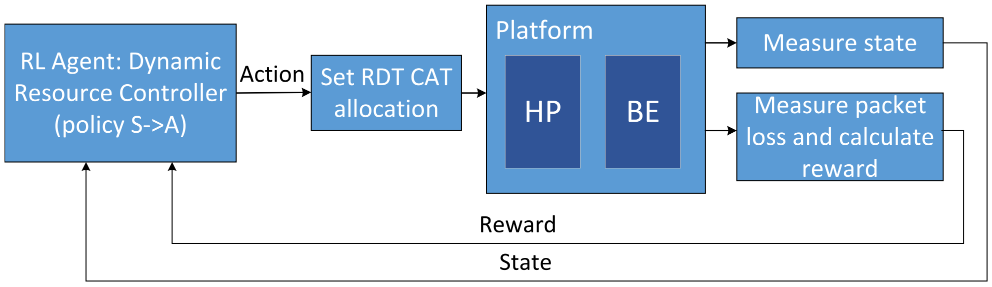

RL Agent: Dynamic Resource Controller (policy S->A)
Action
Set RDT CAT allocation
Platform
HP
BE
Measure state
Measure packet loss and calculate reward
Reward
State

Figure 3.2: Deep RL design for RDT allocation.

Deep RL design for dynamic RDT allocation

Figure 3.2 shows an overview of our proposed deep RL based framework for dynamically controlling RDT allocation among multiple workloads at run time. The RL agent continuously interacts with the system and learns a policy that maximizes the long term reward.

There are four key components in RL: (1) action, (2) state, (3) policy, and (4) reward. The policy takes the state and outputs the Q-values for each possible action. The action with the maximum Q-value is applied by the agent to the environment. After each action, a new state and reward are obtained from the system. The reward is then used to improve the policy of the agent.

The key challenge of applying deep RL to solve real world problems is to select the right algorithm for the problem, and to define the appropriate states (feature selection), actions, and the rewards. Most prior works that apply deep RL for optimization are simulator based, for example works on gaming, for which sample efficiency is less relevant. In our design, since we train the deep RL model on real machines, sample efficiency is a major consideration when choosing the algorithm. Stability is also a factor. For these reasons, we chose dueling double deep Q-learning (DDDQN) [136, 137] with prioritized experience replay as our deep RL algorithm. We also experimented with the original DQN model, but its performance and stability could not match DDDQN. Below is the detailed design for the DDDQN algorithm for controlling RDT:

Actions A: The action A is the number of RDT LLC ways allocated to HP and BE workloads for the next time window.

State S: The state S consists of the ingress traffic rate for the past N time windows, as well as the current RDT LLC way allocation to the HP VNF and the BE workload.

Reward R: In our design, the reward reflects the goal of allocating the fewest possible LLC ways for the HP workloads with the lowest possible packet loss, and allocating the remaining LLC ways to the BE workloads to improve server utilization. We designed the reward function as follows:

$$R_{pkt_loss} = \begin{cases} -m_1 & if\ pkt_{loss} > th_1 \\ -m_2 & else\ if\ pkt_{loss} > th_2 \\ -m_3 & else\ if\ pkt_{loss} > th_3 \\ +m_4 & else\ if\ pkt_{loss} <= th_3 \end{cases} \tag{3.1}$$

$$R_{rdt} = \begin{cases} LLC_{HP} & if\ pkt_{loss} > th_3 \\ Total_{LLC} - LLC_{HP} & if\ pkt_{loss} <= th_3 \end{cases} \tag{3.2}$$

$$R_{total} = R_{pkt_loss} + R_{rdt} \tag{3.3}$$

Here the pkt_{loss} is the number of packets dropped during the current time window. R_{pkt_loss} is the reward for packet loss. If the pkt_{loss} is smaller than a predefined acceptable threshold th_3 (can be either zero packet loss or low packet loss depending on the use cases), we assign a positive reward for the R_{pkt_loss}. If the packet loss is above this threshold th_3, we assign a negative reward for the R_{pkt_loss} as penalty. The greater the pkt_{loss}, the bigger the penalty ($m1 > m2 > m3$) is. R_{rdt} is the reward for LLC way allocation. When the pkt_{loss} is smaller than threshold th_3, we give higher reward for using fewer LLC ways for the HP workloads. When the pkt_{loss} is above this threshold th_3, we give higher reward for using more LLC ways for these workloads. Total reward R_{total} is the sum of R_{pkt_loss} and R_{rdt}, which considers both packet loss and LLC way allocation.

3.10 Conclusions and Future Directions

Multiple challenges in applying deep reinforcement learning to system optimization problems were reviewed and discussed in this chapter, and a set of metrics to evaluate solutions was proposed. Recent applications of deep RL in system optimization are mainly in packet classification, congestion control, compiler optimization, scheduling, query optimization and cloud computing. The growing complexity in systems requires learning-based approaches.

Deep RL presents a unique opportunity to address the dynamic behavior of systems. Applying deep RL to systems introduces a new set of challenges on how to frame and evaluate deep RL techniques. Solving these challenges will help foster the growth of optimization with deep RL. There are multiple future directions for this growth, and in particular for the deployment of deep RL solutions in system optimization tasks.

The general assumption is that deep RL may be useful in every system problem that can be formulated as a sequential decision-making process, and where meaningful action, state, and reward definitions can be provided. The optimization objective may span a wide range of options, such as energy efficiency, power, reliability, monitoring, revenue, performance, and utilization. At the processor level, deep RL could be used in branch prediction, memory prefetching, caching, data alignment, garbage collection, thread/task scheduling, power management, reliability, and monitoring. Compilers may also benefit from using deep RL to optimize the order of passes (optimizations), knobs/pragmas, unrolling factors, memory expansion, function inlining, vectorizing multiple instructions, tiling and instruction selection. With advancement of in- and near-memory processing, deep RL can be used to determine which portions of a workload should be performed in/near memory and which outside the memory.

At a higher system level, deep RL may be used in SQL/pandas query optimization, cloud computing, scheduling, caching, monitoring (e.g., temperature/failure) and fault tolerance, packet routing and classification, congestion control, FPGA allocation, and algorithm selection. While some of this has already been done, there is great potential for improvement. It is necessary to explore more benchmarks, stable and generalizable learners, transfer learning approaches, RL algorithms, and model-based RL and, more importantly, to provide better encoding of the states, actions and rewards to better represent the system and thus improve the learning. For example, with SQL/pandas join order optimization, the contents of the database are critical for determining the best order, and thus somehow incorporating an encoding of these contents may further improve the performance.

There is room for improvement in the RL algorithms as well. Some action and state spaces can dynamically change with time. For example, when a new node is added to a cluster, the RL agent will always skip the added node and it will not be captured in the environment state. Generally, the state transition function of the environment is unknown to the agent. Therefore, there is no guarantee that if the agent takes a certain action, a certain state will follow in the environment. This issue was presented in [111], where compiler optimization passes were selected using deep RL. The authors mentioned a scenario where the agent is stuck in an infinite loop of repeatedly picking the same optimization (action). This problem arose when a particular optimization did not change the features that describe the state of the environment, causing the neural network to apply the same optimization. To break this infinite loop, the authors limited the number of repetitions to five, and then applied the second best optimization instead. This was done by taking the action that corresponds to the second highest probability from the neural network's probability distribution output. In the following chapters we detail our deep RL solutions to some of the key compiler optimization challenges discussed here.

Chapter 4

AutoPhase

4.1 Introduction

High-Level Synthesis (HLS) automates the process of creating digital hardware circuits from algorithms written in high-level languages. In this chapter we propose a deep RL approach to improve the performance of HLS by optimizing the order in which the compiler applies optimization phases. Modern HLS tools [138–140] use the same front-end as the traditional software compilers. They rely on traditional software compiler techniques to optimize the input program's intermediate representation (IR) and produce circuits in the form of RTL code. Thus, the quality of compiler front-end optimizations directly impacts the performance of HLS-generated circuits. Despite a decade of research on developing sophisticated optimization algorithms, there is still a performance gap between the HLS generated code and the hand-optimized code produced by experts. A program must be just in "the right form" for a compiler to recognize the optimization opportunities. A programmer might easily perform this task, but is often difficult for a compiler.

The AutoPhase [13, 28] system introduced in this chapter was developed with the goal of overcoming the NP-hard phase ordering challenge. The system is built off the LLVM compiler [141], but the used techniques are broadly applicable to any compiler that uses a series of optimization passes. In this case, the optimization of an HLS program consists of applying a sequence of analysis and optimization phases, where each phase in this sequence consumes the output of the previous phase, and generates a modified version of the program for the next phase. Unfortunately, these phases are not commutative, which makes the order in which they are applied critical to the performance of the output.

Consider the program in Figure 4.1, which normalizes a vector. Without any optimizations, the `norm` function will take $\Theta(n^2)$ to normalize a vector. However, a smart compiler will implement the *loop invariant code motion (LICM)* [142] optimization, which allows it to move the call to `mag` above the loop, resulting in the code shown in the left-hand column of Figure 4.2. This optimization brings the runtime down to $\Theta(n)$—a big speedup improvement. Another optimization the compiler could perform is *(function) inlining* [142]. With inlining,

```
__attribute__((const))
double mag(int n, const double *A) {
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    return sqrt(sum);
}
void norm(int n, double *restrict out,
          const double *restrict in) {
    for(int i=0; i<n; i++) {
        out[i] = in[i] / mag(n, in);
    }
}
```

Figure 4.1: A simple program to normalize a vector.

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute = mag(n, in);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute, sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    precompute = sqrt(sum);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

Figure 4.2: Progressively applying LICM (left) following by inlining (right) to the code in Figure 4.1.

a call to a function is simply replaced with the body of the function, reducing the overhead of the function call. Applying inlining will result in the code shown in the right-hand column of Figure 4.2.

Now, consider applying these optimization passes in the opposite order: inlining followed by LICM. After inlining, the result is the code shown on the left in Figure 4.3. The achieved speedup is once again modest, n function calls having been eliminated, though our runtime is still $\Theta(n^2)$. If the compiler afterwards attempted to apply LICM, it would result in the code shown on the right in Figure 4.3. LICM was able to successfully move the allocation of sum outside the loop. However, it was unable to move the instruction setting `sum=0` outside the loop, as doing so would mean that all iterations excluding the first one would end up with a garbage value for sum. Thus, the internal loop will not be moved out.

As this simple example illustrates, the order in which the optimization phases are applied can be the difference between the program running in $\Theta(n^2)$ versus $\Theta(n)$. However, not only

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    for(int i=0; i<n; i++) {
        double sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double sum;
    for(int i=0; i<n; i++) {
        sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

Figure 4.3: Progressively applying inlining (left) followed by LICM (right) to the code in Figure 4.1.

is the optimal phase ordering difficult to determine, but it may also vary from program to program. Furthermore, since the problem of finding the optimal sequence of optimization phases is NP-hard, exhaustively evaluating all possible sequences is infeasible in practice. In this work, for example, the search space extends to more than 2^{247} phase orderings.

Our AutoPhase system overcomes this problem by providing a mechanism that automatically determines good phase orderings for HLS programs to optimize the circuit speed. To this end, recent advancements in deep reinforcement learning (RL) [21] were leveraged to address the phase ordering problem. With RL, a software agent continuously interacts with the environment by taking actions. Each action can change the state of the environment and generate a "reward". The goal of RL is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward. A deep RL algorithm is one that uses a deep neural network to approximate the policy. In our case, the observation from the environment could be the program and/or the optimization passes applied so far. The action is the optimization pass to apply next, and the reward is the improvement in the circuit performance after applying this pass. The particular framing of the problem as an RL problem will be a determining factor in the solution's effectiveness. Understanding how to formulate the phase ordering optimization problem in an RL framework presents significant challenges.

In this chapter, three approaches to represent the environment's state are considered. The first approach is to directly use salient features from the program. The second approach is to derive the features from the sequence of optimizations that were applied while ignoring the program's features. The third approach combines the first two approaches. To evaluate these approaches, a framework that takes a group of programs as input and quickly finds a phase ordering that competes with state-of-the-art solutions is implemented. The main contributions of this chapter are as follows:

- Leveraging deep RL to address the phase-ordering problem.
- An importance analysis on the features using random forests to significantly reduce the

state and action spaces.

- AutoPhase [13][1], a framework that integrates the current HLS compiler infrastructure with the deep RL algorithms and obtains a 28% better execution time over -O3 for nine real benchmarks.

- Demonstrating the potential of deep RL to generalize to thousands of different programs after training on one hundred programs, an accomplishment no state of the art algorithm has achieved.

4.2 Related Work

Since the search space of phase-ordering is too large for an exhaustive search, many heuristics have been proposed to explore the space by using machine learning. Huang *et al.* tried to address this challenge for HLS applications by using a modified greedy algorithm [143, 144]. Their solution achieved 16% improvement vs -O3 on the CHstone benchmarks [145]. This is the greedy algorithm evaluated in Section 4.6. In [8], both independent and Markov models were applied to automatically target an optimized search space for iterative methods to improve the search results. In [6], genetic algorithms were used to tune heuristic priority functions for three compiler optimization passes. Milepost GCC [5] used machine learning to determine the set of passes to apply to a given program, based on a static analysis of its features. It achieved an 11% execution time improvement over -O3, for the ARC reconfigurable processor on the MiBench program suite1. In [7] the challenge was formulated as a Markov process, and supervised learning was used to predict the next optimization, according to the current program state. OpenTuner [4] autotunes a program using an ensemble of AUC-Bandit-meta-technique-directed algorithms. Its current mechanism for selecting the compiler optimization passes does not consider the order or support repeated optimizations. Wang *et al.* [17] surveyed machine learning techniques for compiler optimization, where they also noted the possible benefit of using program features.

4.3 AutoPhase Framework for Automatic Phase Ordering

This section describes how AutoPhase was built by leveraging an existing open-source HLS framework called LegUp [140] that compiles a C program into a hardware RTL design. In [143], an approach based on LegUp is devised to quickly determine the number of hardware execution cycles without requiring time-consuming logic simulation. The RL simulator environment was developed using the existing harness provided by LegUp, and the final results were validated by going through the time-consuming logic simulation. AutoPhase takes

[1]AutoPhase is open-sourced under `https://github.com/ucb-bar/autophase`.

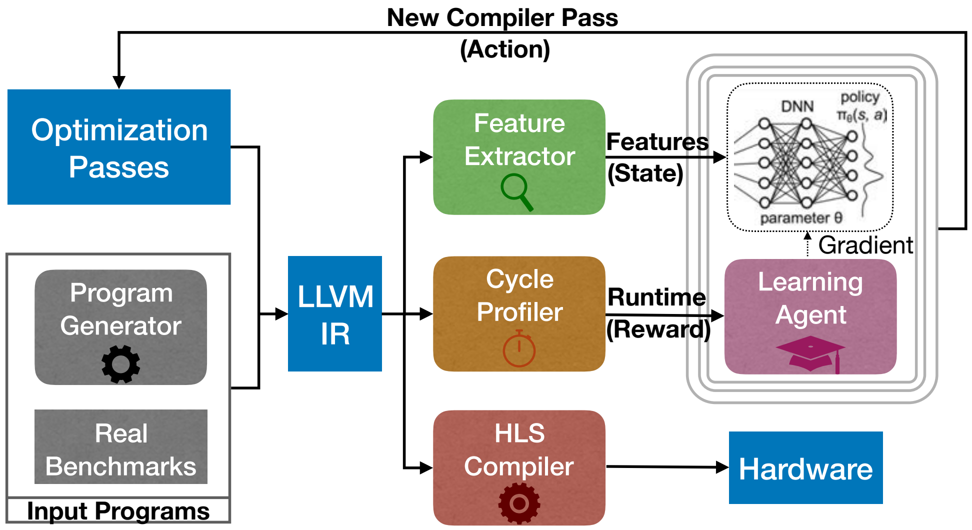


Figure 4.4: The block diagram of AutoPhase. The input programs are compiled to an LLVM IR using Clang/LLVM. The feature extractor is used to generate the input features (state) and the clock cycle profiler is used to generate the runtime improvement (reward) from the IR. The input features and runtime improvement are fed to the deep RL agent as input data to train on. The RL agent predicts the next best optimization passes to apply. After convergence, the HLS compiler is used to compile the LLVM IR to hardware RTL.

a program (or multiple programs) and intelligently explores the space of possible passes to figure out an optimal pass sequence to apply. Table 4.1 lists all the passes used in AutoPhase. The workflow of AutoPhase is illustrated in Figure 4.4.

HLS Compiler

AutoPhase takes a set of programs as input and compiles them to a hardware-independent intermediate representation (IR) using the Clang front-end of the LLVM compiler. Optimization and analysis passes act as transformations on the IR, taking a program as input and generating a new IR as output. The HLS tool LegUp is invoked after the compiler optimization as a back-end pass, which transforms LLVM IR into hardware modules.

Clock-cycle Profiler

Once the hardware RTL is generated, a hardware simulation could be run to gather the cycle count results of the synthesized circuit. This process is quite time-consuming, hindering RL and all other optimization approaches. Therefore, the cycle count is approximated using the profiler in LegUp [143], which leverages the software traces and runs 20× faster than hardware simulation. In LegUp, the frequency of the generated circuits is set as a compiler constraint that directs the HLS scheduling algorithm. In other words, the HLS tool will always try to generate hardware that can run at a certain frequency. In our experiment setting, without loss of generality, the target frequency of all generated hardware is set to 200MHz. Experiments on lower frequencies were also conducted; the improvements were similar but the cycle counts achieved by the different algorithms were better as more logic could be fitted in a single cycle.

IR Feature Extractor

Wang *et al.* [17] proposed to convert a program into an observation by extracting all its features. Similarly, in addition to utilizing the LegUp backend tools, analysis passes to extract 56 static features from the program were developed, such as the number of basic blocks, branches, and instructions of various types. These features were used as partially observable states for the RL learning, in the hope that the neural network can capture the correlation of certain combinations of these features and certain optimizations. Table 4.2 lists all the features used.

Random Program Generator

As a data-driven approach, deep RL generalizes better if the agent is trained on more programs. However, there are a limited number of open-source HLS examples online. Therefore, our training set was expanded by automatically generating synthetic HLS benchmarks. Standard C programs using CSmith [146] were generated first. CSmith is a random C program generator, originally designed to generate test cases for finding compiler bugs. Programs that take more than five minutes to run on CPU or fail the HLS compilation were filtered out.

Overall Flow of AutoPhase

The compilation utilities were integrated into a simulation environment in Python with APIs, similar to OpenAI Gym [118]. The overall flow works as follows:

1. The input program is compiled into LLVM IR using the Clang/LLVM.
2. The IR Feature Extractor is run to extract salient program features.
3. LegUp compiles the LLVM IR into hardware RTL.
4. The Clock-cycle Profiler estimates a clock-cycle count for the generated circuit.
5. The RL agent takes the program features or the histogram of previously applied passes and the improvement in clock-cycle count as input data to train on.
6. The RL agent predicts the next best optimization passes to apply.
7. New LLVM IR is generated after the new optimization sequence is applied.
8. The machine learning algorithm iterates through steps (2)–(7) until convergence.

Table 4.1: LLVM Transform Passes.

0	1	2	3	4	5		
-correlated-propagation	-scalarrepl	-lowerinvoke	-strip	-strip-nondebug	-sccp		
6	7	8	9	10	11		
-globalopt	-gvn	-jump-threading	-globaldce	-loop-unswitch	-scalarrepl-ssa		
12	13	14	15	16	17		
-loop-reduce	-break-crit-edges	-loop-deletion	-reassociate	-lcssa	-codegenprepare		
18	19	20	21	22	23		
-memcpyopt	-functionattrs	-loop-idiom	-lowerswitch	-constmerge	-loop-rotate		
24	25	26	27	28	29	30	
-partial-inliner	-inline	-early-cse	-indvars	-adce	-loop-simplify	-instcombine	
31	32	33	34	35	36	37	38
-simplifycfg	-dse	-loop-unroll	-lower-expect	-tailcallelim	-licm	-sink	-mem2reg
39	40	41	42	43	44	45	
-prune-eh	-functionattrs	-ipsccp	-deadargelim	-sroa	-loweratomic	-terminate	

Note that although AutoPhase uses the LLVM compiler and the passes as listed in Table 4.2, adding support for any compiler or optimization passes is very easy and straightforward. The action and state definitions must be specified again.

4.4 Correlation of Passes and Program Features

As is the case with many deep learning approaches, explainability is a major challenge when applying deep RL to phase-ordering optimization. To analyze and understand the correlation of passes and program features, random forests [147] are used to learn the importance of different features. Random forest is an ensemble of multiple decision trees. The prediction made by each tree can be explained by tracing the decisions made at each node and calculating the importance of different features for each of those decisions. This helps us to identify the effective features and passes and shows whether our algorithms learn informative patterns on data.

For each pass, two random forests are built to predict whether applying it would improve the circuit performance. The first forest takes the program features as inputs while the second takes a histogram of previously applied passes. To gather the training data for the forests, PPO is run with a high exploration parameter on 100 randomly generated programs to generate feature–action–reward tuples. The algorithm assigns higher importance to the input features that have greater effect on the final prediction.

Table 4.2: Program Features.

