Automating Reinforcement Learning Architecture Design for Code Optimization

Huanting Wang
1. Northwest University, China
2. University of Leeds, U. K.

Zhanyong Tang
Northwest University, China

Cheng Zhang
Northwest University, China

Jiaqi Zhao
Northwest University, China

Chris Cummins
Meta AI Research, USA

Hugh Leather
Meta AI Research, USA

Zheng Wang
University of Leeds, U. K.

Abstract

Reinforcement learning (RL) is emerging as a powerful technique for solving complex code optimization tasks with an ample search space. While promising, existing solutions require a painstaking manual process to tune the right task-specific RL architecture, for which compiler developers need to determine the composition of the RL exploration algorithm, its supporting components like state, reward and transition functions, and the hyperparameters of these models. This paper introduces SuperSonic, a new open-source framework to allow compiler developers to integrate RL into compilers easily, regardless of their RL expertise. SuperSonic supports customizable RL architecture compositions to target a wide range of optimization tasks. A key feature of SuperSonic is the use of deep RL and multi-task learning techniques to develop a meta-optimizer to automatically find and tune the right RL architecture from training benchmarks. The tuned RL can then be deployed to optimize new programs. We demonstrate the efficacy and generality of SuperSonic by applying it to four code optimization problems and comparing it against eight auto-tuning frameworks. Experimental results show that SuperSonic consistently improves hand-tuned methods by delivering better overall performance, accelerating the deployment-stage search by 1.75x on average (up to 100x).

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Artificial intelligence*.

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04…$15.00
https://doi.org/10.1145/3497776.3517769

Keywords: Compiler optimization, reinforcement learning, code optimization

ACM Reference Format:
Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3497776.3517769

1 Introduction

There is a growing interest in using auto-tuning techniques for code optimization [9, 18, 27, 32, 62, 70, 88]. Auto-tuning finds good optimizations from empirical observations, often outperforming hand-crafted heuristics [9, 52]. This technique is particularly attractive for frequently used libraries and kernels. For such scenarios, developers are willing to spend hours or weeks of CPU time to automatically search for even a few percentages of performance improvement [18, 23, 50, 83], knowing the optimized code will be used by many users.

In recent years, deep reinforcement learning (RL) has been shown to be effective for navigating a sizeable discrete space, outperforming traditional search techniques on a range of optimization tasks [33, 60, 79, 85, 87]. Deep RL is also a natural fit for many code optimization problems where the task can be seen as applying a sequence of actions to maximize the gain [88]. Examples of such problems include compiler flag selection and ordering [4, 25, 47], instruction scheduling [69, 80, 92] and hardware resource allocation [26]. Indeed, we see a growing interest in the research community and industry [23, 83] in using RL and deep RL to tackle a wide range of code optimization problems [13, 35, 36, 45, 53, 68].

Developing a deep RL solution for code optimization requires choosing and parameterizing several components: (i) a discrete set of actions or transformations that can be applied to a program, such as passes in a compiler; (ii) a state function that can summarize the program after each action as a finite feature vector, and (iii) a reward function that reports the quality of the actions taken so far. Some RL algorithms

may also allow the selection and further parameterization of a transition function, which governs the choice of actions to be applied in each state. Moreover, parameters of individual RL components need to be tuned on benchmarks.

Efforts have been made to provide RL algorithms and high-level APIs for action definitions [23, 51], models for program state representation [12, 21, 86, 90], and tools for training benchmark generation [22–24]. While these recent works have lowered the barrier for integrating RL techniques into compilers, compiler engineers still face a major hurdle. As the right combination of RL exploration algorithms and their state, reward and transition functions and parameters highly depend on the optimization task, developers must carefully choose the RL architecture by finding the right RL component composition and their parameters from a large pool of candidate RL algorithms, machine-learning models and functions. This process currently requires testing and manually analyzing a large combination of RL components. Experience in the field of neural architecture search shows that doing this by hand is an expensive and non-trivial process [28].