0	Number of BBs where total args for phi nodes >5	28	Number of And instructions
1	Number of BBs where total args for phi nodes is [1,5]	29	Number of BBs with instructions in [15,500]
2	Number of BBs with 1 predecessor	30	Number of BBs with less than 15 instructions
3	Number of BBs with 1 predecessor and 1 successor	31	Number of BitCast instructions
4	Number of BBs with 1 predecessor and 2 successors	32	Number of Br instructions
5	Number of BBs with 1 successor	33	Number of Call instructions
6	Number of BBs with 2 predecessors	34	Number of GetElementPtr instructions
7	Number of BBs with 2 predecessors and 1 successor	35	Number of ICmp instructions
8	Number of BBs with 2 predecessors and successors	36	Number of LShr instructions
9	Number of BBs with 2 successors	37	Number of Load instructions
10	Number of BBs with >2 predecessors	38	Number of Mul instructions
11	Number of BBs with Phi node # in range (0,3]	39	Number of Or instructions
12	Number of BBs with more than 3 Phi nodes	40	Number of PHI instructions
13	Number of BBs with no Phi nodes	41	Number of Ret instructions
14	Number of Phi-nodes at beginning of BB	42	Number of SExt instructions
15	Number of branches	43	Number of Select instructions
16	Number of calls that return an int	44	Number of Shl instructions
17	Number of critical edges	45	Number of Store instructions
18	Number of edges	46	Number of Sub instructions
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc instructions
20	Number of occurrences of 64-bit integer constants	48	Number of Xor instructions
21	Number of occurrences of constant 0	49	Number of ZExt instructions
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of binary operations with a constant	52	Number of memory instructions
25	Number of AShr instructions	53	Number of non-external functions
26	Number of Add instructions	54	Total arguments to Phi nodes
27	Number of Alloca instructions	55	Number of unary operations

Importance of Program Features

The heat map in Figure 4.5 shows the importance of different features in the decision as to whether a pass should be applied. The higher the value is, the more important the feature is (the sum of the values in each row is one). The random forest is trained with 150,000 samples generated from the random programs. The index mapping of features and passes can be found in Tables 4.1 and 4.2. For example, the yellow pixel corresponding to feature index 17 and pass index 23 reflects that *number-of-critical-edges* greatly affects the decision as to

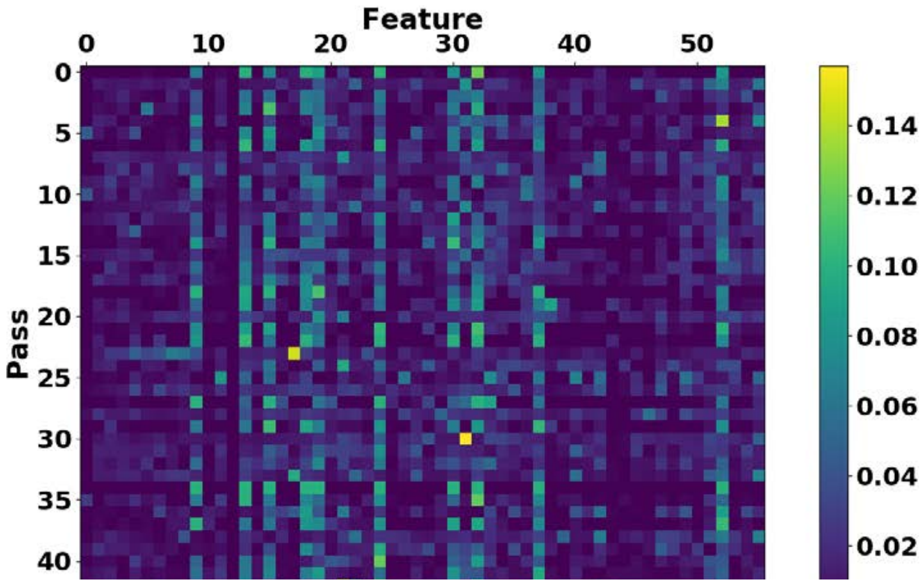


Figure 4.5: Heat map illustrating the importance of feature and pass indices. The higher the value is, the more important the feature is.

whether to apply *-loop-rotate*. A critical edge in the control flow graph is an edge that is neither the only edge leaving its source block nor the only edge entering its destination block. The critical edges often appear in a loop as a back edge, so the number of critical edges might roughly represent the number of loops in a program. The transform pass *-loop-rotate* detects a loop and transforms a while loop to a do-while loop to eliminate one branch instruction in the loop body. Applying the pass results in better circuit performance as it reduces the number of FSM states in a loop.

Other expected behaviors can also be observed in this figure. These include, for example, the correlation between *number of branches* and the transform passes *-loop-simplify*, *-tailcallelim*, and *-lowerswitch* (*-tailcallelim* transforms calls of the current function, *i.e.*, self recursion, followed by a return instruction with a branch to the entry of the function, creating a loop, and *-lowerswitch* rewrites switch instructions with a sequence of branches). Other interesting behaviors are also captured, for example, in the correlation between *binary operations with a constant operand* and *-functionattrs*, which marks different operands of a function as read-only (constant). Some correlations are harder to explain, for example, the correlation between *number of BitCast instructions* and *-instcombine*, which combines instructions into fewer, simpler instructions. This is actually a result of *-instcombine* reducing the loads and stores that call bitcast instructions for casting pointer types. Another example of an interesting behavior is observed in the correlation between *number of memory instructions* and *-sink*, where *-sink* basically moves memory instructions into successor blocks and delays the execution of memory until needed. Clearly, the decision whether to apply *-sink* should be dependent on whether there is any memory instruction in the program. Our last example

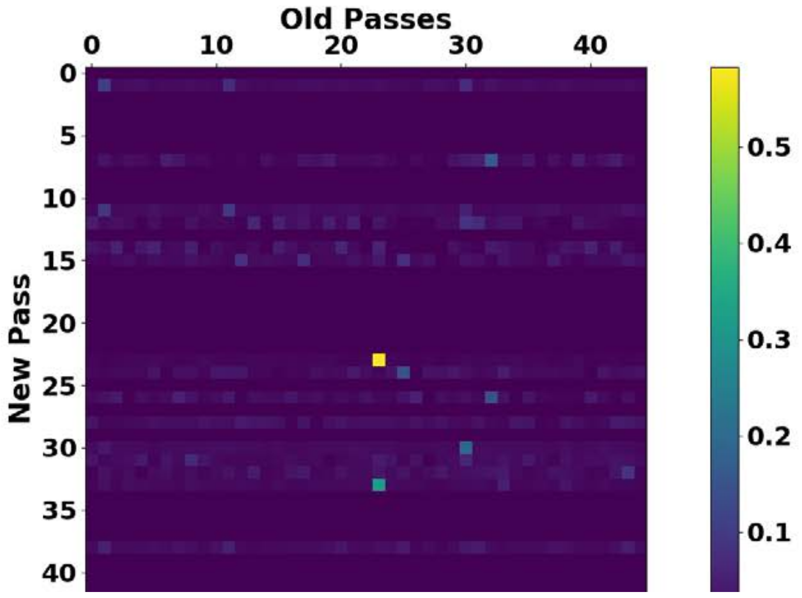


Figure 4.6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply. The higher the value is, the more important having the old pass is.

to show is *number of occurrences of constant 0* and *-deadargelim*, where *-deadargelim* helped eliminate dead/unused constant zero arguments.

Overall, all the passes are correlated to some features and are able to affect the final circuit performance. We also observed that multiple features are not effective at directing decisions and training with them could increase the variance in lower prediction accuracy of our results. For example, the total number of instructions did not give a direct indication of whether applying it would be helpful. This is because more instructions might sometimes improve the performance (for example, due to loop unrolling), and likewise for eliminating unnecessary code. In addition, the importance of features varies among different benchmarks depending on the tasks they perform.

Importance of Previously Applied Passes

Figure 4.6 illustrates the impact of previously applied passes on the new pass to apply. The higher the value is, the more important having the old pass is. This figure shows that passes *-scalarrepl, -gvn, -scalarrepl-ssa, -loop-reduce, -loop-deletion, -reassociate, -loop-rotate, -partial-inliner, -early-cse, -adce, -instcombine, -simplifycfg, -dse, -loop-unroll, -mem2reg, and -sroa* had greater impact on the performance than the rest of the passes regardless of their order in the trajectory. Point (23,23) has the highest value, which implies that pass *-loop-rotate* is very helpful and should be included if it was not applied before. Examining thousands of programs shows that *-loop-rotate* indeed reduces the cycle count significantly. Interestingly, applying this pass twice is not harmful if applied consecutively. However, if other passes are applied between them, performance might be reduced dramatically. Another interesting behavior captured by the heat map is that it was much more useful to apply pass 33 (*-loop-unroll*) after (not necessarily consecutive) pass 23 (*-loop-rotate*) than it was to apply these two passes in the opposite order.

4.5 Problem Formulation

The RL Environment Definition

Assume the optimal number of passes to apply is N and there are K transform passes to select from. Then, the search space $\mathcal{S}$ for the phase-ordering problem is $[0, K^N)$. Given M program features and the history of already applied passes, the goal of deep RL is to learn the next best optimization pass a to apply that minimizes the long-term cycle count of the generated hardware circuit. Note that the optimization state s is partially observable in this case as the M program features cannot fully capture all the properties of a program.

Action Space – the action space $\mathcal{A}$ is defined as $\{a \in \mathbb{Z} : a \in [0, K)\}$ where K is the number of transform passes.

Observation Space – two types of input features were considered in our evaluation: ① **program features $\mathbf{o_f}$** $\in \mathbb{Z}^M$ listed in Table 4.2 and ② **action history**, which is a histogram of previously applied passes $\mathbf{o_a} \in \mathbb{Z}^K$. After each RL step where the pass i is applied, the feature extractor in our environment is called to return new $\mathbf{o_f}$ and update the action histogram element o_{a_i} to $o_{a_i} + 1$.

Reward – the cycle count of the generated circuit is reported by the clock-cycle profiler at each RL iteration. Our reward is defined as $R = c_{prev} - c_{cur}$, where c_{prev} and c_{cur} represent, respectively, the previous and the current cycle count of the generated circuit. Different rewards can be defined for different objectives. For example, the reward could be defined to be proportional to the decrease in the area, resulting in the RL agent optimizing for the area. It is also possible to co-optimize multiple objectives (e.g., area, execution time, power, etc.) by defining a combination of different objectives.

Applying Multiple Passes per Action

An alternative to the action formulation above is to evaluate a complete sequence of passes with length N instead of a single action a at each RL iteration. Upon the start of training a new episode, the RL agent resets all pass indices $\mathbf{p} \in \mathbb{Z}^N$ to the index value $\frac{K}{2}$. For pass p_i at index i, the next action to take is either to switch to a new pass or not. By allowing positive and negative index updates for each p, the number of steps required to traverse all possible pass indices is reduced. The sub-action space a_i for each pass is thus defined as $[-1, 0, 1]$. The total action space $\mathcal{A}$ is defined as $[-1, 0, 1]^N$. At each step, the RL agent predicts the updates $[a_1, a_2, ..., a_N]$ to N passes, and the current optimization sequence $[p_1, p_2, ..., p_N]$ is updated to $[p_1 + a_1, p_2 + a_2, ..., p_N + a_N]$.

Normalization Techniques

In order for the trained RL agent to work on new programs, the program features and rewards must be properly normalized so that they represent a meaningful state for different programs. In this work, we experimented with two techniques: ① taking the logarithm of program features or rewards and, ② normalizing to a parameter from the original input program that roughly depicts the problem size. For technique ①, note that taking the logarithm of the program features not only reduces their magnitude but also correlates them in a different manner in the neural network. Since $w_1 \log(o_{f_1}) + w_2 \log(o_{f_2}) = log(o_{f_1}^{w_1} o_{f_2}^{w_2})$, the neural network learns to correlate the products of features instead of a linear combination of them. For technique ②, the program features are normalized to the number of instructions in the input program ($\mathbf{o_{f\_norm}} = \frac{\mathbf{o_f}}{o_{f_{51}}}$), which is feature #51 in Table 4.2.

4.6 Evaluation

To run the deep RL algorithms, we used RLlib [148], an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib is built on top of Ray [149], a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. The framework runs on a four-core Intel i7-4765T CPUwith a Tesla K20c GPUfor training and inference.

The frequency constraint in HLS is set to 200MHz, and the number of clock cycles reported by the HLS profiler is used as the circuit performance metric. In [143], results showed a one-to-one correspondence between the clock cycle count and the actual hardware execution time under certain frequency constraints. Therefore, better clock cycle count will lead to better hardware performance.

Performance

We evaluated the effectiveness of various deep RL algorithms in tackling the phase-ordering problem on nine real HLS benchmarks and compared them to state-of-the-art approaches, including random search, Greedy Algorithms [143], OpenTuner [4], and Genetic Algorithms [150]. The metrics for comparison were the final HLS circuit performance and the sample efficiency. The benchmarks were adapted from CHStone [145] and LegUp examples. They are: *adpcm*, *aes*, *blowfish*, *dhrystone*, *gsm*, *matmul*, *mpeg2*, *qsort*, and *sha*. For this evaluation, the input features/rewards were not normalized, the pass length was set to 45, and each algorithm was run on a per-program basis. Table 4.3 lists the action and observation spaces used in all the deep RL algorithms.

The bar chart in Figure 4.7 shows the percentage improvement of the circuit performance compared to -O3 results on the nine real benchmarks from CHStone. The dots on the blue line in Figure 4.7 show the number of samples for each program, which is the number of times the algorithm calls the simulator to gather the cycle count. `-O0` and `-O3` are the default compiler optimization levels. `RL-PPO1` is a PPO explorer where all the rewards are set to 0 to

Table 4.3: The observation and action spaces used in the different deep RL algorithms.

	RL-PPO1	**RL-PPO2**	**RL-PPO3**	**RL-A3C**	**RL-ES**
Deep RL Algorithm	PPO	PPO	PPO	A3C	ES
Observation Space	Program Features	Action History	Action History + Program Features	Program Features	Program Features
Action Space	Single-Action	Single-Action	Multiple-Action	Single-Action	Single-Action

test if the rewards are meaningful. `RL-PPO2` is the PPO agent that learns the next pass based on a histogram of applied passes. `RL-A3C` is the A3C agent that learns based on the program features. `Greedy` performs the greedy algorithm, which always inserts the pass that achieves the highest speedup at the best position (out of all possible positions it can be inserted into) in the current sequence. `RL-PPO3` uses a PPO agent and the program features but with the action space described in Section 4.5. `OpenTuner` runs an ensemble of six algorithms, which includes two families of algorithms: particle swarm optimization [151] and GA, each with three different crossover settings. `RL-ES` is similar to the A3C agent that learns using the program features, but updates the policy network using the evolution strategy instead of backpropagation. `Genetic-DEAP` [150] is a genetic algorithm implementation. `random` randomly generates a sequence of 45 passes at once instead of sampling them one-by-one.

`Greedy` results show that always inserting that pass in the best position (*i.e.*, the one with the highest reward) in the current sequence leads to sub-optimal circuit performance. `RL-PPO2` achieves better performance than `RL-PPO1`, which shows that the deep RL captures useful information during training. Using the histogram of applied passes results in better sample efficiency, but using the program features with more samples results in a slightly higher speedup. `RL-PPO2`, for example, achieves 50× better sample efficiency than `OpenTuner`, with only 4% drop in speedup. Using ES to update the policy is supposed to be more sample efficient for problems with sparse rewards like ours; however, our experiments did not demonstrate this benefit. Furthermore, `RL-PPO3` with multiple action updates achieves a better speedup than the other deep RL algorithms. This is due in part to its ability to apply more passes per compilation, which makes it run faster and more efficiently. On the other hand, the other deep RL algorithms apply a single pass at a time.

Generalization

With deep RL, the search should benefit from prior knowledge learned from other programs. This knowledge should be transferable from one program to another. For example, as discussed in Section 4.4, applying pass *-loop-rotate* is always beneficial, and *-loop-unroll* should be applied after *-loop-rotate*. Note that the black-box search algorithms, such as OpenTuner, GA, and greedy algorithms, cannot generalize. For these algorithms, a new search with many compilations must be run for every new program, as they do not learn any patterns from the programs to direct the search and can be viewed as performing smart random search.

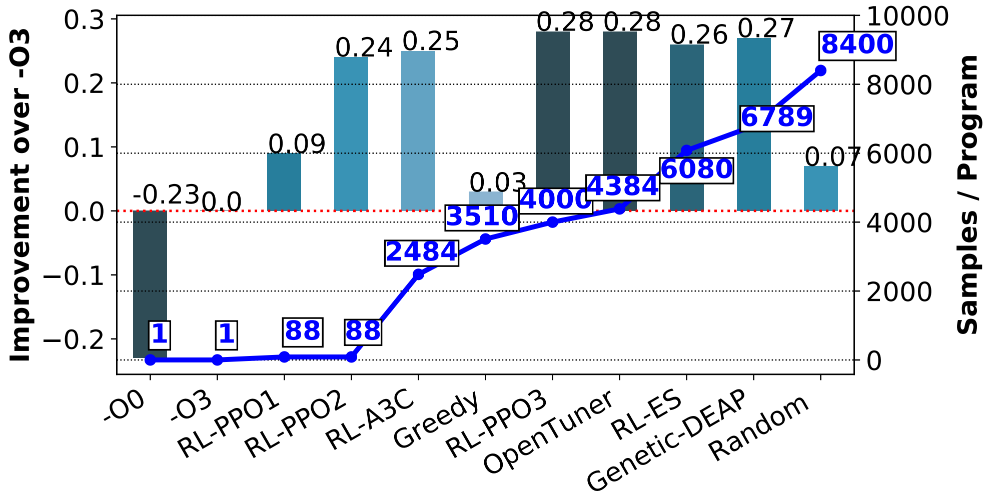


Figure 4.7: Circuit speedup and sample size comparison.

To evaluate how generalizable deep RL might be with different programs and whether any prior knowledge could be useful, a deep RL network was trained on 100 randomly generated programs using PPO. Random programs were used for transfer learning because not enough benchmarks were available and because it is the worst-case scenario, *i.e.*, these programs are very different from the programs that were used for inference. The performances improves further when the network is trained on programs that are similar to the ones inferenced. A network with 256 $\times$ 256 fully connected layers was trained, the histogram of previously applied passes concatenated to the program features was used as the observation, and passes were used as actions.

As described in Section 4.5, two normalization techniques for the program features were tested: ① taking the logarithm of all the program features, and ② normalizing the program features to the number of instructions in the program. In each pass sequence, the intermediate reward was defined as the logarithm of the decrease in cycle count after applying each pass. The logarithm was chosen so that the RL agent will not give much larger weights to big rewards from programs with longer execution time. Three approaches were evaluated: `filtered-norm1`, which uses the filtered program features and passes from Section 4.4 with normalization technique ①; `original-norm2`, which uses all the program features and passes with normalization technique ②; and `filtered-norm2`, which uses the filtered program features and passes from Section 4.4 with normalization technique ②. Filtering the features and passes might not be ideal, especially when different programs have different feature characteristics and the impact of a particular pass is not the same. However, reducing the number of features and passes helps to reduce variance among all programs and significantly narrows the search space.

Figure 4.8 shows the episode reward mean as a function of the step for the three approaches. We see that `filtered-norm2` and `filtered-norm1` converge much faster and achieve a higher

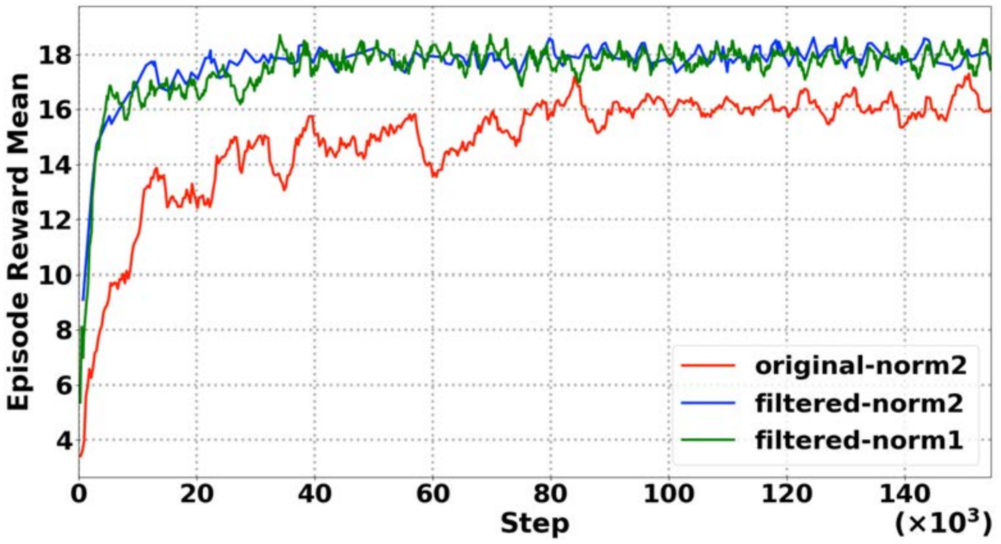


Figure 4.8: Episode reward mean as a function of step for the original approach where all the program features and passes are used, and for the filtered approach where the passes and features (with different normalization techniques) are filtered. Higher values indicate faster circuit speed.

episode reward mean than `original-norm2`, which uses all the features and passes. At roughly 8,000 steps, `filtered-norm2` and `filter-norm1` already achieve a very high episode reward mean, with minor improvements in later steps. Furthermore, the episode reward mean of the filtered approaches is still higher than that of `original-norm2` even when allowed to train for 20 times more steps (*i.e.*, 160,000 steps). This indicates that filtering the features and passes significantly improved the learning process. All three approaches learned to always apply pass *-loop-rotate*, and *-loop-unroll* after *-loop-rotate*. Another useful pass that the three approaches learned to apply is *-loop-simplify*, which performs several transformations to transform natural loops into a simpler form that enables subsequent analyses and transformations.

To demonstrate the generalization abilities of `filtered-norm1` and `filtered-norm2` and their performance advantages, we compared them to the black box algorithms. We used 100 randomly generated programs as the training set and nine real benchmarks from CHStone as the testing set. The results of state-of-the-art black-box algorithms were extracted by searching for the best pass sequences that achieved the lowest aggregated hardware cycle counts for the 100 random programs and then directly applying them to the nine test set programs. The bar chart in Figure 4.9 shows the percentage improvement of the circuit performance compared to -O3 on the nine real benchmarks. The dots on the blue line show the total number of samples each inference takes for one new program.