This paper presents SuperSonic[1], an open-source framework to automate the RL architecture search and parameter tuning process to make it easier to integrate RL into compilers. To use SuperSonic, the compiler developer provides the action list according to the problem being tackled and a measurement interface to report metrics like code size or speedup. SuperSonic then automatically assembles an RL architecture for the targeting optimization from an extensible set of built-in RL components. The SuperSonic RL components include pre-trained state functions, such as `Word2Vec` [57] and `CodeBert` [29]. It provides candidate reward functions like `RelativeMeasure` and `tanh` to compute the reward based on the metric given by the measurement interface. The state-transition function can be selected from a further set of predefined transition functions, such as a transition probability matrix or LSTM [39]. Finally, SuperSonic takes a customizable set of predefined RL algorithms, like Proximal Policy Optimization (PPO) [77] and Monte Carlo tree search (MCTS) [14], which may be driven by any of the chosen reward, state, actions and transition function. This creates a large space of possible parameterized RL architectures, which can be defined by the compiler developer with a few lines of Python using an easy-to-use API.

Armed with this space of RL architecture choices, SuperSonic will automatically and efficiently search it to find the one that gives the best results over a sample of user-provided *training* benchmarks. The search is accomplished by a deep RL-based *meta-optimizer* that is designed to be generalizable to any optimization tasks. Once the meta-optimizer has selected the *client RL* architecture and its parameters, the work of SuperSonic is over. As an output, SuperSonic stores the tuned client RL as serialized objects. It provides an API for a compiler or performance tuner to use the stored RL to drive the code transformation pass for *new, unseen* programs. On an unseen program, the client RL may be used to search for the actions that yield the best reward. Alternatively, it can be further generalized by training on additional benchmarks so that actions for new programs can be chosen *directly* from the policy without further search at the deployment time. SuperSonic aims to automatically make the client RL as good as possible for the compiler developer's needs.

As this client RL search and tuning process is automated and requires little RL expertise, SuperSonic further reduces the difficulties for integrating RL into compilers by replacing compiler developer time with machine hours. This client RL search and tuning process is a *one-off* cost performed offline. The compiler end-users (e.g., application developers) will not experience this process – they will use the shipped RL like any other feedback directed compiler passes [16].

We evaluate SuperSonic by applying it to four optimization tasks: Halide schedule optimization [69], neural network code generation [18], compiler phase ordering [25] for code size reduction and superoptimization [43]. Each of the tasks has a large number of combined optimization options, so it is non-trivial to design a good search strategy. We compare SuperSonic against eight tuning methods developed by independent researchers, including search-based strategies specifically designed for the targeting problem [3, 19, 62, 76], generic tuning frameworks like OpenTuner [9] and CompilerGym [23], and hand-tuned RL solutions for the relevant task [5, 68]. Our extensive evaluation shows that SuperSonic consistently gives better overall performance than alternative methods across tasks during deployment. We show that the client RL given by SuperSonic converges fast, and it can start producing better code over competing search methods by using on average 1.75x less search time (up to 100x) for new programs during deployment.

This paper makes the following contributions:

- We present a generic framework to automatically choose and tune a suitable RL architecture for code optimization tasks (Section 3).
- We demonstrate how deep RL can be used as a meta-optimizer to support the integration of RL into performance tuners (Section 3.3).
- We provide a large study validating the effectiveness of RL in four code optimization tasks (Section 5).

2 Background

2.1 Deep Reinforcement Learning

RL is a machine learning technique where agents learn to perform actions in an environment to maximize a cumulative reward [81]. The environment represents the problem to be solved, and the agent represents the learning algorithm. At each time step, the agent observes the environment state and takes an action (e.g., deciding which compiler option

[1] Available at: https://github.com/HuantWang/SUPERSONIC

to be added or removed from the compilation sequence). The agent adjusts its actions based on observations gathered from the environment. It learns a policy to map the states to the corresponding actions, aiming to maximize the expected reward. Some RL algorithms also learn a value function to evaluate the current situation of the environment and the decision-making process of the agent. Unlike a policy function that answers the question of "how to act?", the value function answers the question of "how good the current state is?". Using a deep neural network (DNN) along with RL is known as Deep Reinforcement Learning [37].