This evaluation shows that the deep RL-based inference achieves higher speedup than the predetermined sequences produced by the state-of-the-art black-box algorithms for new

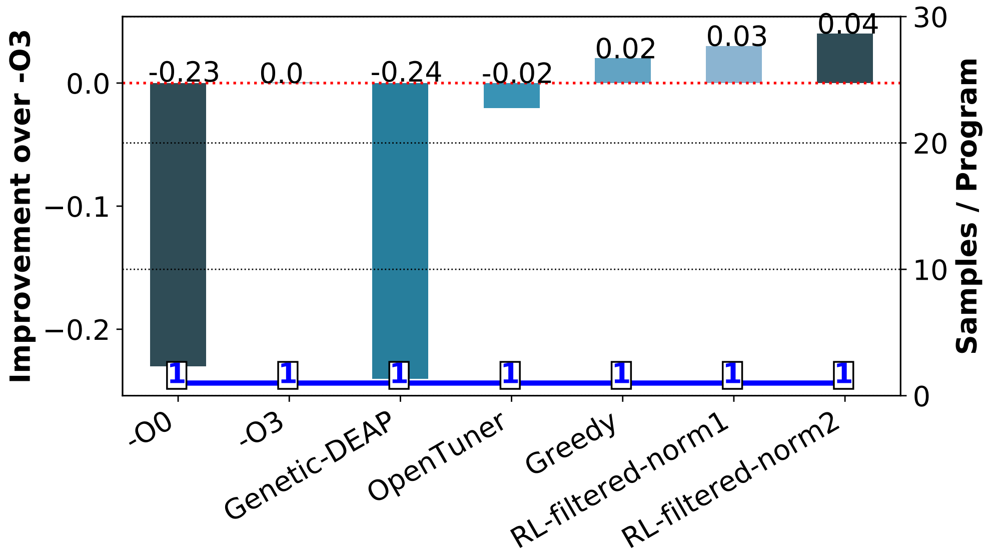


Figure 4.9: Circuit speedup and sample size comparison for deep RL generalization.

programs. The predetermined sequences that are overfitted to the random programs can cause poor performance in unseen programs (*e.g.,* -24% for `Genetic-DEAP`). The evaluation also shows that normalization technique ② works better than normalization technique① for deep RL generalization (4% vs 3% speedup). This indicates that normalizing the different instructions to the total number of instructions in technique ② (*i.e.*, transforming the features to reflect the distribution of the different instructions) allows more universal characteristics to be represented across different programs, while taking the log in technique ① only suppresses the value ranges of different program features. Furthermore, when another set of 12,874 randomly generated programs is used as the testing set with `filtered-norm2`, the speedup is 6% better than -O3.

4.7 Conclusion and Future Directions

Our proposed deep RL approach was shown to improve the performance of HLS designs by optimizing the order in which the compiler applies optimization phases. Random forests were used to analyze the relationship between program features and optimization passes. This relationship was leveraged to reduce the search space by identifying the most likely optimization phases to improve the performance, given the program features. Our RL based approach achieves 28% better execution time than compiling with the -O3 flag after training for a few minutes, and a 24% improvement after training for less than a minute. Furthermore, unlike prior work, the proposed solution shows potential to generalize to a variety of programs.

The static captured features in AutoPhase do not fully capture the entire characteristics of the program or the changes in the IR. Thus, the performance of AutoPhase can be significantly improved with better features, which can include loop polyhedral analysis, graph representation of the code, and abstract syntax tree. For better generalization, AutoPhase should be trained on a wide spectrum of programs, potentially with a more sophisticated deep neural network model. The underlying hardware should be featurized as an input for AutoPhase for it to generalize to different hardware targets. Often the passes can allow a modifiable value (pragma) to be passed in as an optimization (in AutoPhase this value was set to the default). For example, when passing the vectorization or interleaving pass, vectorization and interleaving factors can be passed to the compiler. This improvement is addressed in our NeuroVectorizer system, presented in the next chapter. While here we showed how to apply deep RL to HLS, a similar approach can be successfully applied to software compilation and optimization.

Chapter 5

NeuroVectorizer

5.1 Introduction

Vectorization is another mission-critical compiler optimization challenge, crucial for enhancing the performance of computer-intensive workloads. Modern computers typically have vector instructions that perform multiple basic operations simultaneously, such as Intel Advanced Vector Extensions (AVX) [152]. *Vectorization* is the process of converting a computer program from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation, which performs a single operation on multiple data (SIMD) items at once.

Loops are among the most commonly vectorized parts of the code. Loop vectorization is done by defining the vectorization factor (VF) and the interleaving factor (IF) [153]. VF determines how many instructions to pack together from different iterations of the loop. IF determines the stride of the memory accesses of the packed instructions. IF allows vectorization to be performed on non-consecutive addresses, generally referred to as non-unit stride accesses.

In most C and C++ compilers, intrinsic pragmas or compiler passes can be used to manually vectorize loops by setting the VF and IF. However, manual vectorization is labor intensive, error-prone, and results in code that is difficult to maintain and port. Several solutions for automatic vectorization and loop optimization have been proposed. The current vectorizer used in LLVM, and its proposed improvements [14, 154], rely on linear and constant-cost models to predict the vectorization factors. Unfortunately, these cost models do not consider the computation graph and focus on estimating the cost of different instructions with predefined heuristics. Another common approach is Polly [15]. Polly uses loop polyhedral analysis, which relies on an abstract mathematical representation, namely equations and matrices, to represent loops as polytopes. The polytope representation simplifies the implementation of loop optimizations, though to date the main optimizations in Polly are tiling and loop fusion to improve data locality.

Machine learning is yet another recent approach that has been proposed for automatic vectorization [49]. While this approach improves the cost models implemented by existing compilers, it still requires hand-engineered heuristics to extract features from the assembly code, such as arithmetic intensity. Unfortunately, these features are typically not sufficient to fully capture the code functionality. To overcome this challenge, an end-to-end solution that relies on deep supervised learning was proposed in [51]. However, supervised learning methods require labels to train. These labels are not always available and it can be time-consuming to find them. Furthermore, optimizing for multiple objectives with large search spaces can be challenging for supervised learning methods. Finally, deep RL differs from other machine learning methods in that its inherent trade-off between self-exploration and exploitation [21] can be leveraged.

In this chapter we present a framework for automatic vectorization using deep RL. A human vectorization expert can determine the optimal vectorization factors, *i.e.*, VF and IF, for a specific hardware architecture by examining the computation graph, functionality, operations, and loop bodies in the text code. In our framework we use a code embedding generator that reads the text similarly to a human expert, "understands" it, and then generates an embedding that represents it. We use the generated embedding as an input to another neural network that can learn a mapping from this embedding to optimal vectorization factors similar to those learned by a human expert. This approach efficiently addresses the vectorization challenge end-to-end: from code to optimal factors, enabling the co-optimization of multiple objectives while preserving code correctness.

This chapter makes the following contributions:

- A comprehensive data set of more than 10,000 synthetic loop examples.
- An end-to-end deep reinforcement learning (RL) [21] based auto-vectorization method.
- NeuroVectorizer [18]: an extensible, open-source[1] framework that integrates learning code embedding with multiple machine learning methods to make vectorization predictions on loops. We explore using random search, supervised learning methods, *i.e.*, nearest-neighbor search (NNS) [155], decision trees [156], supervised fully connected neural network (FCNNs), and contextual bandits based on deep RL.
- Rigorous evaluations across different learning hyperparameters and benchmark suites to show the effectiveness of our approaches versus the currently used cost model as well as its effectiveness versus Polly. Our results show $1.29\times-4.73\times$ average performance speedup and only 3% worse speedup than the brute-force solution. Furthermore, NeuroVectorizer shows the ability to generalize to unseen programs.

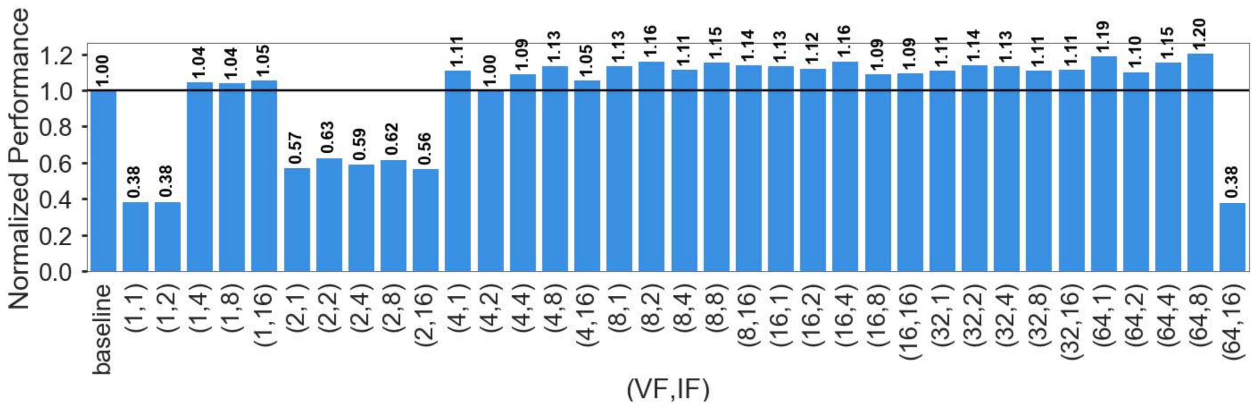


Figure 5.1: Performance of the dot product kernel for different VFs and IFs, normalized to the baseline cost model implemented in LLVM. The best VF and IF corresponding to the baseline cost model are $(VF = 4, IF = 2)$.

5.2 Motivation

Vectorization Characterization

To better understand the vectorization challenges that motivate this work, consider a simple vector dot product kernel function:

```
int vec[512] __attribute__((aligned(16)));
__attribute__((noinline))
int dot_product () {
    int sum = 0;
    for(int i = 0; i<512; i++){
        sum += vec[i]*vec[i];
    }
    return sum;
}
```

To eliminate noise and variance in results, this kernel is run one million times and the average execution time is calculated. The kernel is run on 16 GB 2133 MHz LPDDR3 memory and 2.7 GHz (up to 4.5 GHz) Intel 4-Core i7-8559U [157], which supports AVX. Figure 5.1 shows the performance of this kernel after a brute-force search for different VFs and IFs normalized to the baseline cost model implemented in LLVM. The best VF and IF corresponding to the baseline cost model are $(VF = 4, IF = 2)$. While the baseline improved the performance by 2.6× when compared to the unvectorized code $(VF = 1, IF = 1)$, 26 out of 35 possible factors improve over the baseline. This improvement is maximized by $(VF = 64, IF = 8)$, which performs up to 20% better than the baseline.

[1]https://github.com/intel/neuro-vectorizer.

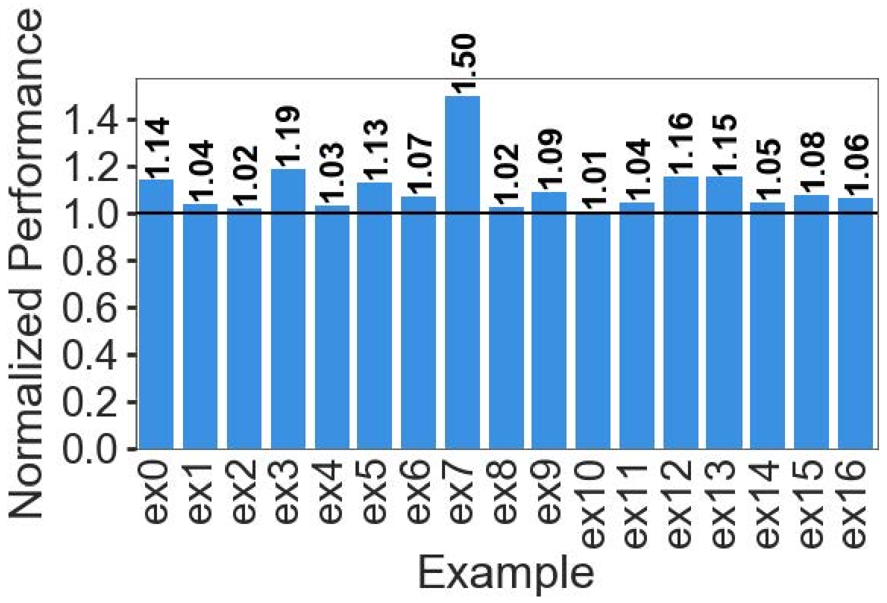


Figure 5.2: Performance of brute-force search of LLVM's vectorizer test suite, normalized to the baseline cost model implemented in LLVM.

To show how the current baseline cost model can be further improved, we ran a brute-force search for all possible VFs and IFs on the vectorization test suite used in the LLVM base code[2], which evaluates the cost model of the baseline vectorizer in LLVM. The performance of the optimal vectorization normalized to the baseline is illustrated in Figure 5.2. In all the tests, the optimal vectorization performed better than the baseline. and for more complicated tests, the performance gap increased by up to 1.5×. There results provide additional motivation for using deep RL to automatically vectorize loops. In what follows, we show how deep RL can learn with fewer samples than supervised learning methods, without the need for brute-force search, which can be impractical for a larger number of samples. We further show how deep RL — unlike supervised learning — can co-optimize for multiple objectives, such as code size, compilation time, and execution time.

5.3 The Proposed Framework for Automatic Vectorization

The proposed framework for automatic vectorization with deep RL and its components are illustrated in Figure 5.3. The directory of code files is fed to the framework as text code. This code is fed to an automatic loop extractor. The extractor finds and outputs all the

[2]The test suite is available on:
https://github.com/llvm/llvm-test-suite/tree/master/SingleSource/UnitTests/Vectorizer.

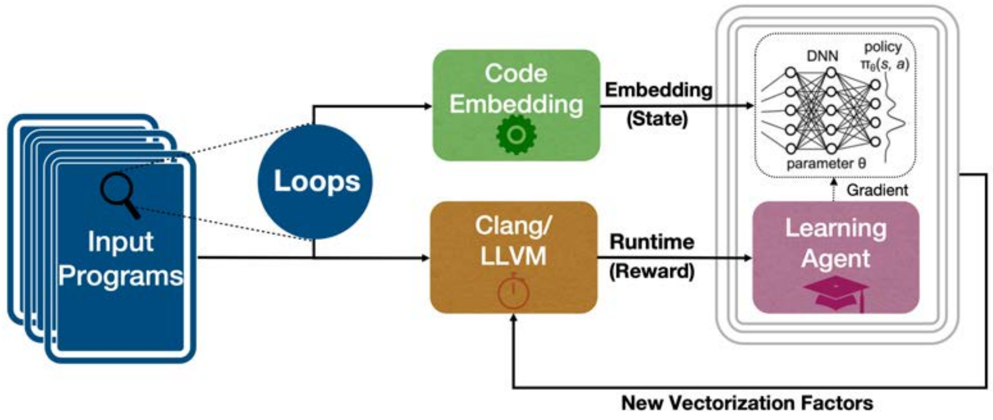
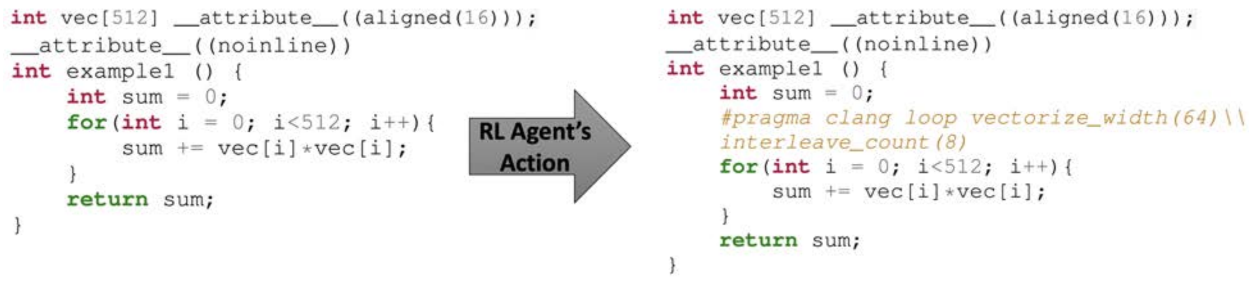


Figure 5.3: The proposed framework for automatic vectorization with deep RL. The programs are read to extract the loops. The loop texts are fed to the code embedding generator to generate an embedding. The embedding is fed to the RL agent. The RL agent learns a policy that maps this embedding to optimal vectorization factors by injecting compiler pragmas and compiling the programs with Clang/LLVM to gather the rewards: the execution time improvements.



```
int vec[512] __attribute__((aligned(16)));
__attribute__((noinline))
int example1 () {
     int sum = 0;
     for(int i = 0; i<512; i++){
          sum += vec[i]*vec[i];
     }
     return sum;
}
```

RL Agent's Action

```
int vec[512] __attribute__((aligned(16)));
__attribute__((noinline))
int example1 () {
     int sum = 0;
     #pragma clang loop vectorize_width(64)\\
     interleave_count(8)
     for(int i = 0; i<512; i++){
          sum += vec[i]*vec[i];
     }
     return sum;
}
```

Figure 5.4: An example of the automatically injected VF and IF pragmas by the RL agent.

loops and their contexts in all the source codes. These outputs are fed to a code embedding generator to learn and generate an embedding. The latter is fed to the deep RL agent to predict the vectorization factors. The agent automatically injects vectorization pragmas as shown in Figure 5.4. The agent then compiles the program with clang/LLVM to gather the execution time improvements, which are used as rewards to the RL agent. Once the model is trained it can be plugged in as-is for inference without further retraining[3]. Note that our framework cannot introduce new errors in the compiled code. Our framework injects pragmas only. These pragmas are used as hints to make vectorization decisions on the loops. However, sometimes the compiler can decide not to consider these pragmas if it is not feasible to do so.

[3]It can still be beneficial to keep online training activated so that when completely new loops are observed, the agent can learn how to optimize them too.

For example, predicates and memory dependency can make it difficult to find the optimal VF and IF. In that case, if the agent accidentally injected bad pragmas, the compiler will ignore them.

It is also possible to vectorize from the command line by giving the passes *-force-vector-width=VF* and *-force-vector-interleave=IF*. However, this option was not used as it restricts us to a single VF and IF pair for the entire code, which is far from optimal. Furthermore, for nested loops, the pragma is injected to the innermost loop. Next, we discuss the details of each component in the proposed framework.

Code Embedding

The ultimate goal of the code embedding generator is to learn a function that maps the input loop codes to a point in a latent multidimensional space where similar loop codes are mapped to points close to each other in that space. This can allow the RL agent to make similar vectorization decisions on similar codes using the learned embedding. There are multiple ways to generate/learn an embedding for the input code. One is to use Polly's mathematical representation of loops as an embedding. We see this as a potential future direction for this work. It is also possible to use a neural network model pretrained with labels that describe the functionality, *e.g.*, matrix multiplications, dot product, convolution, etc.

In this work we use code2vec [158]. Code2vec is a neural network model that relies on natural language processing [159] and attention [160] to represent snippets of code as continuously distributed vectors. Code2vec represents a code snippet as a single fixed-length code vector, which can be used to predict the semantic properties of the snippet. This vector is composed of 340 features that embed the program code based on the mapping learned by the code2vec neural network. This vector captures many characteristics of the code, such as semantic similarities, combinations, and analogies. The code is first decomposed to a collection of paths in its abstract syntax tree. Then, the network simultaneously learns the atomic representation of each path while learning how to aggregate a set of them.

The RL Environment Definition

To learn a good policy, it is necessary to appropriately define actions, rewards, and states. We define the agent's reward as follows:

$$reward = (t_{baseline} - t_{RL})/t_{baseline}, \tag{5.1}$$

where $t_{baseline}$ is the execution time when compiled with the currently implemented baseline cost model in LLVM and t_{RL} is the execution time when compiled with the pragmas injected by the RL agent. We normalize the execution time by $t_{baseline}$ so that our reward metric is robust to the variations in the programs' execution times. We also use $t_{baseline}$ as a bias in our reward so that a positive reward means the current configuration improves over the baseline. This also reduces the variance in the learned policy.

An action picks the VF and the IF, respectively, from the following values:

$$\begin{aligned} VF &\in [2^0, 2^1, 2^2, ..., \text{MAX\_VF}], \\ IF &\in [2^0, 2^1, 2^2, ..., \text{MAX\_IF}], \end{aligned} \tag{5.2}$$

where MAX_VF and MAX_IF are, respectively, the maximum VF and IF supported by the underlying architecture. Note that the actions for VF and IF can be defined to have values that are not powers of two. Here they were defined as powers of two only because this is what LLVM currently supports. Initially, we trained two agents, one that predicts VF and another that predicts IF independently. However, as shown in our experiment, performance improve when these two agents are combined into one agent with a single neural network that predicts the VF and IF simultaneously. This also aligns with the fact that IF and VF are directly correlated, and in the LLVM compiler code they are defined as functions of one another.

The states of the RL agent were defined as the vector output embedding from the code embedding generator. For the inputs of the code embedding generator, we experimented with different snippets of the loop bodies and observed that for nested loops, feeding the loop body of the outermost loop, which also includes the bodies of the inner loops, yielded better performance than feeding the body of the innermost loop only. This is mainly because the entire loop nest better captures the functionally of the code, and reveals the access patterns and strides.

Dataset Description

Neural networks require many samples for training. We first tried to train our model with long-running benchmarks that include code that is not restricted to loops only. Training was time-consuming because every time a pragma is injected for a loop, the whole program has to be recompiled and executed. Even if we had the resources to overcome the challenges of long execution time, the number of open-source benchmarks available for training is very small [50].

To speed up the training and make it more efficient, we built a dataset that includes loops only. We built generators that generate more than 10,000 synthetic loop examples automatically from the LLVM vectorization test suite. For example, some new tests are created by changing the names of the parameters, which was crucial for reducing noise in the code embedding generator, as the names of the parameters might often bias the embedding. Other examples included the stride, the number of iterations, the functionality, the instructions, and the number of nested loops. Below are some of the loop examples in the dataset and the (commented) pragma line that the RL agent will inject:

```
/* Example #1 */
//#pragma clang loop vectorize_width(VF) interleave_count(IF)
for (i = 0; i < N-1; i+=2) {
    assign1[i] = (int) short_a[i];
```

```
     assign1[i+1] = (int) short_a[i+1];
     assign2[i] = (int) short_b[i];
     assign2[i+1] = (int) short_b[i+1];
     assign3[i] = (int) short_c[i];
     assign3[i+1] = (int) short_c[i+1];
}
/* Example #2 */
for (i=0; i<M; i++) {
//#pragma clang loop vectorize_width(VF) interleave_count(IF)
    for (j=0; j<N; j++) {
       G[i][j] = x;
     }
}
/* Example #3 */
//#pragma clang loop vectorize_width(VF) interleave_count(IF)
for (i=0; i<N*2; i++){
    int j = a[i];
    b[i] = (j > MAX ? MAX : 0);
}
/* Example #4 */
for (i = 0; i < M; i++){
    for (j = 0; j < L; j++){
        float sum = 0;
//#pragma clang loop vectorize_width(VF) interleave_count(IF)
        for (k = 0; k < N; k++) {
            sum += alpha*A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
/* Example #5 */
//#pragma clang loop vectorize_width(VF) interleave_count(IF)
for (i = 0; i < N/2-1; i++){
    a[i] = b[2*i+1] * c[2*i+1] - b[2*i] * c[2*i];
    d[i] = b[2*i] * c[2*i+1] + b[2*i+1] * c[2*i];
}
```

Figure 5.5 shows the distribution of optimal vectorization factors when running a brute-force search with MAX_VF = 16 and MAX_IF = 8 on the dataset. While these loops do not represent all the existing loops, the results show that different loops have different optimal VF and IF. All combinations of VF and IF should be considered, if optimal performance is to be guaranteed. Interestingly, the factors with the highest percentage of programs are $(VF = 4, IF = 2)$. In our experiments, these factors were the default values the baseline cost model also outputted.

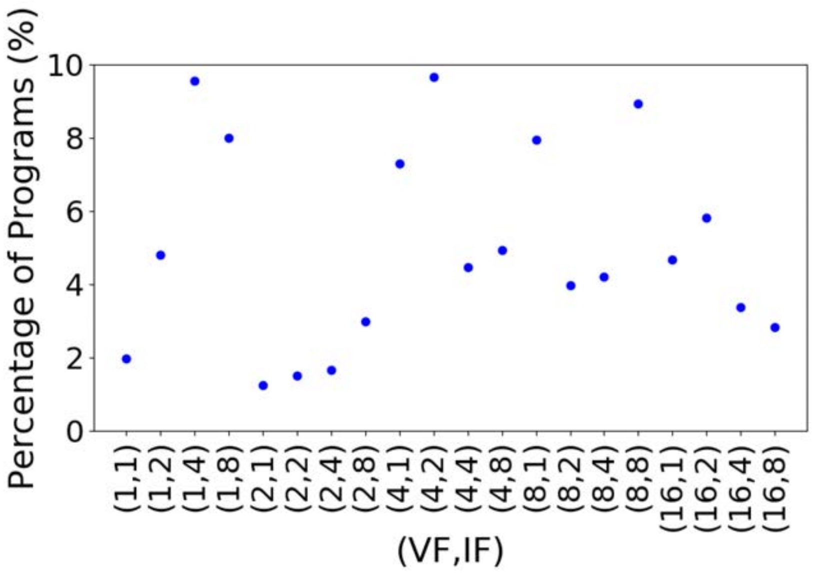


Figure 5.5: The distribution of optimal VF and IF with brute-force search for different programs in the dataset.

Handling Long Compilation Time

During training, some of the programs took a long time to compile, mainly when the agent was trying to vectorize more than is plausible given the available resources. To overcome this, the compilation time was limited to ten times the time it takes to compile a program with the baseline cost model. If the program took longer than that to compile, we gave a penalty reward of -9 (equivalent to assuming it takes ten times the execution time of the baseline) so that the agent will learn not to overestimate the vectorization. Moreover, the programs that took a relatively long time to compile did not, in the end, yield better results. Nonetheless, in contexts where compile time is important to the user, our reward definition can also incorporate it, by making the reward proportional to the decrease in compilation time or penalizing for longer compilation times. The reward can also be defined as a combination of the compilation time, execution time, generated assembly code size, *etc*, allowing for the simultaneous optimization of multiple objectives.

5.4 Evaluation

We evaluated the proposed framework following the methodology in Section 5.2. For code2vec we used the open-source code and modified it to work with our RL agent implementation. To run our RL algorithms, we used RLlib [122] and Tune [161], open-source libraries for RL that offer high scalability, hyper-parameter tuning, and a unified API for a variety of applications. RLlib and Tune are built on top of Ray [149], a high-performance distributed execution

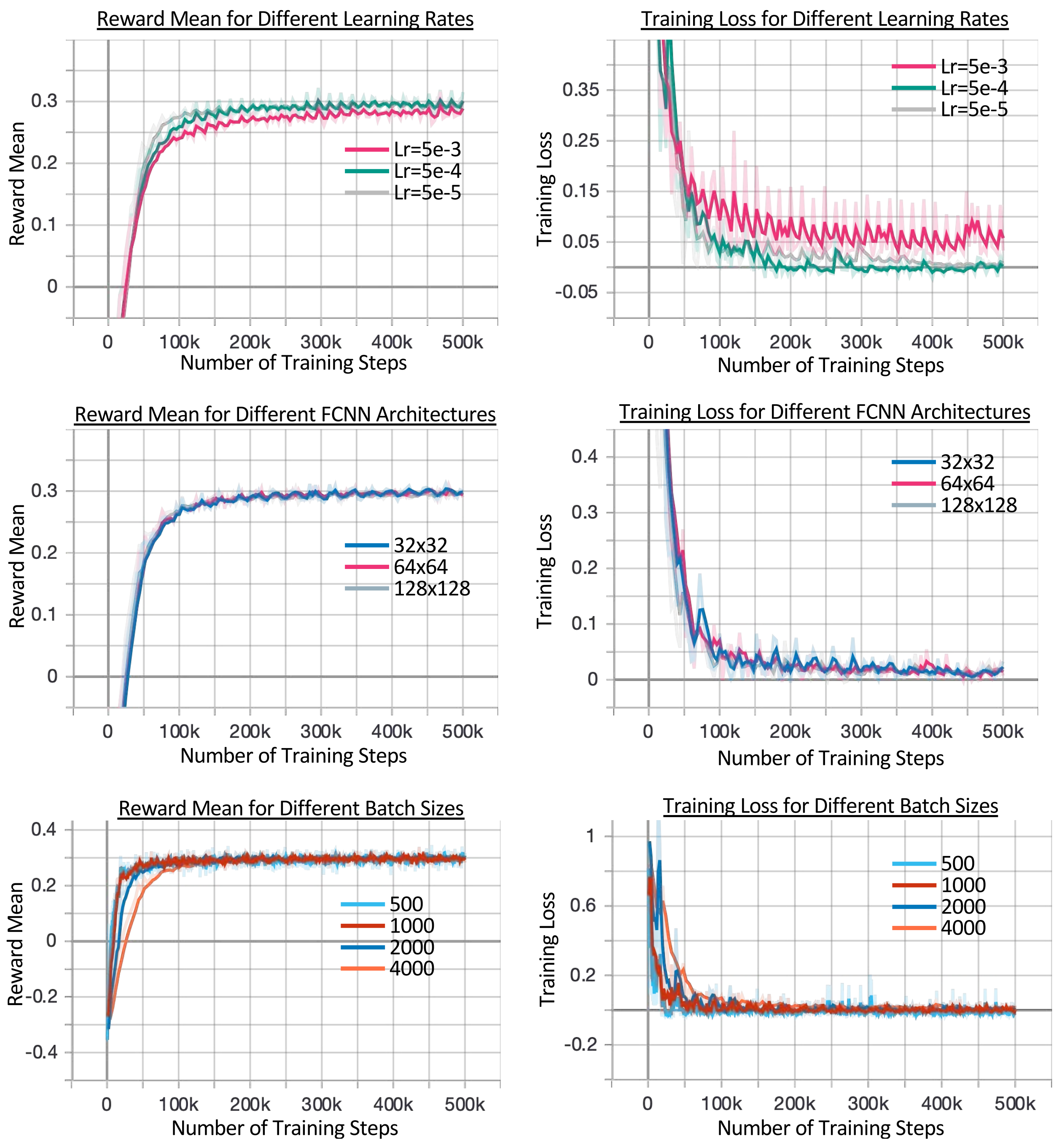


Figure 5.6: Reward mean and training loss for different learning rates, FCNN architectures, and batch sizes.

framework targeted at large-scale machine learning and RL applications. We first trained the framework with the RL agent and code2vec until convergence. We then ran a brute-force

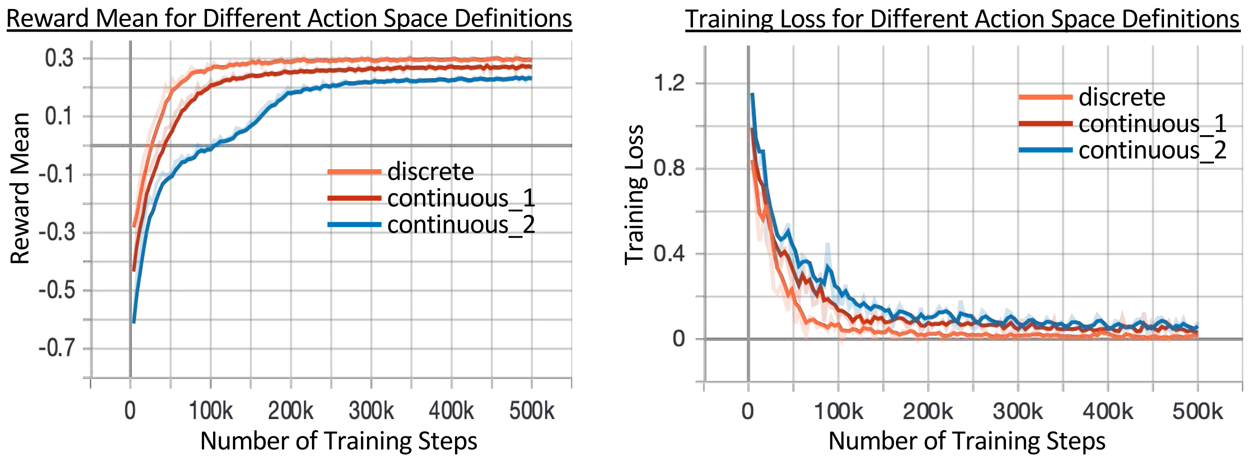


Figure 5.7: Reward mean and training loss for different action space definitions.

search on the dataset to find the best vectorization factors and used them as labels for NNS, the decision tree, and the supervised FCNN. Since the brute-force search requires a long time to run, we limited our training set to 5,000 samples and used this set for the rest of our evaluation. To report performance, we took twelve completely different benchmarks from the test set. These benchmarks combine completely different benchmarks from the LLVM test suite. These benchmarks include loops with different functionality and access patterns, for example, predicates, memory accesses with different strides, bitwise operations, unknown loop bounds, if statements, unknown misalignment, multidimensional arrays, summation reduction, type conversions, and different data types. We compared the performance of our framework to Polly and the baseline cost model.

We started with a 64×64 FCNN, with training batch size of 4,000, a learning rate of 5e-5 — a hyperparameter which determines to what extent newly acquired information overrides old information — and discrete actions. We then experimented with changing one parameter at a time. For discrete actions, the neural network picks two integer numbers that index into the arrays of possible VFs and IFs. We experimented with different hyperparameters. Figure 5.6 shows a hyperparameter sweep over different hyperparameters as function of the number of training steps, *i.e.*, compilations. We trained up to 500,000 steps to see whether more training yields better rewards, but it is clear that the policy converges with much fewer steps.

These results show that the current framework is robust to noise and different parameters. When the learning rate was set to 5e-5, the reward mean reached the maximum in the shortest time. For learning rate 5e-3, the reward mean never reached a higher maximum than that of the smaller learning rates and the training loss was the highest. Minor differences were observed for the different FCNN architectures. We also tried single hidden layer networks and deeper networks, not included in the figures because the results were similar. The policy converged with fewer samples as the batch size was decreased. We also experimented with

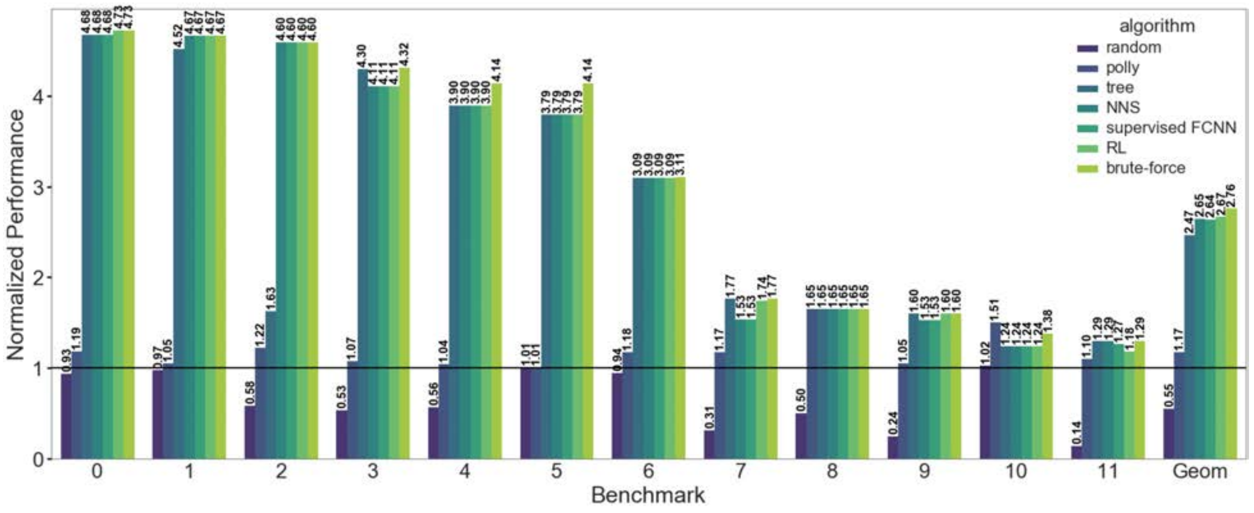


Figure 5.8: The performance of the proposed vectorizer compared to brute-force search, Polly and the baseline cost model. The vectorizer can be configured to use NNS, random search, decision trees, and RL. The performance is normalized to the baseline.

smaller batch sizes; these resulted in unstable policies that did not outperform the scenario when the batch size was set to 500.

The results also show that the policy converged and arrived at a highly rewarding state with 5,000 samples (for the lowest batch size), 35× less than that required for a brute-force search or a supervised learning method. Note that higher than 0 means better, on average, than the baseline, according to the reward definition in Section 5.3. It is important to emphasize that this training is performed once and that the framework can be used later for inference, which requires a single step only, similar to the baseline cost model. By contrast, a brute-force method would require repeating the search.

Figure 5.7 shows the reward mean and total training loss as function of the number of training steps for different action space definitions. We experimented with three action space definitions: ① a discrete action space, where the agent picks two integer numbers that correspond to indices in the arrays of VFs and IFs, ② a continuous action space where the agent picks one continuous number that encodes both the VF and IF, and ③ a continuous action space where the agent picks two continuous numbers that encode both the VF and IF. The numbers in the continuous action spaces are rounded to the closest integers. The results show that the discrete action space performs the best.

The performance on different benchmarks for the baseline, random search, Polly, decision tree, NNS, supervised FCNN, and RL and brute-force search are shown in Figure 5.8. RL outperformed the baseline by 2.67× on average and achieved performance only 3% worse than that of the brute-force search. The availability of more vectorizable instructions in a benchmark was a key factor in the performance difference. NNS and decision trees also performed well, with respectively 2.65× and 2.47× better performance than the baseline. This shows that the embedding learned by the code embedding generator during the end-to-end training is good; thus other learning methods that cannot be trained end-to-end can also perform well using this embedding.

Random search performed much worse than the baseline. This shows that the framework learned a structure in the observations that manifested in the vectorization decisions it made. Polly outperformed the baseline by 17% but performed 56% worse than the proposed RL solution. In benchmark #10, it outperformed the brute-force search because it performs loop transformations that optimize beyond vectorization. This shows the potential for further improving performance by combining Polly and deep RL. We plan to explore this in future work.

While NNs and decision trees cannot be trained end-to-end and require special handling, the supervised FCNN can be trained end-to-end and performs comparably to deep RL. However, RL does not require labels and thus can be trained without a brute-force search. To demonstrate the advantage of deep RL, Figure 5.9 shows its normalized average (geomean) performance compared to supervised FCNN as a function of the number of compilations required (samples). Deep RL performs very well with as few as 5,000 compilations (only 5% worse than the peak) and 1.26× better than the supervised FCNN. Supervised FCNN achieved this only after 70,000 compilations, making it 14× less sample efficient than deep RL. Furthermore, in the long run, we believe that deep RL can better handle large search spaces with multiple objectives to co-optimize.

Transfer Learning

To assess how well the framework generalizes to a completely new code, we evaluated the trained model on two benchmarks: MiBench [162], where the loops constitute a minor portion of the code, and PolyBench [163], where the loops constitute a major portion of the code. MiBench is a set of free and commercially representative embedded benchmarks for telecommunication, networking, security, office, and automation. Note that some of the MiBench benchmarks cannot be vectorized. For example, due to memory dependencies, control-flow or lack of loops, it was not possible to vectorize *adpcm, dijkstra, basicmath, blowfish, etc.* PolyBench includes benchmarks that perform matrix operations, decomposition, and linear algebra, on which Polly is optimized to run.

Figure 5.10 shows the performance of deep RL, Polly, and the baseline on PolyBench. Deep RL achieves on average 3.42×[4] better performance than the baseline and 1.33× better than Polly. Despite Polly being optimized to run on PolyBench, deep RL outperformed it on three out of the six benchmarks. Polly nonetheless performed better on some benchmarks due to its ability to perform loop transformations that optimize beyond vectorization. The lack of enough benchmarks in the dataset leads the deep RL agent to be less generalizable to new

[4]Note that we take the average performance improvement over multiple inferences. If instead we take the best performance, the deep RL improves by 4.77× on average: 3.71× for *2mm*, 6.74× for *bicg*, 6.92× for *ajax*, 5.21× for *gemm*, 1.61× for *gemver*, and 8.16× for *cholesky*.

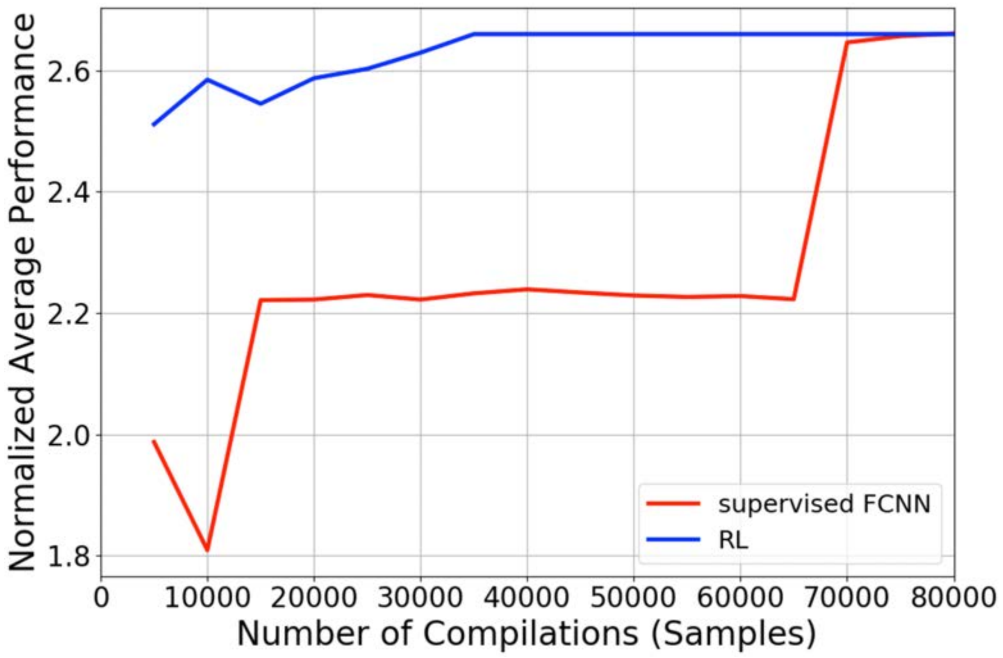


Figure 5.9: Normalized average performance of supervised FCNN and deep RL as a function of the number of compilations (samples) used for training.

benchmarks. The high penalty incurred by the deep RL agent due to the long compilation time also worked to Polly's advantage: when the RL agent tried to increase the values of VF and IF, the reward sometimes decreased due to the time factor. In such cases, the agent learns to avoid being overly optimistic about increasing these values. With more training data the agent can generalize better to larger loop bounds on new examples. When Polly is combined with deep RL, the average potential performance improvement is 4.35×.

Figure 5.11 shows the performance of deep RL, Polly, and the baseline on MiBench. Deep RL outperforms both Polly and the baseline in all the benchmarks. The average performance improvement was 1.1× over the baseline. This result is promising despite the improvement being small as the benchmarks did not rely heavily on loops, and the measured execution time was for all the code not restricted to loops.

Discussion: Deployability

Vendors and commercial companies are generally reluctant to adopt machine learning and deep learning methods in compiler optimization. The main reason is their need for methods that are deterministic, simple, easy to explain, and performant on a large scale of applications. This also explains why most of the optimizations and implementation in compilers are based on manual engineering and heuristics. With that being said, we believe that the growing complexity in systems and workloads, along with the increasing availability of data, are

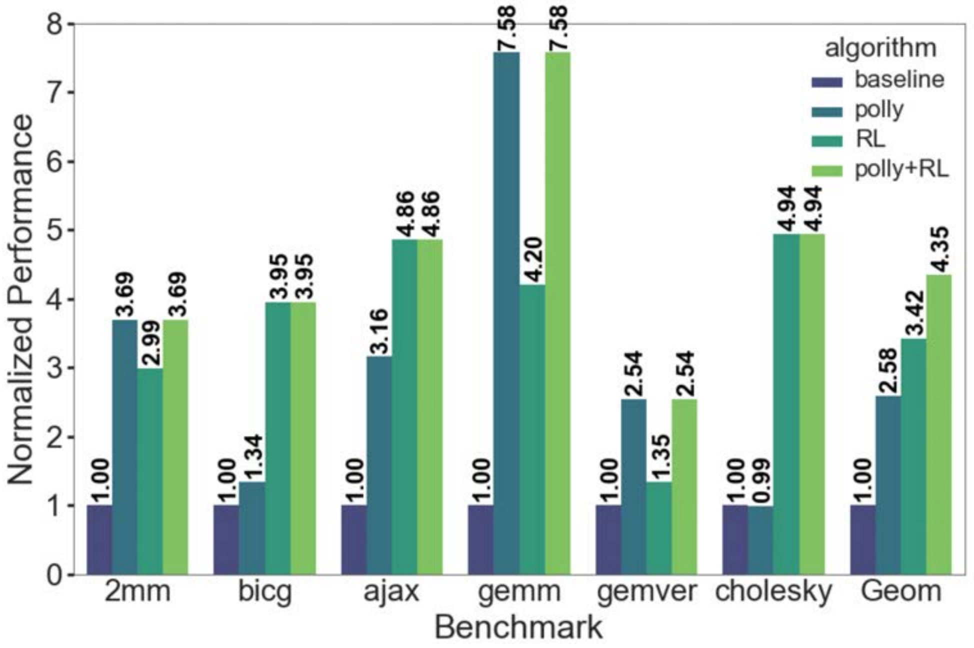


Figure 5.10: The performance of the proposed vectorizer on Polybench compared to Polly and the baseline cost model. The performance is normalized to the baseline.