2.2 Problem Definitions

SuperSonic automates RL component searching and parameter tuning. Within SuperSonic, a *client RL* consists of an exploration algorithm to choose actions (e.g., compiler options), a reward function for computing the expected cumulative reward based on past observations of the environment (e.g., execution time after applying a transformation), a method for modeling the environment/program state (e.g., a DNN or a linear function), and an action list provided by the user (e.g., legitimate code transformation options). Depending on the exploration algorithm, this can also include an state transition function to compute the probability of going from one state to another. Each of the components can be chosen from a pool of SuperSonic built-in candidate methods, and the combination of these components can result in a large policy search space. SuperSonic is designed to automatically find and tune the right combination of RL components and their hyperparameters. It complements existing RL platforms like CompilerGym [23] and RLLib [51], by helping the compiler developers to choose and optimise a suitable RL algorithm.

2.3 Multi-armed Bandit Problem

The effectiveness of an RL algorithm is highly dependent on the problem to be solved, but manually choosing the right RL components is non-trivial [44]. In this work, we formulate RL component search as a multi-armed bandit (MAB) problem [30]. Given a budget of n trials and k candidate RL component configurations (or slot machines), we use a deep RL-based MAB solver to allocate the trials among candidate policy architectures to test their effectiveness. After n trials, we determine which of the k RL configurations to use for the given problem. This technique is detailed in Section 3.3.

3 Our Approach

3.1 Overview

Figure 1 gives an overview of SuperSonic. At the core of SuperSonic is a meta-optimizer that builds upon MAB and deep RL techniques. Given a client RL search space defined by the SuperSonic Python API, the meta-optimizer searches for a suitable RL component configuration for an optimization task. It then automatically tunes a set of tunable hyperparameters of the chosen components. The tuned client RL can then be used to optimize unseen programs through inference (including potential retraining), which is outside the scope of SuperSonic. The search space definition and RL client architecture search and parameter tuning are a *one-off offline* process, which is within the scope of SuperSonic.

Implementation. We implement SuperSonic in Python and use gRPC for distributed communications. SuperSonic builds upon CompilerGym [23] and RLlib (Ray) [51] by utilizing their APIs for task definitions and RL algorithm implementations. SuperSonic currently supports 23 RL algorithms from RLLib [51] and 10 pre-trained DNNs and functions for representing the program state or computing the reward. Compiler developers may choose a subset, add their own, or include all (the default) supported RL algorithms and models for the meta-optimizer to search over.

Task definition. The compiler developer first defines the optimization problem by creating an RL policy interface (Figure 2). The definition includes a list of client RL components for the meta-optimizer to search over.

Client RL search. Calling to `policy_search()` invokes SuperSonic meta-optimizer, where the developer can also limit the number of trials spent on client RL searching.

Client RL parameter tuning and deployment. After choosing an RL architecture, the meta-optimizer will fine-tune a set of model-specific hyper-parameters of the selected client RL (see also Table 1). Hyperparameter tuning is performed on the training benchmarks. The tuned client RL and its parameters are saved, which can be shipped with a compiler to optimize unseen programs at deployment time.

Measurement engine. The measurement engine evaluates a code transformation option using an user-supplied interface (line 24 in Figure 2). Measurements are used during client RL search and tuning, as well as deployment phases to obtain feedback for a chosen optimization action.

3.2 Task Definition

The user-defined client RL search space typically includes candidate functions (or models) for representing the environment state, objective functions for computing the reward, and the set of possible actions that can be taken from a given state. This search space definition can optionally include a chosen set of RL exploration algorithms and transition functions to be used by a client RL algorithm. By default, SuperSonic automatically search over all supported RL algorithms where each algorithm has a default transition function. Furthermore, the compiler engineer also needs to provide a `run` function, which provides the measurement of an action to compute the reward. These are implemented in a small Python program to interface with the SuperSonic API. The definition code is similar to the programming environment of mainstream auto-tuners like OpenTuner and CompilerGym, allowing the developer to quickly port their code to use the SuperSonic search and tuning components.

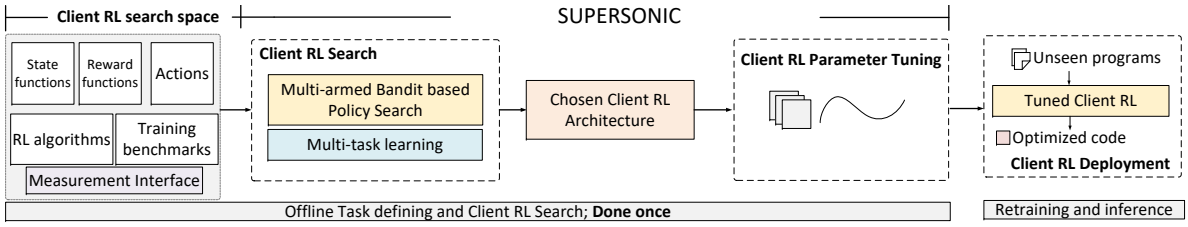


Figure 1. Overview of SuperSonic components. This framework enables developers to express the optimization space. It automatically searches for the optimal client RL architecture to be used for inference during deployment.

```
1  import SuperSonic as ss
2  from SuperSonic.stateFunctions.models import *
3  ...
4  statefs = [Word2Vec(...),Doc2Vec(...),CodeBert
       (...),ActionHistory(...)]
5  tranfs = [DNN(...),CNN(...),LSTM(...)]
6  rewards = [RelativeMeasure(...),tanh(...)]
7  rl_algs = [MCTS(...),PPO(...),DQN(...),QLearning
       (...)]
8  actions = [Init(...)]
9  ...
10
11 class SuperOptimizer(ss.PolicyInt):
12   def __init__(self, statefs, tranfs, actions,
         rewards, rl_algs):
13       self.PolicySpace = {
14         "StatList": statefs,
15         "TranList": tranfs,
16         "ActList": actions,
17         "RewList": rewards,
18         "AlgList": rl_algs,
19       }
20       self.search_engine = SearchEngine(self.
             PolicySpace)
21
22   def run(self):
23     #user code for compilation and execution
24     ...
25     return Result(time=run_result['time'])
26
27 if __name__=='__main__':
28   opt = SuperOptimizer(statefs, tranfs, actions,
         rewards, rl_algs)
29   policy = opt.policy_search(
         training_benchmark_list, num_of_trials=100)
```

Figure 2. Simplified tasks definition for superoptimizaton.

Figure 2 gives a simplified example that defines the client RL search space for superoptimizaton [43, 55]. This example specifies candidate methods for representing the environment state, functions for computing rewards, the definition for action space, and a chosen set of client RL algorithms. The `run` function implements the measurement interface, including user code for compiling and executing the program for a given code transformation action. Finally, the program invokes the policy search API by passing in a list of training benchmarks and the number of search trials. We stress that measurement interface defines how to compile and execute a test program, but the target program can be of any language.

3.3 Client RL Search

Given a client RL search space, the SuperSonic meta-optimizer automatically finds a suitable client RL architecture (i.e., <state function, transition function, reward function, RL algorithm> and potentially a value function) from training benchmarks. We formulate the client RL search problem as a MAB problem that is solved using a parallel DRL algorithm. Deep RL can reuse knowledge from other tasks to speed up the search. In contrast, evolutionary algorithms have to search from afresh. Our work is the first to formulate client RL tuning for code optimization as a MAB problem and employs deep RL as a meta-solver.

Our meta-optimizer is a variant of the Asynchronous Advantage Actor Critic (A3C) algorithm [38, 59]. A3C is a distributed algorithm, where multiple workers independently update a global value function – hence "asynchronous". We choose this algorithm because it is shown to be effectively in other RL application domains [38] and permits us to develop a parallel policy search engine. Specifically, the meta-optimizer consists of two RL models, an *actor* for computing an action based on observation and a *critic* for estimating a reward value. In a nutshell, the actor is a policy RL that takes as input the environment state and outputs the best action (a policy architecture in our context). The actor essentially controls how the meta-optimizer choose a candidate client policy to try out. By contrast, the critic is value-based RL that evaluates the action by adjusting its value function to estimate the maximum future reward based on the historical observations obtained from training benchmarks. As time passes, the actor is learning to produce better actions, and the critic is getting better at evaluating those actions.