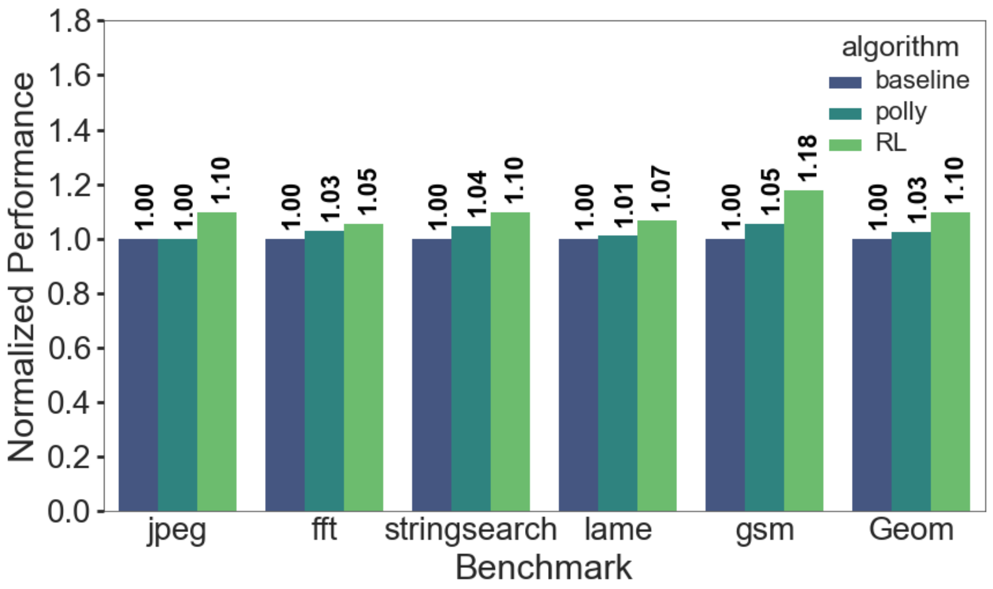


Figure 5.11: The performance of the proposed vectorizer on Mibench compared to Polly and the baseline cost model. The performance is normalized to the baseline.

best addressed by learning-based approaches. Deep RL and other deep learning methods present a unique opportunity to address these compiler challenges end-to-end and improve upon manual engineering. In our evaluation, we showed that deep RL can generalize to new benchmarks. With enough training data, deep RL can be deterministic and performant on a large scale of applications. Since it will be applied mainly for inference, it will also be simple to use and deploy. The main challenge will remain in interpretability. This challenge is not only a limitation of deep RL in vectorization, but it is also a limitation of neural networks in general. Many recent works focus on explaining neural network decisions [164]; better understanding of these decision-maker processes will benefit the use of deep RL in code optimization. Moreover, neural networks have been adopted to solve many advanced real-world challenges regardless of the interpretability limitation. We believe that compilers and code optimization should also follow.

5.5 Related Work

Previous work has utilized machine learning in compiler optimization [16, 17]. For example, deep supervised and RL methods were proposed in [5, 13, 28] to overcome the phase ordering challenge. In [49], multiple machine learning methods for automatic vectorization were proposed. Our work differs in that it is the first to propose a solution based on deep RL to explore the vectorization space and compiler optimization in general. Moreover, all these works rely primarily on extracted/engineered (hand-crafted) features from the program, *e.g.*, arithmetic intensity, memory operations, number of different instructions, or distance between producer and consumer. However, these features do not fully represent the original code. Our work addresses automatic vectorization by learning useful features in an end-to-end fashion, from the text code itself to the optimal factors without any loss of information. In [51] end-to-end supervised deep learning is used to learn compiler heuristics. While such an approach can achieve comparable performance, finding the labels for training can be time consuming, and optimizing for multiple objectives with large search spaces can be challenging.

Automatic vectorization with other methods has also been proposed. For example, the currently implemented cost model in LLVM and recently proposed cost models in [14, 154, 165] rely on predefined cost functions that calculate the expected execution time of a vectorized loop based on a linear formula from the instruction distribution. In [166], superword level parallelism (SLP) [167] is exploited to prevent unnecessary vectorization. But loop vectorization is not addressed and the baseline cost model is used to predict when some portions of code are better off not vectorized. In [168] heuristics are used to guide automatic vectorization. Finally, [169, 170] improve SLP and rely on fixed cost models such as weighted instruction count or the current LLVM cost models.

5.6 Conclusion and Future Directions

Our end-to-end NeuroVectorizer framework automatically detects loops, learns their structures, and applies deep RL to inject vectorization pragmas to the compiler. NeuroVectorizer can generalize to unseen programs and demonstrates an average performance improvement $1.29\times -4.73\times$ compared to the baseline cost model implemented in LLVM and performs only 3% worse on average than the brute-force solution.

We see multiple future directions for this work. Computation cost can be reduced using loop polyhedral analysis, which operates on loop snippets for the code embedding. Combining deep RL and Polly can further boost the performance, and training the RL agent to predict using Polly will be advantageous. The deep RL vectorizer can also be employed at the intermediate representation level, which can better reflect the effects of the vectorization on the code. This in turn could help the agent learn better predictions. Features representing different underlying architectures should be added and separate models trained for each as different architectures behave differently and have different VF and IF action spaces. In this work, we showed the potential of the deep RL vectorizer as the first step toward end-to-end code optimization with machine learning and deep RL. However, in order for this vectorizer to become a standardized optimization stage in the LLVM compilation stack, training must be conducted on a wide range of applications and target architectures.

In our approach, we assumed the agent makes a single decision per loop nest (*i.e.*, episode). However, with new compiler features such as the support of vectorization at different levels of a nested loop, the deep RL agent will not be restricted in this way. It will be able to perform multiple sequential decisions that collectively form a single episode of multiple actions and states.

Pragmas such as loop unrolling, distribution, and vector predication can be similarly tuned. The user needs only to define an appropriate action space and a reward function that depends on the desired objective. Many compiler optimizations today are global rather than local. For example, the phase ordering of compiler passes is applied indiscriminately to all the functions in the code. By fine-tuning the pragmas, different phase orderings and optimizations can be determined automatically for different sections of the code.

Our framework can also support vanilla deep neural network methods instead of deep RL. One direction we are exploring is to use a neural network that learns a ranking scheme on the VFs and IFs. Given an embedding and pragmas, the network can learn the execution times normalized to the non-vectorized code. This is equivalent to learning a new cost model for each VF and IF. These models could replace the baseline cost models. This method – unlike NNs and decision trees – can be trained end-to-end.

Chapter 6

ProTuner

6.1 Introduction

Most deep learning and image processing programs rely heavily on loops, which represent the vast majority of a program's total execution time. The large number of loops in each loop nest makes it possible to schedule them in many different, yet functionally equivalent ways. Choosing a bad program schedule can dramatically increase execution time. Similar schedules can include different optimizations such as inlining, tiling, vectorization, and multithreading, which can significantly impact program performance. The number of possible optimizations grows exponentially with the number of loops in the loop nest. With the end of Moore's law and the boom of new, application-specific integrated circuits (ASICs), the scheduling challenge becomes harder as the scheduler also needs to generate different schedules for different target architectures.

Due to its complexity, scheduling is often done using heuristics and tremendous amounts of hand engineering. Big vendors often hire engineers whose only job is to manually write schedules for different applications. This incurs huge costs both in terms of time and human capital. Heuristics mostly fall short of optimal performance [13, 18, 50]. Ideally, the scheduler should consider all the possible schedules and the target hardware to find the optimal schedule. Unfortunately, the space of possible schedules for different hardware targets is prohibitively large to explore. To cope with that, a recent work [12] proposed using beam search with a learned cost model to find good schedules. While this approach achieves promising improvements over the baseline default auto-scheduler in Halide [65], it often fails to find the optimal schedule. The main problem is in the greediness of beam search and the inability of the cost model to accurately predict the performance of partially scheduled (not meaningful) programs, which is required at every intermediate step of the beam search. This results in a multiplied error at every decision made in the beam search tree and an inability to explore schedules that are less rewarding in the short term but potentially more rewarding in the long term.

ProTuner overcomes these challenges by using Monte Carlo Tree Search (MCTS) [171].

In ProTuner, the scheduling problem is formulated as a Markov decision process (MDP) where each intermediate schedule is represented as a state and the actions are the different intermediate optimizations that could be applied next. The reward is proportional to the execution time improvement. The solution would be the actions that lead to the optimal schedule; hence, solving the MDP can guarantee the optimal schedule.

One promising algorithm for solving MDPs is MCTS. MCTS builds a search tree using selection, expansion, simulation, and backpropagation to explore the search space. After some number of iterations, the tree decides which next step (of intermediate scheduling optimization) to perform next. When this happens, a new root is determined and the MCTS starts again. With the upper confidence bound (UCB) [42], MCTS is guaranteed to converge to the optimal solution after enough iterations.

MCTS makes decisions by looking ahead, evaluating complete schedules, avoiding greediness, and considering the expected long-term reward of scheduling decisions, which also makes it more resilient to noise in the cost model. To test our conjecture that MCTS is a better fit than beam search for finding the optimal schedule, we implemented ProTuner with MCTS on top of Halide [65] and evaluated its performance. ProTuner with MCTS outperformed or performed comparably to beam search on all the evaluated benchmarks, with up to 3.25× speedup on the best performing benchmark.

This chapter makes the following contributions:

- Formulates the scheduling problem as an MDP and solves it using MCTS with the UCB.
- Explains how ProTuner was implemented on top of Halide and explores different MCTS techniques to improve and fine-tune its performance.
- Presents rigorous evaluations that show a performance improvement of up to 3.25× better than beam search on a suite of 16 real benchmarks.
- Shows how ProTuner can combine real execution time evaluation with the learned cost model to further boost its performance.

We also briefly discuss Ansor [32], which is similar to ProTuner but uses evolutionary search instead of MCTS to explore optimizations outside the search space of other frameworks.

6.2 Challenges in Beam Search

A beam-search based approach [12] was recently proposed as a scheduler in Halide, with state-of-the-art results. In this approach, a cost model is trained as a proxy for predicting the true execution time of intermediate schedules used by the beam search to determine which schedules to choose. Unfortunately, the cost model is trained on fully scheduled programs and cannot predict the execution time of incomplete/partially-scheduled programs. We also

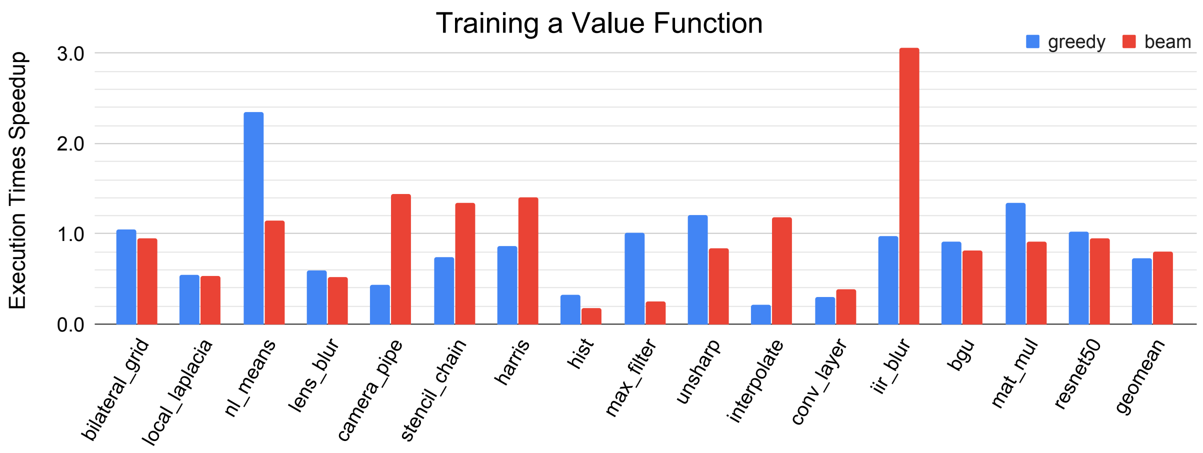


Figure 6.1: Speedup of greedy and beam search with a cost model trained to predict the cost of the future complete schedules. For each algorithm, the speedup is normalized to the performance when the original cost model (trained on complete schedules only) is used.

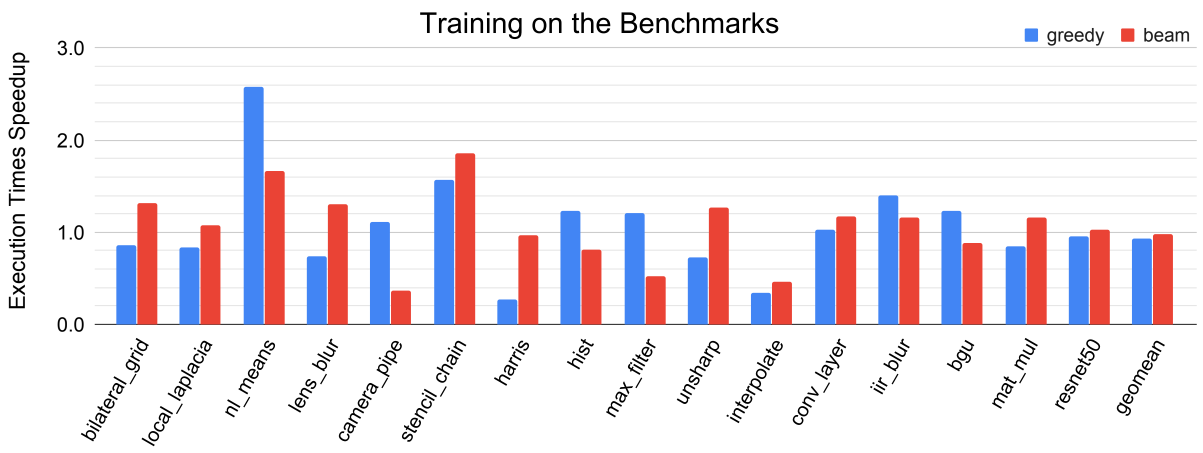


Figure 6.2: Speedup of greedy and beam search with a cost model trained directly on random schedules of all the benchmark algorithms themselves. For each algorithm, the speedup is normalized to the performance when the original cost model (trained on complete schedules only) is used.

observed that the cost model often falls short in properly predicting execution times of fully scheduled programs. Thus, minimal cost does not necessarily mean optimal execution time. This makes the beam search very sensitive to inaccuracies in the cost model. Since beam search queries the cost model at every scheduling decision, this error aggregates.

To illustrate the disadvantage of beam search, we perform two experiments for predicting the execution time of incomplete programs. In the first we trained a cost model as was done in [12], except that we trained it on the fully scheduled benchmarks that we later run the search on. Figure 6.2 shows the results on beam search and greedy search (*i.e.*, beam search with a beam size of one) with the new cost model. We observed that the performance improves for some benchmarks while for others it deteriorates and overall the performance is similar. This was also observed by the authors in [12] when they retrained their cost model on the specific benchmark programs that they were also autotuning. Even if the model is trained on the benchmarks on which we later run the inference, it is hard for the cost model to accurately predict the execution time of incomplete schedules during the search. In the second experiment, shown in Figure 6.1, we trained the model to predict the future cost of the current schedule. This also did not work well because there are multiple options for scheduling the rest of the program; thus, the same partial schedule can lead to different costs.

These problems can be avoided if we formulate the scheduling problem as an MDP. Because a graph node represents an intermediate schedule/program, and edges between nodes represent potential scheduling actions, applying a particular schedule to a program can be seen as a simple graph traversal. The goal is to find a node/set of actions that maximizes the reward of the end state. In our use case, the reward would be the inverse of the execution time (thereby ensuring that maximizing the reward gives the fastest program). This motivated our choice of MCTS, which, in addition to the advantages mentioned in Section 6.1, is theoretically guaranteed to find the best node with sufficient time. Moreover, its UCB formula strikes a good balance between the exploration of new states and the exploitation of existing states, thus leveraging the tradeoff discussed in Chapter 5.

Finally, MCTS allows us to combine real execution time measurements and the cost model's predictions to further improve performance.

6.3 The Proposed ProTuner Scheduler

Our MDP is defined by actions that correspond to intermediate scheduling decisions and states that represent intermediate schedules. The cost is the execution time of the schedule. To enumerate the possible schedules and evaluate their costs, we use the same techniques used in [12]. Given an n-dimensional tensor, the scheduling is split to n stages. Starting from the last stage and back to the input, a new scheduling decision is made at each stage, with a new tiling and a compute and storage granularity for inserting the new stage. The new tilings can be unrolled or spread across parallel threads or single instruction multiple data (SIMD) lanes. The costs of the complete schedules (at the end of simulation) are evaluated using a cost model trained on random, fully scheduled programs.

Figure 6.3 shows the block diagram of ProTuner. The MCTS starts from the last stage and explores the possible schedules back to the inputs. Every simulation from one node in the MCTS ends by computing the cost from the cost model and backpropagating to the parent nodes with the terminating fully scheduled state. These nodes update the future best cost so far, the terminating state, and the value function that stores the *average* cost so far.

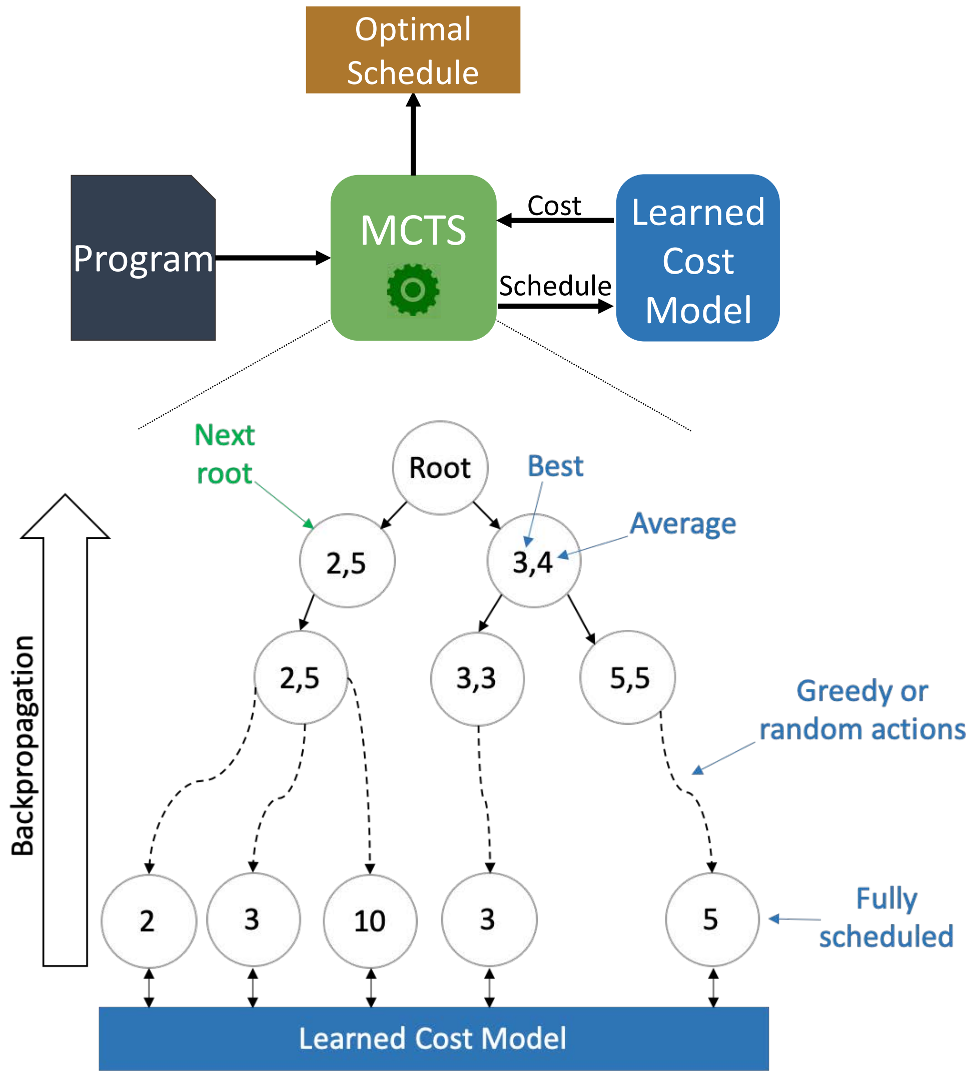
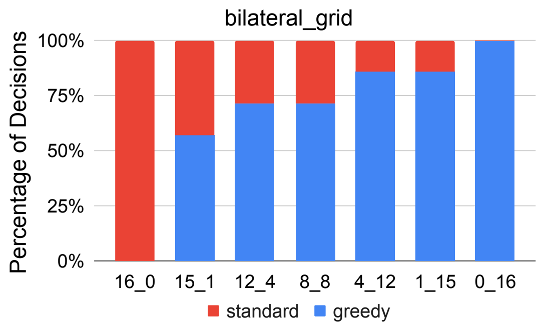
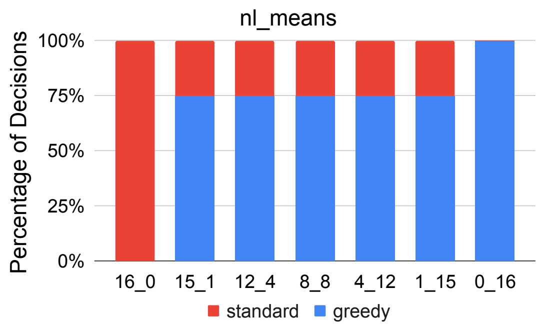


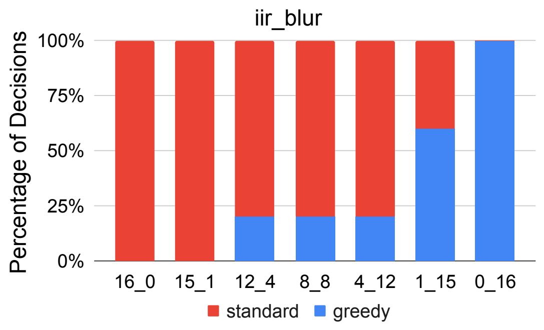
Figure 6.3: The block diagram of ProTuner. The program is fed to the MCTS, which interacts with the learned cost model to find the optimal schedule. To make each intermediate scheduling decision, the MCTS explores the benefits of the possible next actions based on the average cost but eventually picks the root that leads to the best cost. Each node stores the average costs, the best cost so far, and the complete schedule that has this best cost. The simulation can either be greedy or random. The backpropagation returns costs or 0/1 based on whether it outperforms the global best. When running an ensemble of MCTSes, the next root is picked to be the best from all the best roots.