Like [38], we implement the actor and the critic using a stacked neural network consisting of a ResNet convolution neural network (CNN) [46] that is followed by a LSTM recurrent neural network (RNN) [39]. We use the output of the LSTM to update the policy function of the actor and the value function of the critic. Input to the actor model is a 1-dimensional history vector containing the last 20 actions (policies) that the meta-optimizer has tested. Input to the critic model is a history vector plus a cumulative reward averaged across benchmarks, computed using an Area under the Curve (AUC) function.

Client RL searching strategy. At each of the n trials, the meta-optimizer obtains an action (i.e., a client policy architecture to test) from the actor model. The meta-optimizer uses the user-provided measurement function to obtain the observation, which is then used to compute the current reward. The current reward and the environment state is passed to the critic model to update its value function. The value function of the critic also estimates the future reward which is given to the actor to update its policy network. By default, the meta-optimizer runs each client RL for 50 exploration steps during a trial to allow it to converge before taking the observation. We use the 20 most recently chosen policy architectures as the state. We then measure the area under the reward curve of the recently chosen 20 policies to compute the current reward. After the meta-optimizer performs n trails on training benchmarks, we check the latest 20 actions chosen by the actor. We then use the most frequently chosen policy architecture as the outcome of policy search. Our intuition is that the actor would be more efficient in picking good policy architectures towards the end of the search and would choose the optimal policy as the action more often. This search process also implicitly model the learning time of a candidate client RL. A client RL is chosen because it can learn quickly to give good results on training benchmarks within the given time.

Failure during client RL search. In this work, we did not observe failure in combining RL components in our case studies. When using a client RL, failures can happen - the underlying compiler (driven by RL) may fail due to e.g., invalid combinations of compiler flags or compiler bugs. These are automatically handled by RL which will avoid trying these options in future iterations. Furthermore, while SuperSonic could miss an RL architecture that can be improved through fine-tuning, we have performed a large-scale search on our case studies and did not observe this. This issue can also be mitigated by performing RL-architecture search and parameter fine-tuning in a single process.

Multi-task learning. If multiple optimization tasks are defined, we then use multi-task learning (MTL) to find an individual policy for each task given a total budget of n trials. SuperSonic provides a distributed, parallel meta-optimizer, building on top of an open-source MTL framework [38]. For each optimization task, it creates an environment and assigns a meta-optimizer instance to the environment, so that client RL search for different optimization tasks can be performed in different, potentially distributed environments. In our evaluation, we implement an execution environment in a Docker container. SuperSonic realizes different environments and their associated actors as parallel workers that can run on a single machine or multiple distributed machines. An issue of using a standard MTL algorithm is that the learner is likely to give more resources (e.g., #trials, time or machines) to tasks with higher rewards, leading to unfair resource allocation among tasks. To address this issue, we use a regularization mechanism, similar to [84], by adding a normalization layer to the actor and the critic network.

Table 1. Example tunable parameters

Algorithms	Parameters
Common param.	Batch size for workers; Train batch size; Mini-batch size; learning rate
MCTS	Dirichlet noise and epsilon; Puct coefficient; Simulation times; Loss temperature
PPO	Entropy coefficient; Adam optimizer step size; GAE estimator parameter; Policy ratio clipping
DQN	#atoms; Discrete supports; Adam epsilon; Clip gradients
Q-Learning	Behavior Cloning Pretraining numbers; Q-Learning loss temperature; Lagrangian threshold; Min Q weight multiplier

3.4 Client RL Parameter Tuning

During the client RL search stage, SuperSonic uses the default hyperparameter and pre-trained models. After a client RL architecture is chosen, the SuperSonic meta-optimizer uses training benchmarks to fine-tune a set of common and algorithm-specific hyperparameters. Each SuperSonic built-in DNN model also has a standard training API. Therefore, the meta-optimizer also uses the measurements and observations generated during parameter tuning to fine-tune the relevant DNN models, e.g. for state representation. Table 1 gives some of the example hyperparameters supported by SuperSonic. We note that the user does not need to explicitly supply these parameters because they are known to SuperSonic. Our parameter tuning method is a parallel population-based training (PBT) algorithm [41] from RLlib. Like client RL search, the user can also specify how much trials can be spent on parameter tuning. Once the budget is used up, the best-found parameter setting is returned. Our evaluation applies cross-validation to ensure the tuned RL is always tested on new, previously unseen benchmarks.