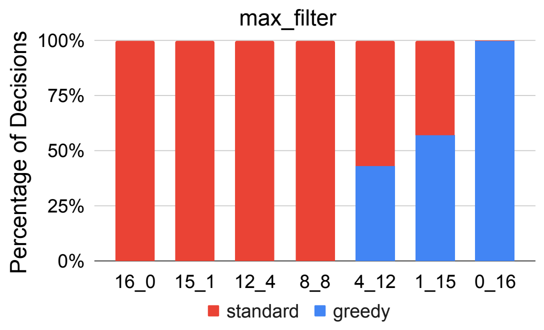
(a) Proportion of decisions made by standard and greedy MCTSes on the `bilateral_grid` test.



(b) Proportion of decisions made by standard and greedy MCTSes on the `nl_means` test.



(c) Proportion of decisions made by standard and greedy MCTSes on the `iir_blur` test.



(d) Proportion of decisions made by standard and greedy MCTSes on the `max_filter` test.

Figure 6.4: The proportion of decisions made by greedy MCTSes as a function of different numbers of standard and greedy MCTSes on a suite of four real applications. X_Y corresponds to X standard MCTSes and Y greedy MCTSes. The overall number of trees is 16.

During the search the MCTS uses the average cost to determine the next child to explore. We tried using the best cost in the search but that resulted in non-smooth value functions where the children that got lucky earlier and found better costs performed significantly more simulations than less lucky children. This often results in the greedy behavior we are trying to avoid.

When the computation budget is reached either after exceeding the number of allowed iterations or due to time out, a winning action (schedule) for the current stage is determined and the new root is the state this action leads to. The winner is determined based on the *best* cost so far, as in [172]. We found this method to outperform taking the child with the best *average* cost by 25%. This is mainly because it can guarantee that later steps need to find schedules that are better than the best so far (rather than average best), which can be helpful when fewer iterations are available. This also allows us to combine real execution time measurements with cost model predictions at a negligible overhead.

To further improve our results, we ran multiple MCTSes in parallel across multiple cores that synchronize when picking a new root at every intermediate scheduling decision, which is the best child from all the best children of all the MCTSes. In addition to the performance benefits conferred by this parallelism, an ensemble of MCTSes is proven to outperform a single MCTS with the number of iterations equal to the combined number of iterations available in the ensemble [173]. Furthermore, MCTS does not need to evaluate the costs of all the children during simulation or compute their state features (which we found to consume more than 92.3% of the overhead in beam search). Instead, it randomly and continuously picks a possible child and only evaluates one state when it is fully scheduled.

Improving Scheduling Time by Adding Greedy MCTSes

We explored multiple techniques to improve the scheduling time of our MCTS. We found that adding some greediness to our algorithm allows it find good schedules in a shorter time. First, we explored adding greediness by picking the best next action with probability $\frac{1}{2}$ instead of randomly picking an action during the simulation phase. This, however, offered no benefits over simulating random actions. We then tried instead to mimic the MCTS scheme in single-player games with 0/1 rewards. When running from the first root, MCTS finds the best cost, and then the children that later become roots get 1 point if they beat their parent's cost, and 0. This normalizes the reward, simplifies the hyperparameter tuning of the cost, and forces the new roots to beat the costs of their ancestors. However, this resulted in 9% worse performance.

What finally worked very well was to combine standard MCTSes with an MCTS that simulates greedily. In the latter, after the node to be expanded based on the UCB formula is determined, it is expanded with a `random` child but the simulation is done purely greedily using the cost model. To determine how many trees should simulate randomly and how many greedily, we experimented with different numbers of random or greedy MCTSs on four real applications, as shown in Figures 6.4 and 6.5. We found that applications such as `bilateral_grid` and `nl_means` benefited from adding greedy MCTSes while `iir_blur` and `max_filter` did not. We also observed that it is sufficient to use a single MCTS that simulates greedily as it struck a good balance between greediness and uniform exploration. Figure 6.4 shows the number of decisions made by greedy MCTSes as a function of different numbers of random and greedy MCTSes. Adding more greedy MCTSes did not change the number of decisions made by greedy MCTSes for `nl_means`, which benefits from greediness. For `bilateral_grid`, the number of decisions made by greedy MCTSes increased slightly, but this did not impact the performance as we found that greedy MCTSes often found similar best states. For `iir_blur` and `max_filter`, which benefit mostly from standard MCTSes, adding more greedy MCTSes slightly increased the number of decisions made by greedy MCTSes but resulted in worse performance.

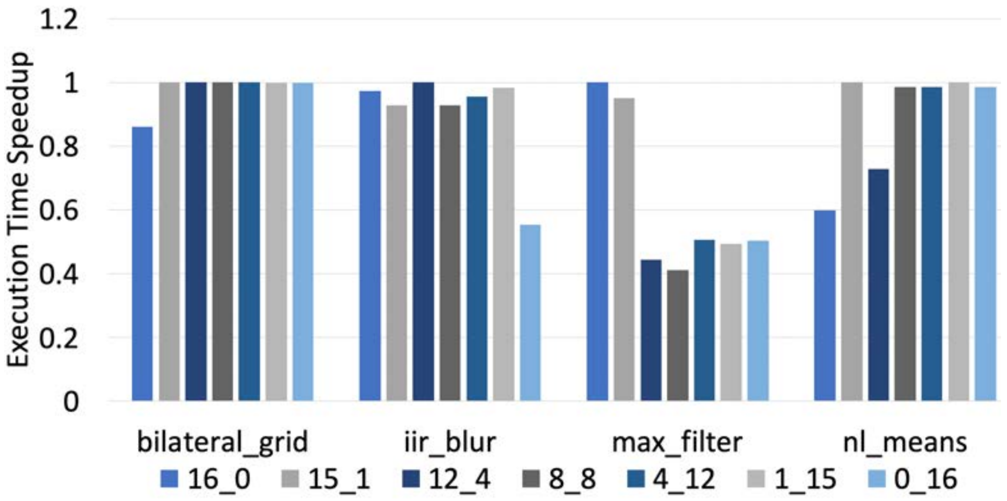


Figure 6.5: The execution time speedup to the best execution time on a suite of four real applications (higher is better). X_Y corresponds to X standard MCTSes and Y greedy MCTSes. The overall number of trees is 16. The 15_1 setting did best overall.

Combining the Cost Model and Real Execution Time Measurement

Despite their inaccuracies, cost models are often used because calculating the real measurement is time consuming. To compensate for inaccuracies in the cost model while incurring minimal additional overhead, we added real execution time measurements at every iteration where a new root is declared. Our final algorithm is shown in Figure 6.6. The algorithm initializes one greedy MCTS and 15 standard ones. While the program is not fully scheduled, it runs the MCTSes in parallel from the current root. The returned best roots are evaluated based on the best real execution time measurement (the commented line).

To implement real execution time measurement in our C++ code, the head thread `forks` multiple children. Each compiles a benchmark application serially for each of the candidate schedules returned by the greedy and standard MCTSes. Afterward, each of the compiled benchmark applications is run serially (to make sure they do not interfere with each other's run). The schedule with the best real execution time, rather than the one with the lowest cost, is used as the new root of the MCTSes for the next iteration.

To compile the benchmark applications, we used Halide's rudimentary "RunGen" wrapper, described in the official documentation [175]. RunGen wraps a solitary scheduled Halide function (which is what we compiled) into a simple benchmark application that returns the execution time of the scheduled program.

The RunGen wrapper fails to compile an error-free program for various applications, such as `camera_pipe` and `bgu`. For these applications, we instead compiled the scheduled functions with the custom wrappers found under the `apps` directory in the official Halide repository.

```
all_mcts=[]
all_mcts.append(init_greedy_mcts())
all_mcts.extend(init_standard_mcts(num_mcts=15))
current_root = state0 //empty schedule
best_fully_scheduled_states={}
next_best_roots={}
while(!fully_scheduled){
  parallel_for(i=0...15){
    best_fully_scheduled_states[i],
    next_best_roots[i] =
        all_mcts[i].run(root=current_roots[i])
  }
  best_index = get_best_state_index_from_costs(
                    best_fully_scheduled_states)
  /* Uncomment the next lines to evaluate
  the real execution time instead of
  estimated cost:
  best_index =
      get_best_state_index_real_measure(
        best_fully_scheduled_states) */
  parallel_for(i=0...15){
    current_roots[i] =
        next_best_roots[best_index]
  }
  optimal_schedule =
      best_fully_scheduled_states[best_index]
  fully_scheduled =
      next_best_roots[best_index].is_leaf
}
```

Figure 6.6: Pseudocode of the MCTS scheduling algorithm that combines 15 standard MCTSes and one greedy MCTS. The best next root can be determined based on the best cost of the best fully scheduled states or based on the best execution time measurement of the best fully scheduled states as shown in the commented line.

We could not use real execution time measurements for ResNet50 as it requires many simultaneously scheduled functions that a single forked child cannot run on its own. Therefore, for ResNet50, we report the results from the MCTS run that schedules it using the cost model only.

Name	Seconds for Iteration	Expansion Formula	Measurement
`mcts_30s`	30	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+\sqrt{\frac{ln(n)}{n_j}})$	cost model
`mcts_10s`	10	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+\sqrt{\frac{ln(n)}{n_j}})$	cost model
`mcts_1s`	1	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+\sqrt{\frac{ln(n)}{n_j}})$	cost model
`mcts_Cp10_30s`	30	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+10\sqrt{\frac{ln(n)}{n_j}})$	cost model
`mcts_sqrt2_30s`	30	$\frac{\sum_i \frac{1}{ExecTime_i}}{n_j}+\sqrt{2}\sqrt{\frac{2ln(n)}{n_j}}$	cost model
`mcts_cost+real_30s`	30	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+\sqrt{\frac{ln(n)}{n_j}})$	cost model + real
`mcts_cost+real_1s`	1	$\frac{1}{\frac{\sum_i ExecTime_i}{n_j}}(1+\sqrt{\frac{ln(n)}{n_j}})$	cost model + real

Table 6.1: The MCTS configurations explored. We explored different timeouts (time to determine a new root) in seconds per MCTS iteration, expansion formulas where we modify the UCB, and execution time measurement schemes. `mcts_sqrt2_30s` is the algorithm that gives the most weight to exploration and is the closest to the original UCB formula. `mcts_Cp10_30s` gives the second highest weight to exploration. `mcts_cost+real_30s` combines `mcts_30s` from the first row and real execution time measurement. `mcts_10s` and `mcts_1s` reduce the seconds for iteration to ten seconds and one second, respectively. `mcts_cost+real_1s` combines `mcts_1s` from the first row and real execution time measurement. `mcts_sqrt2_30s` uses the original UCB formula and encourages much more exploration than the other MCTS algorithms. We used $C_p = \frac{1}{\sqrt{2}}$ as suggested in [174], which showed that it works well with rewards in range $[0, 1]$ as it satisfies the Hoeffding inequality. Multiplying the exploitation term with the exploration term encourages early exploitation.

6.4 Evaluation

To evaluate ProTuner we built it on top of Halide in C++. We ran ProTuner on AWS m5.8xlarge instances. These instances run Intel Xeon Platinum 8259CL processors with 16 physical cores with 128 GB RAM and 100 GB SSD storage. AWS often provides different CPUs for different instances even if they are from the same type (*e.g.*, m5.8xlarge can have Intel Xeon Platinum 8259CL processors or Intel Xeon Platinum 8175M processors). To minimize variance between runs, we ran all our results on the same instance, at the same time of the day, when it is night in the time zone and after turning off hyperthreading.

We set the timeout limit for picking a new root in the MCTS to 30 seconds. We also explored reducing this limit to one second and include real execution time measurements. We used 16 MCTSes (one greedy, 15 standard) that run in parallel and synchronize every timeout for picking a new root. For an apples-to-apples comparison, we also ran 16 beam

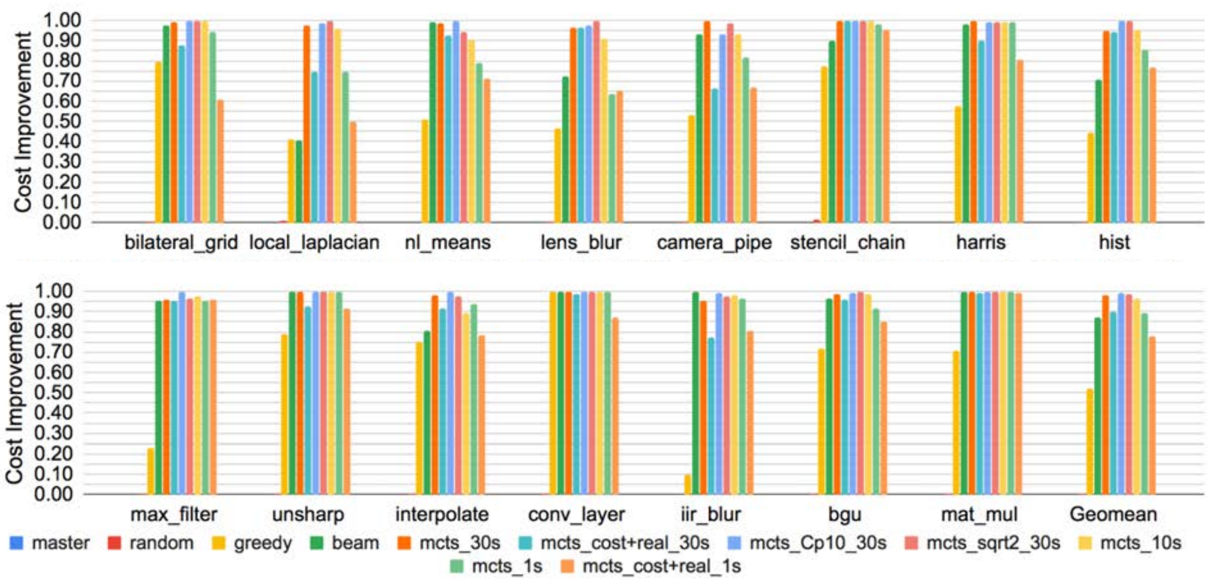


Figure 6.7: The minimum cost found by every algorithm normalized to the best cost found by all the algorithms on a suite of 16 real benchmarks.

searches in parallel. We used the open-sourced code of Halide's beam search algorithm with the artifacts published by the original authors with the same configuration provided in [12]: a beam size of 32 and five passes (iterations of beam search). We also compared our results to a greedy auto-scheduler (beam size of 1), the default scheduler on Halide's master branch, and random search. Random search does not use the cost model. It runs for ten minutes and outputs the program with the best real execution time it found. The other algorithms run with three different seeds and the best performing schedule found by each algorithm is reported.

We auto-scheduled a suite of 16 real applications. These applications range from matrix multiplications to various blurs, convolutions and interpolations, to full implementation of ResNet50 [176]. These applications were taken from the baseline beam search work we compared against and are available in the Halide repository. We experimented with multiple MCTS configurations as shown in Table 6.1. We mainly experimented with different timeouts for determining a new root while running the MCTS algorithm, the expansion formula that determines which child in the tree gets expanded, and integrating real execution time measurements during the search.

Cost

Figure 6.7 shows the minimum costs found by our MCTS algorithms compared to random, greedy, and beam search. The costs are normalized to the best cost found by all the algorithms.

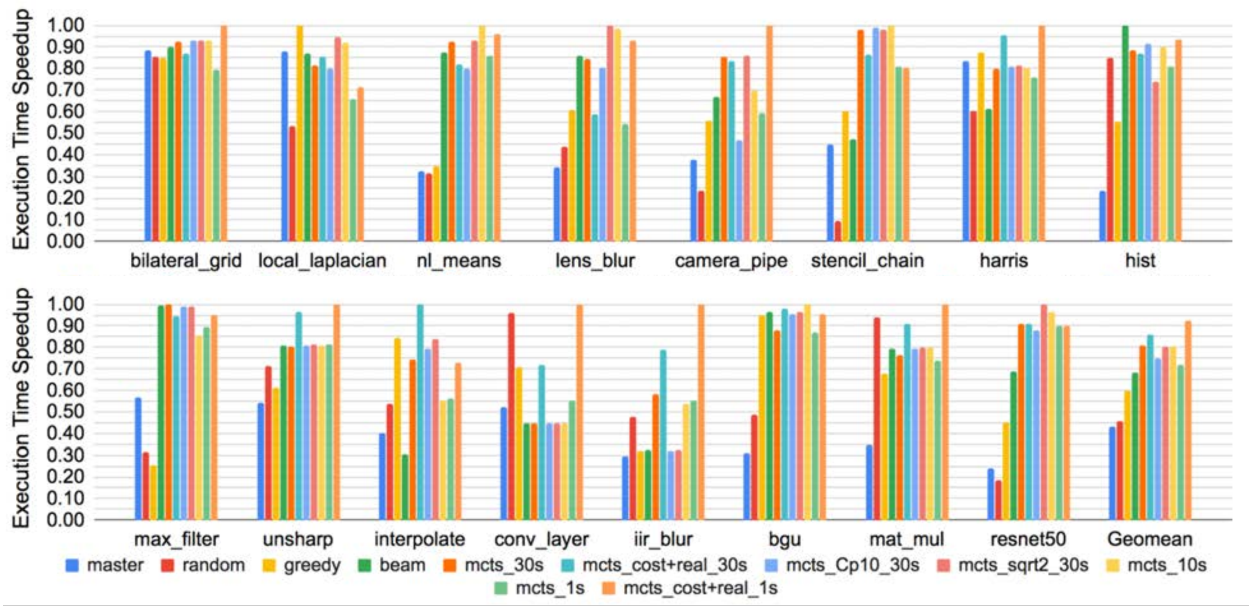


Figure 6.8: The minimum execution time found by every algorithm normalized to the best execution time found by all the algorithms on a suite of 16 real benchmarks.

The cost of ResNet50 is omitted because the application includes multiple stages, each of which is auto-scheduled separately (with costs in different ranges), after which the stages are merged back to form the final application. All our MCTS configurations outperformed beam, greedy, and random search cost-wise in geometric mean. This means that with a 100% accurate cost model, our MCTS performs better than beam, greedy, and random search. `mcts_Cp10_30s` outperformed beam search in all the benchmarks and `mct_30s` outperformd it in all the benchmarks except `iir_blur`, where its cost-wise performance was 4.5% worse than beam search. `mct_10s` outperformed beam search in all the benchmarks except `nl_means` and `iir_blur`, where its cost-wise performance was 8.9% worse than beam search for the former and 1.1% worse for the latter. `mcts_sqrt2_30s` likewise outperformed beam search in all the benchmarks except `nl_means` and `iir_blur`, where its cost-wise performance was 5.2% worse than beam search for the former and 2.4% worse for the latter. `mcts_cost+real_30s` and `mcts_cost+real_1s` achieved the worst geometric mean cost among the MCTS algorithms. This means that they found schedules with better execution times but at higher costs.

Execution Time

Figure 6.8 shows the minimum execution time each algorithm found normalized to the best execution time found by all the algorithms. Note that while better costs generally mean better execution times, the improvement ratios are not similar. Time-wise in geometric mean, however, all the MCTS algorithms outperformed beam search (1.06×-1.36×), similar to the

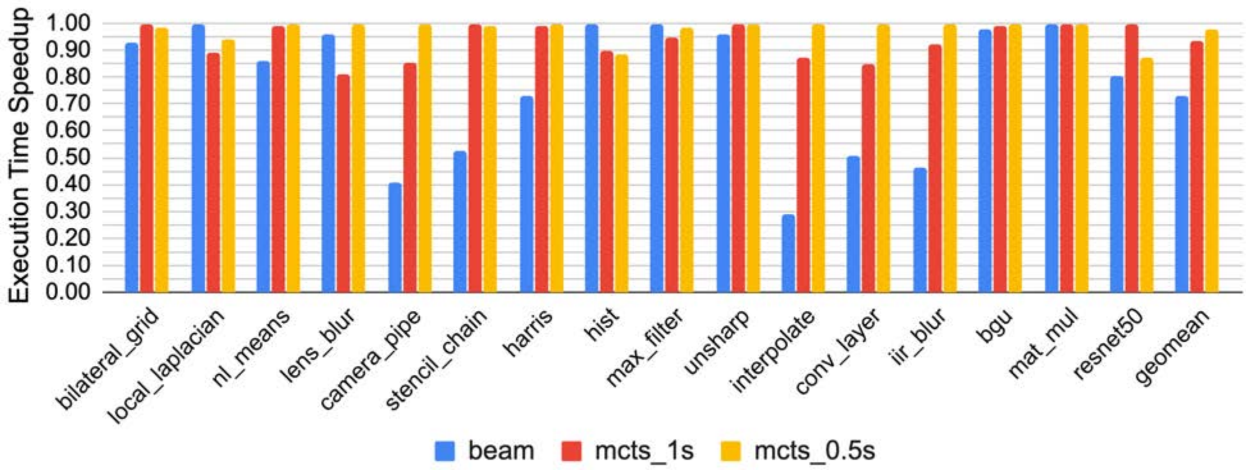


Figure 6.9: Execution time speedup normalized to the best execution time using beam search, `mcts_1s`, and `mcts_0.5s` with autotuning on a suite of 16 real benchmarks. Each algorithm is rerun with a different seed until a timeout of 15 minutes is reached and the best performance found by each algorithm is reported.

cost-improvement trend. This was true even for `mcts_1s`, which gives one second for each MCTS iteration. The biggest execution time improvement was achieved by `interpolate`, which performed 1.8×-3.25× better in the different MCTS algorithms.

As expected, `mcts_cost+real_30s` and `mcts_cost+real_1s` achieved the best performance in geometric mean, despite not having achieved the best geometric mean in costs. This shows that real execution time measurement is effective. Interestingly, `mcts_cost+real_1s` performed better than `mcts_cost+real_30s`. A clear example can be seen in `conv_layer`, which is the smallest benchmark. While both `mcts_cost+real_1s` and `mcts_cost+real_30s` outperformed the other MCTS algorithms, `mcts_cost+real_1s` performed better as it stops evaluating the cost model earlier and hence it is less likely to overfit to the cost model, especially on smaller benchmarks.