3.5 Client RL Deployment

Finally, the tuned client RL is saved as serialized objects. The chosen hyperparameters and action space are stored in JSON files. SuperSonic provides APIs to load and reuse the stored objects to optimize any new program. To apply a tuned RL, SuperSonic creates a session to apply a standard RL loop to optimize the input program by using the chosen RL exploration algorithms to select an action for a given state. For example, the state could be a vector recording the last n compiler options added into the compiler flags or a DNN (see also Section 5.5.3).

3.6 Measurement Engine

For a given optimization option, the measurement engine invokes the user-supplied `run` function to compile and execute the program in the target environment. The user function

Table 2. Case studies in our evaluation

Use cases	#Bench.	Competing Methods	Search space
C1: Optimizating image pipeline	10	Halide master [62], auto-schedulers [3], HalideRL [68], OpenTuner	$7^3 * 2^7 \sim 7^3 * 3^7$
C2: Neural network code optimization	5	AutoTVM [19], Chameleon[5], OpenTuner	$5 * 10^3 \sim 20 * 10^3$
C3: Code size reduction	43	CompilerGym [23], OpenTuner	$123^{60} \sim 123^{150}$
C4: Superoptimization	40	STOKE [76], OpenTuner [9]	$9^{100,000} \sim 9^{16,000,000}$

Table 3. Candidate state functions and reward functions

		Case Studies: 1	2	3	4
State func.	Word2Vec [58]				
	Doc2vec [49]		✓		✓
	CodeBert [29]				
	Manual features (e.g. LLVM IR representation from [40])	✓			
	Action History				
	Hash of Action History			✓	
Reward func.	Relative measure (e.g. speedup, code size reduction ratio)	✓	✓		
	function output (e.g. tanh())			✓	✓

reports the result for each execution, which is stored in a result database implemented using SQLLite. The database also holds information obtained during the search, including the action history, reward, and execution outputs for each benchmark. The client RL gathers the result of an action by querying the result database. Decoupling the RL exploration and measurement allows the parallel execution of measurement and the RL agent, possibly across different machines. Parallel execution can reduce the measurement cost, which often dominates the auto-tuning process.

4 Experimental Setup

We evaluate our approach by applying it to four code optimization tasks and compare it against eight tuning methods, including hand-tuned RL solutions. Table 2 summarizes our evaluation setup, including the search space size and competing methods for each case study. We note that the overhead of RL is dominated by gathering feedback from the environment through, e.g., compiling and executing the program. While case studies 3 and 4 have a larger search space than case studies 1 and 2, obtaining feedback incurs lower overhead in case studies 3 and 4 compared to 1 and 2, leading to an overall faster search time for case studies 3 and 4.

4.1 Case Study 1: Optimizing Image Pipelines

The problem. This task aims to improve the optimization heuristic of the Halide compiler framework for image processing [69].A Halide program separates the expression of the computation kernels and the application processing pipeline from the pipeline's *schedule*. Here, the schedule defines the order of execution and placement of data on the hardware. The goal of this task is to automatically synthesize schedules to minimize the execution time of the benchmark.

Methodology. This task builds upon Halide version 10. Our evaluation uses ten Halide applications that have been heavily tested on the Halide compiler. We measure the execution time of each benchmark when processing three image datasets provided by the Halide benchmark suite. The benchmarks are optimized to run on a multi-core CPU.

Competing methods. We compare SuperSonic against four prior methods designed for optimizing Halide schedules. These include two Halide built-in auto-scheduling algorithms (Halide master [62] and auto-scheduler [3]), a hand-tuned RL method (HalideRL) [68], and OpenTuner. We show speedups over Halide master.

Actions. Each Halide program comes with a scheduling template that defines an n stages schedule. We can apply optimizations like loop tiling and vectorization to each stage. We apply four actions to construct a n-stage scheduling sequence. These include adding or removing an optimization to the stage and decreasing or increasing the value (by one) of an enabled parameterized option.