We observe that `mcts_10s` performed similarly to `mcts_30s` and `mcts_sqrt2_30s`, and better than `mcts_Cp10_30s` in geometric mean. This means that 10 seconds per MCTS iteration is sufficient for our benchmarks. Adding more time might be useful for large benchmarks but can be harmful to smaller benchmarks due to the aforesaid overfitting problem. An ideal configuration should consider the size of the application when setting the MCTS parameters.

Search Time

Our MCTS algorithms with the cost model schedule programs in seconds to minutes. For example, the average auto-scheduling times of `mcts_1s`, `mcts_10s`, and `mcts_30s` are 31, 155, 422 seconds respectively. This includes the time to compile the search code, the search time, and the time to compile and benchmark the applications. In smaller applications, such as `conv_layer` and `mat_mul`, this time is dominated by the compilation and benchmarking time. In larger applications, this time is dominated by the search time. Our performance analysis shows that most of the search time (88%) is spent during the generation of new children (schedules) in the simulation phase and only 7.5% of the time is spent in the cost evaluation. However, our standard MCTS simulation needs a single randomly generated child. The rest of the children are generated but not used. Therefore we see a potential for 8× speedup in the search time of MCTS, which we seek to implement in future work.

For `mcts_cost+real_1s` and `mcts_cost+real_30s`, the average auto-scheduling times are 23 and 35 minutes, respectively. Most benchmarks require roughly 3× more time to auto-schedule with real execution time measurement when the MCTS iteration is set to 30 seconds. This time was mostly consumed in the forked children processes that serially compile and evaluate the candidates of potential next roots. While possibly inefficient, serial compilation and evaluation ensures that there will be no interference between threads during execution time measurement.

Autotuning with Limited Time Budget

Figure 6.9 shows the autotuning performance comparison between beam search and MCTS. We limited the autotuning time to 15 minutes for each application. This time includes the compilation and execution time of the benchmarks as well as the search time. The execution time of the generated program was normalized to the best execution time found by beam search and MCTS. For time efficiency we used `mcts_1s`. We further explored using half a second per MCTS iteration (`mcts_0.5s`), which allows for more real execution time measurements during autotuning. Each algorithm was rerun with a different seed until the 15 minute timeout was reached, and the best performance found by each algorithm is reported. `mcts_0.5s` achieved the best overall performance and outperformed beam search by up to 3.43× and by 1.35× in geometric mean, in the same time budget. `mcts_1s` also outperformed beam search by up to 3× and by 1.29× in geometric mean in the same time budget. This was mainly due to more accurate scheduling decisions derived from meaningful costs of fully scheduled programs rather than incomplete programs.

6.5 Ansor: Overview of a Different Approach

Here we give a brief overview of Ansor [32], a tensor program generation framework for deep learning applications. Ansor pursues similar goals to ProTuner but uses evolutionary search instead of MCTS to explore optimizations outside the search space of other frameworks such as beam search. By sampling program schedules from a hierarchical representation of the search space, Ansor is able to explore many more optimization combinations than existing search strategies. This representation decouples high-level structures and low-level details,

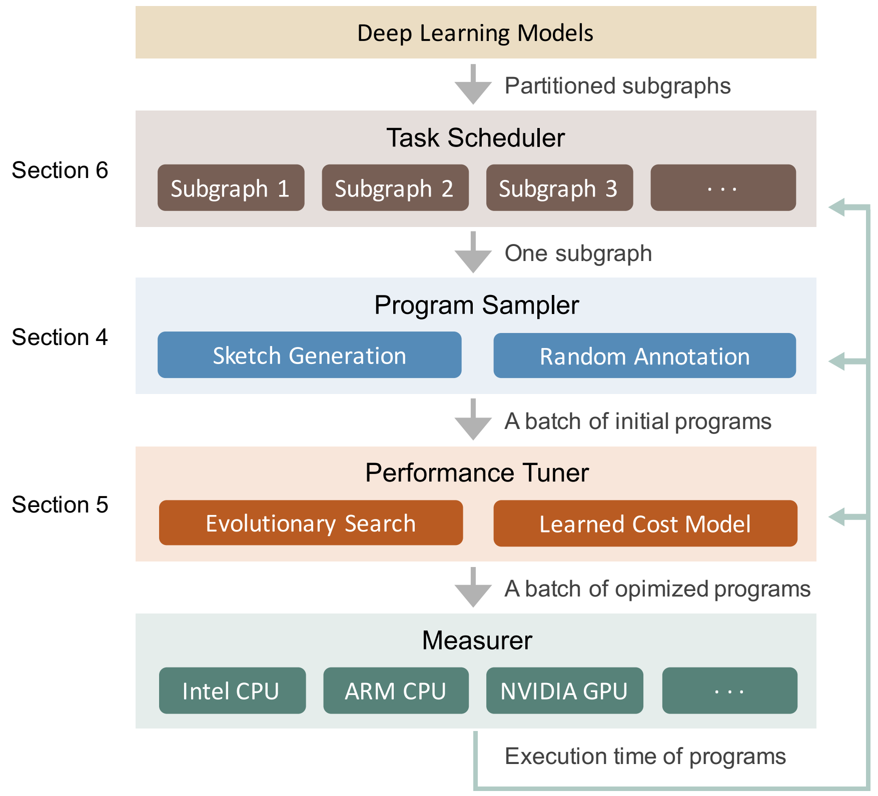


Figure 6.10: System Overview. The gray arrows show the flow of extracting subgraphs from deep learning models and generating optimized program schedules for them. The green arrows mean the measurer returns profiling data to update the status of all components in the system.

enabling flexible enumeration of the former and efficient sampling of the latter. The space is constructed automatically for a given computation definition. Ansor then fine-tunes the sampled program schedules with evolutionary search and a learned cost model to identify the best program schedules. Evolutionary search uses mutation and crossover to perform out-of-order rewrite and break the limitation of sequential construction. Thus, Ansor is able to find high-performance program schedules that are outside the search space of existing state-of-the-art approaches. Moreover, Ansor utilizes a task scheduler to simultaneously optimize multiple subgraphs in deep neural networks.

Figure 6.10 shows the overall architecture of Ansor. The input of Ansor is a set of to-be-optimized DNNs. Ansor uses a tool [177] to convert DNNs from popular model formats (e.g., ONNX [178], TensorFlow PB) to partitioned small subgraphs and then generates tensor program schedules for these subgraphs. Ansor has three major components: (1) a program sampler that constructs a large search space and samples diverse program schedules from it; (2) a performance tuner that fine-tunes the performance of sampled program schedules; (3) a task scheduler that allocates time resources for optimizing multiple subgraphs in the DNNs.

Program sampler. One key challenge is to generate a large search space for a given computational graph. To cover diverse tensor program schedules with various high-level structures and low-level details, Ansor utilizes a hierarchical representation of the search space with two levels: *sketch* and *annotation*. Ansor defines the high-level structures of program schedules as sketches and leaves billions of low-level choices (*e.g.*, tile size, parallel, unroll annotations) as annotations. Ansor includes a program sampler that randomly samples program schedules from the space to provide comprehensive coverage of the search space.

Performance tuner. The performance of randomly sampled program schedules is not necessarily good. To fine-tune it, Ansor uses re-sampled new program schedules as well as good program schedules from previous iterations as the initial population to start the evolutionary search. Querying the learned cost model is orders of magnitude faster than actual measurement, so we can evaluate thousands of program schedules in seconds via the cost model.

Task scheduler. Program sampling and performance fine-tuning allow Ansor to find high-performance tensor program schedules for a computational graph. Although treating an entire DNN as a single computational graph and generating a full tensor program for it might result in the optimal schedule, this method is inefficient due to the unnecessary exponential explosion of the search space. Typically, the compiler partitions the large computational graph of a DNN into several small subgraphs [66, 177]. This partition has a negligible effect on the performance thanks to the layer-by-layer construction of DNNs. This brings the final challenge of Ansor: how to allocate time resources when generating program schedules for multiple subgraphs. The task scheduler in Ansor uses a scheduling algorithm based on gradient descent to allocate resources to the subgraphs that are more likely to improve the end-to-end DNN performance.

6.6 Related Work

Multiple previous attempts to automatically schedule Halide programs have been proposed. In the original paper [65], heuristics and genetic search over random schedule rewrites were used. The scheduling relied on measuring real execution time, which resulted in a search time that can take days for moderate benchmarks. OpenTuner [4] autotunes a program using an AUC-Bandit-meta-technique-directed ensemble selection of algorithms. It is effective in scheduling simple pipelines [9].

The auto-scheduler that comes with the master Halide repo, which we compared against in Section 6.4, is based on [10]. It uses a cost model with a greedy search algorithm that allows it to run quickly, with no autotuning or benchmarking at all. However, it only considers a fixed set of optimization heuristics for things like parallelism, vectorization and unrolling, and it has a single level of tiling and fusion. This cost model was improved in [179], but the the restricted search space was not expanded. In [11, 180] the search space was improved and a manual cost model was used. However, the search space is still smaller than ours.

Many compilers use loop polyhedral analysis to perform automatic scheduling of affine loop nests [9, 15, 181–184]. Many possible Halide schedules are excluded in these compilers. However, it might be possible to use the polyhedral representation to build more accurate cost models.

AutoTVM [185] uses tree-based algorithms to auto-schedule programs on TVM [66], which is an optimization stack for deep learning. This approach, however, still requires the user to manually write the search space for each loop. Furthermore, each operation is optimized in isolation without exploring large programs. In [186], deep reinforcement learning is used to schedule deep learning pipelines. This method results in better performance than AutoTVM.

Machine learning in compiler optimization has been proposed in many prior works [16, 17]. This includes phase ordering [6–8, 143, 144, 187], tiling factors [188], mappings of kernels to CPUs or GPUs [51] with supervised learning, auto-vectorization [14, 18, 49, 154, 165–170, 189], and the throughput of basic blocks [190].

Multi-Level Intermediate Representation (MLIR) [191] has been recently proposed to help scale performance with the end of Moore's law. One objective of MLIR is to represent kernels in a form suitable for optimization, and allow easy integration of search algorithms such as reinforcement learning, MCTS, and beam search.

6.7 Conclusion and Future Directions

The ProTuner framework that we proposed and developed in this chapter uses an MCTS-based algorithm to automatically tune programs for high-performance deep learning and image processing applications. MCTS looks ahead, evaluates complete programs, and does not evaluate incomplete ones or suffer from greediness. This was reflected in the results, which demonstrated up to 3.25× better performance compared to the state-of-the-art beam-search algorithm. Looking forward, we foresee a potential opportunity to continue scaling performance—despite the end of Moore's Law—through automatic program tuning and optimization, with machine learning algorithms such as MCTS.

We see multiple future directions for this work. We could combine a value function cost model that can predict the advantage of taking an action instead of running the MCTS simulation. Another direction can include applying deep reinforcement learning methods to solve the scheduling MDP as done in similar domains in compiler optimization in [13, 18, 28, 186]. We believe that if deep reinforcement learning (with a neural network) can generalize in that case, then the runtime of the algorithm can be significantly improved because it will only need to run inference rather than retrain/research from scratch as in MCTS or beam search. With limited resources, more accurate cost models are necessary for scheduling, especially with the recent trends in customized hardware and the explosion of new applications. Such cost models need to consider the target hardware parameters and program features.

Another point for improvement is in the generation of the next states during the random simulation. During the MCTS simulation phase, our implementation now generates all the possible children (next possible intermediate schedules) and then randomly picks one child.

In this setting, our algorithm's cost evaluation overhead is 7.5% while 88% of the time is spent on enumerating children that our standard MCTSes do not use. In other words, we see the potential for 8× runtime improvement for our MCTS algorithm. Different configurations of the MCTS impact the performance differently. Some applications seem to benefit more from greedy behavior while others work better with random behavior. The C_p could be further tuned and our performance could be improved if we had a different C_p for different programs. Because different programs have different costs/runtimes, using the same C_p for of them encourages less exploration in shorter programs. Furthermore, the C_p could be reduced to encourage less exploration as the standard deviation in the costs of the children decreases. We also observed that our MCTS can overfit to the cost model if we run it for too long.

The search space can also be extended to include more optimizations such as vector sizes different than the native vector size, or not forcing the multi-core parallelism to be at the outermost loop level. More hardware targets and autotuning methods can be explored to further improve the performance.

Chapter 7

Conclusion and Future Directions

This thesis explored the application of recent machine learning techniques in compiler optimization. Multiple compiler optimization challenges were tackled using deep RL, MCTS, and evolutionary algorithms. Multiple systems were built and open sourced to tackle these challenges. AutoPhase uses deep RL to generate better compiler phase orderings, a problem known to be NP-hard. NeuroVectorizer uses deep RL in a contextual bandit scheme for auto-vectorization. To generate better program schedules, ProTuner uses MCTS and Ansor uses evolutionary algorithms. RLDRM uses deep RL to optimize the cache allocated resources for different applications.

Unlike prior state-of-the-art methods, our approaches do not rely on hand-engineered features, greedy algorithms, or hand-written optimization, and can find better solutions in less time. Our deep RL methods can also generalize to new unseen programs and outperform prior approaches. Looking forward, we foresee an opportunity to continue scaling performance, despite the end of Moore's Law, and keep up with the continuously changing hardware through automatic program optimization. Deep RL algorithms that combine features from the underlying hardware and programs have the potential to generalize not only to new unseen programs but also to new unseen hardware, thus allowing us to build more intelligent compilers.

There are a number of future directions where this work can evolve. Some of them are outlined below.

Cost Model Improvements

Cost models are the main bottleneck to achieving optimal program performance with machine learning. The following approaches can be used to improve them.

- Better management of overhead tradeoffs: some use cases might benefit from faster search time over negligible improvement in performance while others might require the best performance while tolerating longer search time. Hybrid heterogeneous models that take into account both the performance and the search time can help address this challenge.

- Developing ways to fully represent different programs by mapping them to different points in the latent multidimensional space and generalizing to new unseen programs.
- Fully representing hardware in a meaningful way that enables generalization over different hardware targets.
- Combining different objectives such as code size, compilation time, search time, and performance.

Combining deep RL with such cost models will allow us to compensate for the end of Moore's law and the continuously changing hardware, saving the time and burden of manual optimization by software engineers.

Extending the Action Space

- **Hardware-level actions**. Hardware generators [192, 193] now enable the co-optimization of hardware and software by by means of actions that not only modify the program code but also modify the underlying hardware. This can be the ultimate end-to-end method for optimal performance. For example, Gemmini [193] is a systolic-array based matrix multiplication accelerator generator with many configurable architectural parameters such as local memory size, systolic array dimensions, pipeline depth, host CPU, dataflow and precision. These parameters can be tuned simultaneously with the software compiler optimizations to generate optimized systolic-array based matrix multiplication accelerators and compiled software.
- **Software-level actions**. Improved software action spaces (compiler optimizations) can be realized with better, design specific languages. Halide and TVM generate programs/schedules from simpler higher level code that focuses more on the functionality than on the low level implementation. This focus allows the compiler to generate better-optimized codes, which is much easier to do when the code itself is generated by the compiler. Optimizing existing low level code is more difficult than optimizing generated code. For example, it is easier to optimize programs written in Halide than in C++, as Halide facilitates rewriting the generated C++. Rewriting the code in C++ itself is a considerably more difficult task.

Supporting Distributed Applications

Another important future research direction is the distributed setting. With the end of Moore's law, the world is shifting to distributed computing, where machine learning will inevitably be the technology of choice for optimizing workloads on multiple nodes. This is a distinctly different task than optimizing for a single node, one that brings its own set of unique challenges. For example, how should the data be partitioned? How should the tasks be scheduled to achieve optimal performance? How should dependencies be handled? How can utilization be improved? How can the monetary cost of resources be minimized? Further exploration will be required to address performance optimization in the distributed setting.

Engaging with the Vendor Communities

More engagement from the research and vendor communities is necessary to fully realize machine learning in compiler optimization. Generalizable models cannot be realized without sufficient data to train them on. This data is composed of programs and benchmarks that can be open sourced by these communities and reused for the benefit of all the users. Vendors are often reluctant to adopt machine learning in compilers due to its added complexity, its unreliability, and its lack of explainability. Therefore, researchers need to work with the compiler engineers in tech companies to help bridge these gaps to facilitate the deployment of machine learning in commercial compilers.

Bibliography

[1] Spyridon Triantafyllis et al. “Compiler optimization-space exploration”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society. 2003, pp. 204–215.

[2] Lelac Almagor et al. “Finding effective compilation sequences”. In: *ACM SIGPLAN Notices*. Vol. 39. 7. ACM, 2004, pp. 231–239.

[3] Zhelong Pan and Rudolf Eigenmann. “Fast and effective orchestration of compiler optimizations for automatic performance tuning”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2006, pp. 319–332.

[4] Jason Ansel et al. “OpenTuner: An extensible framework for program autotuning”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM. 2014, pp. 303–316.

[5] Grigori Fursin et al. “Milepost GCC: Machine learning enabled self-tuning compiler”. In: *International Journal of Parallel Programming*. Vol. 39. 3. Springer, 2011, pp. 296–327.

[6] Mark Stephenson et al. “Meta Optimization: Improving Compiler Heuristics with Machine Learning”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI ’03. 2003.

[7] Sameer Kulkarni and John Cavazos. “Mitigating the compiler optimization phase-ordering problem using machine learning”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’12. 2012.

[8] Felix Agakov et al. “Using machine learning to focus iterative optimization”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2006, pp. 295–305.

[9] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. “Polymage: Automatic optimization for image processing pipelines”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 1. ACM New York, NY, USA, 2015, pp. 429–443.

[10] Ravi Teja Mullapudi et al. “Automatically scheduling Halide image processing pipelines”. In: *ACM Transactions on Graphics (TOG)*. Vol. 35. 4. ACM New York, NY, USA, 2016, pp. 1–11.

[11] Savvas Sioutas et al. “Schedule synthesis for Halide pipelines through reuse analysis”. In: *ACM Transactions on Architecture and Code Optimization (TACO)*. Vol. 16. 2. ACM New York, NY, USA, 2019, pp. 1–22.

[12] Andrew Adams et al. “Learning to optimize Halide with tree search and random programs”. In: *ACM Transactions on Graphics (TOG)*. Vol. 38. 4. ACM New York, NY, USA, 2019, pp. 1–12.

[13] Ameer Haj-Ali et al. “AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning”. In: *Third Conference on Machine Learning and Systems (MLSys 2020)*. 2020.

[14] Xinmin Tian et al. “LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading”. In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE. 2016, pp. 21–31.

[15] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. “Polly: Performing polyhedral optimizations on a low-level intermediate representation”. In: *Parallel Processing Letters*. Vol. 22. 04. World Scientific, 2012.

[16] Amir H Ashouri et al. “A survey on compiler autotuning using machine learning”. In: *ACM Computing Surveys (CSUR)*. Vol. 51. 5. ACM, 2018, pp. 1–42.

[17] Zheng Wang and Michael O’Boyle. “Machine learning in compiler optimization”. In: *Proceedings of the IEEE*. Vol. 106. 11. 2018, pp. 1879–1901.

[18] Ameer Haj-Ali et al. “NeuroVectorizer: end-to-end vectorization with deep reinforcement learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 242–255.

[19] Richard Bellman. “A Markovian decision process”. In: *Journal of Mathematics and Mechanics*. Vol. 6. 5. 1957, pp. 679–684.

[20] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: a survey”. In: *Journal of Artificial Intelligence Research*. Vol. 4. 1996, pp. 237–285.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: an introduction*. MIT Press, 2018.

[22] Volodymyr Mnih et al. “Playing Atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602*. 2013.

[23] Kenji Doya. “Reinforcement learning in continuous time and space”. In: *Neural Computation*. Vol. 12. 1. MIT Press, 2000, pp. 219–245.

[24] Jens Kober, J Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research*. Vol. 32. 11. Sage Publications UK: London, England, 2013, pp. 1238–1274.

[25] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. "Reinforcement learning for humanoid robotics". In: *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots*. 2003, pp. 1–20.

[26] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature*. Vol. 529. 7587. Nature Publishing Group, 2016, pp. 484–489.

[27] Ameer Haj-Ali et al. "A view on deep reinforcement learning in system optimization". In: *arXiv preprint arXiv:1908.01275*. 2019.

[28] Qijing Huang et al. "AutoPhase: Compiler phase-ordering for HLS with deep reinforcement learning". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[29] Ameer Haj-Ali et al. "Learning to vectorize using deep reinforcement learning". In: *Workshop on ML for Systems at NeurIPS*. 2019.

[30] Bin Li et al. "RLDRM: closed loop dynamic cache allocation with deep reinforcement learning for network function virtualization". In: *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 335–343.

[31] Ameer Haj-Ali et al. "ProTuner: tuning programs with Monte Carlo tree search". In: *arXiv preprint arXiv:2005.13685*. 2020.

[32] Lianmin Zheng et al. "Ansor: generating high-performance tensor programs for deep learning". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 2020.

[33] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[34] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 627–635.

[35] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems*. 2000, pp. 1057–1063.

[36] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*. 2017.

[37] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971*. 2015.

[38] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine Learning*. Vol. 8. 3-4. Springer, 1992, pp. 279–292.

[39] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Technical Report 166. University of Cambridge, Department of Engineering. Cambridge, England, 1994.

[40] Peter Auer. "Using confidence bounds for exploitation-exploration trade-offs". In: *Journal of Machine Learning Research*. Vol. 3. 2002, pp. 397–422.

[41] Donald A Berry and Bert Fristedt. "Bandit problems: sequential allocation of experiments". In: *Monographs on Statistics and Applied Probability*. Vol. 5. London: Chapman and Hall, 1985, pp. 71–87.

[42] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine Learning*. Vol. 47. 2-3. Springer, 2002, pp. 235–256.

[43] Wei Chu et al. "Contextual bandits with linear payoff functions". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 208–214.

[44] Tomas Jakl. *Arimaa challenge-comparison study of MCTS versus alpha-beta methods*. Univerzita Karlova, Matematicko-fyzikalni fakulta, 2011.

[45] David E Goldberg. *Genetic algorithms*. Pearson Education India, 2006.

[46] Edoardo Conti et al. "Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents". In: *Advances in Neural Information Processing Systems*. 2018, pp. 5032–5043.

[47] Tim Salimans et al. "Evolution strategies as a scalable alternative to reinforcement learning". In: *arXiv preprint arXiv:1703.03864*. 2017.

[48] D Raj Reddy et al. "Speech understanding systems: a summary of results of the five-year research effort". In: *Department of Computer Science. Carnegie-Mellon University, Pittsburgh, PA*. Vol. 17. 1977.

[49] Kevin Stock, Louis-Noël Pouchet, and P Sadayappan. "Using machine learning to improve automatic vectorization". In: *ACM Transactions on Architecture and Code Optimization (TACO)*. Vol. 8. 4. ACM, 2012, p. 50.

[50] Chris Cummins et al. "Synthesizing benchmarks for predictive modeling". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[51] Chris Cummins et al. "End-to-end deep learning of optimization heuristics". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 219–232.

[52] S. Cho and L. Jin. "Managing distributed, shared L2 caches through OS-level page allocation". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006, pp. 455–468.

[53] M. K. Qureshi and Y. N. Patt. "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. Dec. 2006, pp. 423–432.

[54] Ravi Iyer. "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms". In: *Proceedings of the 18th Annual International Conference on Supercomputing*. ICS '04. New York, NY, USA: ACM, 2004, pp. 257–266. ISBN: 1-58113-839-3.

[55] Ravi Iyer et al. "QoS Policies and Architecture for Cache/Memory in CMP Platforms". In: *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '07. New York, NY, USA: ACM, 2007, pp. 25–36.

[56] Bin Li et al. "CoQoS: coordinating QoS-aware shared resources in NoC-based SoCs". In: *Journal of Parallel and Distributed Computing*. Vol. 71. 5. Orlando, FL, USA, May 2011, pp. 700–713.

[57] Bin Li et al. "Dynamic QoS Management for Chip Multiprocessors". In: *ACM Transactions on Architecture and Code Optimization*. Vol. 9. 3. New York, NY, USA: ACM, Oct. 2012, 17:1–17:29.

[58] Andrew Herdrich et al. "Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family". In: *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA 2016)*. Barcelona, Spain, 2016.

[59] V. Selfa et al. "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017, pp. 194–205.

[60] Cong Xu et al. "dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: ACM, 2018, 14:1–14:13.

[61] Harshad Kasture and Daniel Sanchez. "Ubik: efficient cache sharing with strict Qos for latency-critical workloads". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 729–742.

[62] D. Sanchez and C. Kozyrakis. "Vantage: Scalable and efficient fine-grain cache partitioning". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 57–68.

[63] Henry Cook et al. "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness". In: *SIGARCH Computer Architecture News*. Vol. 41. 3. New York, NY, USA: ACM, June 2013, pp. 308–319.

[64] *Intel 64 and IA-32 Architectures Software Developer's Manual.*

[65] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM Sigplan Notices*. Vol. 48. 6. ACM New York, NY, USA, 2013, pp. 519–530.

[66] Tianqi Chen et al. “TVM: end-to-end optimization stack for deep learning”. In: *arXiv preprint arXiv:1802.04799*. 2018.

[67] Abdul Hameed et al. “A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems”. In: *Computing*. Vol. 98. 7. Springer, 2016, pp. 751–774.

[68] Mohammad Saeid Mahdavinejad et al. “Machine learning for Internet of Things data analysis: A survey”. In: *Digital Communications and Networks*. Vol. 4. 3. Elsevier, 2018, pp. 161–175.

[69] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. “A taxonomy and survey of grid resource management systems for distributed computing”. In: *Software: Practice and Experience*. Vol. 32. 2. Wiley Online Library, 2002, pp. 135–164.

[70] Nguyen Cong Luong et al. “Applications of deep reinforcement learning in communications and networking: A survey”. In: *IEEE Communications Surveys & Tutorials*. IEEE, 2019.

[71] Anup Das et al. “Reinforcement learning-based inter-and intra-application thermal optimization for lifetime improvement of multicore systems”. In: *Proceedings of the 51st Annual Design Automation Conference*. ACM. 2014, pp. 1–6.

[72] Qijun Zhu and Chun Yuan. “A reinforcement learning approach to automatic error recovery”. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*. IEEE. 2007, pp. 729–738.

[73] Johannes Zeppenfeld et al. “Learning classifier tables for autonomic systems on chip”. In: *GI Jahrestagung (2)*. Vol. 134. 2008, pp. 771–778.

[74] Engin Ipek et al. “Self-optimizing memory controllers: A reinforcement learning approach”. In: *ACM SIGARCH Computer Architecture News*. Vol. 36. 3. IEEE Computer Society. 2008, pp. 39–50.

[75] Eva Andreasson, Frank Hoffmann, and Olof Lindholm. “To collect or not to collect? Machine learning for memory management.” In: *Java Virtual Machine Research and Technology Symposium*. 2002, pp. 27–39.

[76] Leeor Peled et al. “Semantic locality and context-based prefetching using reinforcement learning”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2015, pp. 285–297.

[77] Nuno Diegues and Paolo Romano. “Self-tuning Intel transactional synchronization extensions”. In: *11th International Conference on Autonomic Computing ({ICAC} 14)*. 2014, pp. 209–219.

[78] Wei Li et al. “Learning-based and data-driven TCP design for memory-constrained iot”. In: *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE. 2016, pp. 199–205.

[79] Aloizio P Silva et al. “Smart congestion control for delay-and disruption tolerant networks”. In: *2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE. 2016, pp. 1–9.

[80] Samuel PM Choi and Dit-Yan Yeung. “Predictive Q-routing: A memory-based reinforcement learning approach to adaptive traffic control”. In: *Advances in Neural Information Processing Systems*. 1996, pp. 945–951.

[81] Michael Littman and Justin Boyan. “A distributed reinforcement learning scheme for network routing”. In: *Proceedings of the International Workshop on Applications of Neural Networks to Telecommunications*. Psychology Press. 2013, pp. 55–61.

[82] Justin A Boyan and Michael L Littman. “Packet routing in dynamically changing networks: A reinforcement learning approach”. In: *Advances in Neural Information Processing Systems*. 1994, pp. 671–678.

[83] Michail G Lagoudakis and Michael L Littman. “Algorithm selection using reinforcement learning”. In: *International Conference on Machine Learning (ICML 2000)*. 2000, pp. 511–518.

[84] Alireza Sadeghi, Fatemeh Sheikholeslami, and Georgios B Giannakis. “Optimal and scalable caching for 5G using reinforcement learning of space-time popularities”. In: *IEEE Journal of Selected Topics in Signal Processing*. Vol. 12. 1. IEEE, 2017, pp. 180–190.

[85] Fahimeh Farahnakian, Pasi Liljeberg, and Juha Plosila. “Energy-efficient virtual machines consolidation in cloud data centers using reinforcement learning”. In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2014, pp. 500–507.

[86] Zhiping Peng et al. “Random task scheduling scheme based on reinforcement learning in cloud computing”. In: *Cluster Computing*. Vol. 18. 4. Springer, 2015, pp. 1595–1607.

[87] Pooyan Jamshidi et al. “Self-learning cloud controllers: Fuzzy Q-learning for knowledge evolution”. In: *2015 International Conference on Cloud and Autonomic Computing*. IEEE. 2015, pp. 208–211.

[88] Enda Barrett, Enda Howley, and Jim Duggan. “Applying reinforcement learning towards automating resource allocation and application scalability in the cloud”. In: *Concurrency and Computation: Practice and Experience*. Vol. 25. 12. Wiley Online Library, 2013, pp. 1656–1674.

[89] Hamid Arabnejad et al. “A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling”. In: *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press. 2017, pp. 64–73.

[90] Seyedakbar Mostafavi, Fatemeh Ahmadi, and Mehdi Agha Sarram. "Reinforcement-learning-based foresighted task scheduling in cloud computing". In: *arXiv preprint arXiv:1810.04718*. 2018.

[91] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Ph.D. thesis. Carnegie-Mellon University School of Computer Science, 1993.

[92] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM. 2016, pp. 50–56.

[93] Ying He et al. "Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach". In: *IEEE Communications Magazine*. Vol. 55. 12. IEEE, 2017, pp. 31–37.

[94] Ying He, Nan Zhao, and Hongxi Yin. "Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach". In: *IEEE Transactions on Vehicular Technology*. Vol. 67. 1. IEEE, 2017, pp. 44–55.

[95] Gerald Tesauro, Rajarshi Das, and Nicholas K Jong. "Online performance management using hybrid reinforcement learning". In: *Proceedings of SysML*. 2006.

[96] Zhiyuan Xu et al. "A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs". In: *2017 IEEE International Conference on Communications (ICC)*. IEEE. 2017, pp. 1–6.

[97] Ning Liu et al. "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 372–382.

[98] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. "URL: A unified reinforcement learning approach for autonomic cloud management". In: *Journal of Parallel and Distributed Computing*. Vol. 72. 2. Elsevier, 2012, pp. 95–105.

[99] Jia Rao et al. "VCONF: a reinforcement learning approach to virtual machines auto-configuration". In: *Proceedings of the 6th International Conference on Autonomic Computing*. ACM. 2009, pp. 137–146.

[100] Chi Jin et al. "Is q-learning provably efficient?" In: *Advances in Neural Information Processing Systems*. 2018, pp. 4863–4873.

[101] Nathan Jay et al. "A deep reinforcement learning perspective on internet congestion control". In: *International Conference on Machine Learning*. 2019, pp. 3050–3059.

[102] Fabian Ruffy, Michael Przystupa, and Ivan Beschastnikh. "Iroko: a framework to prototype reinforcement learning for data center traffic control". In: *arXiv preprint arXiv:1812.09975*. 2018.

[103] Eric Liang et al. "Neural packet classification". In: *arXiv preprint arXiv:1902.10319*. 2019.

[104] Victor Zhong, Caiming Xiong, and Richard Socher. "Seq2sql: Generating structured queries from natural language using reinforcement learning". In: *arXiv preprint arXiv:1709.00103*. 2017.

[105] Kelvin Guu et al. "From language to programs: Bridging reinforcement learning and maximum marginal likelihood". In: *arXiv preprint arXiv:1704.07926*. 2017.

[106] Chen Liang et al. "Neural symbolic machines: Learning semantic parsers on freebase with weak supervision". In: *arXiv preprint arXiv:1611.00020*. 2016.

[107] Sanjay Krishnan et al. "Learning to optimize join queries with deep reinforcement learning". In: *arXiv preprint arXiv:1808.03196*. 2018.

[108] Jennifer Ortiz et al. "Learning state representations for query optimization with deep reinforcement learning". In: *arXiv preprint arXiv:1803.08604*. 2018.

[109] Ryan Marcus and Olga Papaemmanouil. "Deep reinforcement learning for join order enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM. 2018, pp. 1–4.

[110] Ryan Marcus et al. "Neo: A learned query optimizer". In: *arXiv preprint arXiv:1904.03711*. 2019.

[111] Sameer Kulkarni and John Cavazos. "Mitigating the compiler optimization phase-ordering problem using machine learning". In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM. 2012, pp. 147–162.

[112] Ravichandra Addanki et al. "Placeto: learning generalizable device placement algorithms for distributed machine learning". In: *arXiv preprint arXiv:1906.08879*. 2019.

[113] Aditya Paliwal et al. "REGAL: transfer learning for fast optimization of computation graphs". In: *arXiv preprint arXiv:1905.02494*. 2019.

[114] Katherine E Coons et al. "Feature selection and policy optimization for distributed instruction placement using reinforcement learning". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM. 2008, pp. 32–42.

[115] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 6645–6649.

[116] Stefan Schaal. "Learning from demonstration". In: *Advances in Neural Information Processing Systems*. 1997, pp. 1040–1046.

[117] Todd Hester et al. "Deep q-learning from demonstrations". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[118] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: `1606.01540 [cs.LG]`.

[119] Hongzi Mao et al. "Park: An open platform for learning-augmented computer systems". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019, pp. 2494–2506.

[120] Stefan Schaal. "Is imitation learning the route to humanoid robots?" In: *Trends in Cognitive Sciences*. Vol. 3. 6. Elsevier, 1999, pp. 233–242.

[121] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290*. 2018.

[122] Eric Liang et al. "Ray rllib: A composable and scalable reinforcement learning library". In: *arXiv preprint arXiv:1712.09381*. 2017.

[123] Itai Caspi et al. *Reinforcement Learning Coach*. 2017. URL: `https://doi.org/10.5281/zenodo.1134899`.

[124] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. *Tensorforce: a TensorFlow library for applied reinforcement learning*. Web page. 2017. URL: `https://github.com/tensorforce/tensorforce`.

[125] Jason Gauci et al. "Horizon: Facebook's open source applied reinforcement learning platform". In: *arXiv preprint arXiv:1811.00260*. 2018.

[126] Pablo Samuel Castro et al. "Dopamine: a research framework for deep reinforcement learning". In: *arXiv preprint arXiv:1812.06110*. 2018.

[127] Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. "Learning to control a low-cost manipulator using data-efficient reinforcement learning". In: *Robotics: Science and Systems V*. 2011, pp. 57–64.

[128] Xiaoxiao Guo et al. "Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning". In: *Advances in Neural Information Processing Systems*. 2014, pp. 3338–3346.

[129] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[130] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. "Variance reduction techniques for gradient estimates in reinforcement learning". In: *Journal of Machine Learning Research*. Vol. 5. 2004, pp. 1471–1530.

[131] Ben Pfaff et al. "The design and implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA, May 2015, pp. 117–130. ISBN: 978-1-931971-218.

[132] Y. Wang et al. "Optimizing Open vSwitch to support millions of flows". In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. Dec. 2017, pp. 1–7.

[133] Intel Corporation. *Data Plane Development Kit (DPDK)*. `https://www.dpdk.org`. 2018.

[134] Yifan Yuan et al. "HALO: accelerating flow classification for scalable packet processing in NFV". In: *Proceedings of the 46th International Symposium on Computer Architecture*. ISCA '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 601–614. ISBN: 9781450366694.

[135] Intel Corporation. *Intel® Data Direct I/O (DDIO).* `https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`. 2018.

[136] Hado van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-Learning". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 2094–2100.

[137] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning". In: *Proceedings of the 33rd International Conference on Machine Learning*. ICML'16. 2016, pp. 1995–2003.

[138] Xilinx. *Vivado High-Level Synthesis*. 2019. URL: `%7Bhttps://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html%7D`.

[139] Intel. *Intel High-Level Synthesis Compiler*. 2019. URL: `%7Bhttps://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html%7D`.

[140] Andrew Canis et al. "LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems". In: *ACM Transactions on Embedded Computing Systems (TECS)*. Vol. 13. 2. ACM, 2013. DOI: `10.1145/2514740`.

[141] Chris Lattner and Vikram 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.

[142] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[143] Qijing Huang et al. "The effect of compiler optimizations on high-level synthesis for FPGAs". In: *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2013)*. IEEE. 2013, pp. 89–96.

[144] Qijing Huang et al. "The effect of compiler optimizations on high-level synthesis-generated hardware". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. Vol. 8. 3. ACM, 2015. DOI: `10.1145/2629547`.

[145] Yuko Hara et al. "CHstone: A benchmark program suite for practical c-based high-level synthesis". In: *IEEE International Symposium on Circuits and Systems (ISCAS 2008)*. 2008, pp. 1192–1195.

[146] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *ACM SIGPLAN Notices*. Vol. 46. ACM. 2011, pp. 283–294.

[147] Leo Breiman. "Random forests". In: *Machine Learning*. Vol. 45. 1. Springer, 2001, pp. 5–32.

[148] Eric Liang et al. "RLlib: abstractions for distributed reinforcement learning". In: *arXiv preprint arXiv:1712.09381*. 2017.

[149] Philipp Moritz et al. “Ray: a distributed framework for emerging AI applications”. In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[150] Félix-Antoine Fortin et al. “DEAP: Evolutionary algorithms made easy”. In: *Journal of Machine Learning Research*. Vol. 13. July 2012, pp. 2171–2175.

[151] James Kennedy. “Particle swarm optimization”. In: *Encyclopedia of Machine Learning*. Springer, 2010, pp. 760–766.

[152] Chris Lomont. “Introduction to Intel advanced vector extensions”. In: *Intel White Paper*. 2011, pp. 1–21.

[153] Dorit Nuzman, Ira Rosen, and Ayal Zaks. “Auto-vectorization of interleaved data for SIMD”. In: *ACM SIGPLAN Notices*. Vol. 41. 6. ACM, 2006, pp. 132–143.

[154] Konrad Trifunovic et al. “Polyhedral-model guided loop-nest auto-vectorization”. In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2009, pp. 327–337.

[155] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. “Nearest neighbor queries”. In: *ACM Sigmod Record*. Vol. 24. 2. ACM. 1995, pp. 71–79.

[156] J. Ross Quinlan. “Induction of decision trees”. In: *Machine Learning*. Vol. 1. 1. Springer, 1986, pp. 81–106.

[157] Intel Inc. *Intel Core i7-8559U Processor Specification*. 2018. URL: `https://ark.intel.com/content/www/us/en/ark/products/137979/intel-core-i7-8559u-processor-8m-%5C%20cache-up-to-4-50-ghz.html`.

[158] Uri Alon et al. “code2vec: learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages*. Vol. 3. POPL. ACM, 2019, pp. 1–29.

[159] Ronan Collobert and Jason Weston. “A unified architecture for natural language processing: Deep neural networks with multitask learning”. In: *Proceedings of the 25th International Conference on Machine Learning*. ACM. 2008, pp. 160–167.

[160] Kelvin Xu et al. “Show, attend and tell: neural image caption generation with visual attention”. In: *International Conference on Machine Learning*. 2015, pp. 2048–2057.

[161] Richard Liaw et al. “Tune: A research platform for distributed model selection and training”. In: *arXiv preprint arXiv:1807.05118*. 2018.

[162] Matthew R Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14.

[163] Louis-Noel Pouchet. “Polybench: The polyhedral benchmark suite”. In: *URL: http://www. cs. ucla. edu/pouchet/software/polybench*. 2012.

[164] David Gunning. “Explainable artificial intelligence (XAI)”. In: *Defense Advanced Research Projects Agency (DARPA)*. 2017.

[165] Dorit Nuzman et al. “Vapor SIMD: Auto-vectorize once, run everywhere”. In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 151–160.

[166] Vasileios Porpodas and Timothy M Jones. “Throttling automatic vectorization: When less is more”. In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 432–444.

[167] Samuel Larsen and Saman Amarasinghe. “Exploiting superword level parallelism with multimedia instruction sets”. In: *ACM Sigplan Notices*. Vol. 35. 5. 2000.

[168] Daniel S McFarlin et al. “Automatic SIMD vectorization of fast Fourier transforms for the larrabee and AVX instruction sets”. In: *Proceedings of the International Conference on Supercomputing*. ACM. 2011, pp. 265–274.

[169] Vasileios Porpodas. “SuperGraph-SLP Auto-Vectorization”. In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 330–342.

[170] Vasileios Porpodas, Alberto Magni, and Timothy M Jones. “PSLP: Padded SLP automatic vectorization”. In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 190–201.

[171] Cameron B Browne et al. “A survey of Monte Carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games*. Vol. 4. 1. IEEE, 2012, pp. 1–43.

[172] Yngvi Bjornsson and Hilmar Finnsson. “Cadiaplayer: a simulation-based general game player”. In: *IEEE Transactions on Computational Intelligence and AI in Games*. Vol. 1. 1. IEEE, 2009, pp. 4–15.

[173] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. “Parallel Monte-Carlo tree search”. In: *International Conference on Computers and Games*. Springer. 2008, pp. 60–71.

[174] Levente Kocsis, Csaba Szepesvari, and Jan Willemson. “Improved Monte-Carlo search”. In: *University of Tartu, Estonia, Technical Report*. Vol. 1. 2006.

[175] Steven Johnson. “Running and Benchmarking Halide Generators”. In: *Halide GitHub Repository: https://github.com/halide/Halide/blob/master/README_rungen.md*. Aug. 2019.

[176] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 770–778.

[177] Jared Roesch et al. “Relay: A high-level IR for deep learning”. In: *arXiv preprint arXiv:1904.08368*. 2019.

[178] Junjie Bai, Fang Lu, Ke Zhang, et al. *ONNX: Open Neural Network Exchange*. 2019.

[179] Abhinav Jangda and Uday Bondhugula. "An effective fusion and tile size model for optimizing image processing pipelines". In: *ACM SIGPLAN Notices*. Vol. 53. 1. ACM New York, NY, USA, 2018, pp. 261–275.

[180] Savvas Sioutas et al. "Loop transformations leveraging hardware prefetching". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 254–264.

[181] Nicolas Vasilache et al. "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions". In: *arXiv preprint arXiv:1802.04730*. 2018.

[182] Riyadh Baghdadi et al. "Pencil: A platform-neutral compute intermediate language for accelerator programming". In: *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE. 2015, pp. 138–149.

[183] Uday Bondhugula et al. "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2008, pp. 101–113.

[184] Riyadh Baghdadi et al. "Tiramisu: A polyhedral compiler for expressing fast and portable code". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 193–205.

[185] Tianqi Chen et al. "Learning to optimize tensor programs". In: *Advances in Neural Information Processing Systems*. 2018, pp. 3389–3400.

[186] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. "Reinforcement learning and adaptive sampling for optimized DNN compilation". In: *arXiv preprint arXiv:1905.12799*. 2019.

[187] Grigori Fursin et al. "MILEPOST GCC: machine learning based research compiler". In: *GCC Summit*. 2008.

[188] Mohammed Rahman, Louis-Noël Pouchet, and P Sadayappan. "Neural network assisted tile size selection". In: *International Workshop on Automatic Performance Tuning (IWAPT'2010)*. 2010.

[189] Charith Mendis et al. "Compiler auto-vectorization with imitation learning". In: *Advances in Neural Information Processing Systems*. 2019, pp. 14598–14609.

[190] Charith Mendis et al. "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks". In: *arXiv preprint arXiv:1808.07412*. 2018.

[191] Chris Lattner et al. "MLIR: a compiler infrastructure for the end of Moore's law". In: *arXiv preprint arXiv:2002.11054*. 2020.

[192] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro*. Vol. 40. 4. 2020, pp. 10–21. DOI: `10.1109/MM.2020.2996616`.

[193] Hasan Genc et al. "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures". In: *arXiv preprint arXiv:1911.09925*. 2019.