4.2 Case Study 2: Neural Network Code Generation

The problem. This task targets DNN back-end code generation to find a good schedule. e.g., instruction orders and data placement, to reduce execution time on a multi-core CPU.

Methodology. This study is conducted within the TVM compiler v 0.8 [18]. We use 5 CNN kernels where their schedule optimization space is defined by the TVM developer.

Competing methods. We compare our approach against four TVM built-in tuning strategies, including random search, genetic algorithms, grid-based search and XGBoost-based search. In addition to these, we also compare our approach to OpenTuner and Chameleon [5] - a recently proposed, hand-tuned RL method designed for TVM. We show the improvement over the TVM compiler (TVMC) without schedule optimization.

Actions. Each TVM benchmark comes with a schedule template that defines a set of tuning knobs like loop tiling parameters. We consider four actions in this task: adding or removing a knob to the schedule sequence, and decreasing or increasing the parameter value (by one) of a parameterized knob in the optimization sequence. The number of tuning configurations vary across benchmarks (Table 2).

4.3 Case Study 3: Code Size Reduction

The problem. This task is concerned with determining the LLVM passes and their order to minimize the code size.

Methodology. Following the setup of CompilerGym, we compute the code size reduction by measuring the ratio

of LLVM IR instruction count reduction over the LLVM -Oz code size optimization option. This metric is platform-independent and deterministic. Note that the IR instruction count strongly correlates to the binary size - fewer instructions typically lead to a smaller binary. In this evaluation, we use 43 benchmarks: 23 from the CBench suite [1] and 20 single-source benchmarks from the LLVM test suite [2].

Competing methods. We compare our approach against OpenTuner and the Greedy and Random search strategies for code size reduction implemented by CompilerGym. The CompilerGym Greedy algorithm has a threshold, e, for controlling how often the algorithm switches between random and greedy searches, with $e = 0$ for a solely greedy strategy and $e = 1$ for a purely random algorithm. We set e to 0.1, which produces comparable results as CompilerGym developers reported on their platforms.

Actions. We consider all the 123 semantics-preserving passes of LLVM. The RL agent determines which pass to be added into or removed from the current compiler pass sequence. Note that the length of the compiler pass sequence is unbounded. An LLVM pass can appear multiple times in the pass sequence and be inserted before or after any pass.

4.4 Case Study 4: Superoptimization

The problem. This classical compiler optimization task finds a valid code sequence to maximize the performance of a loop-free sequence of instructions [43, 55]. Superoptimization is an expensive optimization technique as the number of possible configurations grows exponentially as the instruction count to be optimized increases.

Methodology. In this task, we apply SuperSonic to find a client RL for STOKE, the state-of-the-art superoptimizer [76]. Given a set of test cases consisting of input-output pairs and a subset of x86-64 instructions, STOKE synthesizes a program (at the assembly code level) that uses these instructions and agrees with the test cases. We use all the 25 benchmarks from the STOKE Hacker dataset. Additionally, we also extract 15 loop-free and frequently-executed functions from SPEC CPU 2017 and the LLVM test suite, 10 from SPEC and 5 from the LLVM test suite. The seed input to the performance tuner is the assembly code generated by compiling the C code with the -O0 compiler option. We use STOKE's mutation engine to modify the target instructions and its equivalent testing method to verify if the transformed code satisfies the test cases. We also manually verify the correctness of the best-performing version found by each method.

Competing methods. We compare SuperSonic to the Markov chain Monte Carlo (MCMC) based STOKE search technique and OpenTuner. We test all schemes on LLVM and GCC and use -O3 as the baseline. *As noted in the STOKE document, the STOKE implementation is not mature enough to improve -O3 code.* However, as we will show later, SuperSonic can deliver noticeable improvement over -O3 on certain test cases, demonstrating the potential of auto-tuning techniques.

Actions. We consider all the instruction-level transformations supported by STOKE. These include replacing the opcode and operand of an instruction as well as inserting, replacing and swapping instructions.

4.5 Client RL Architecture Search Space

We consider all the SuperSonic-supported RL algorithms when searching the client RL. SuperSonic chooses to use PPO for code size reduction and MCTS for the other three tasks. Table 3 lists the state and reward functions considered in each task, where we highlight the SuperSonic chosen function using a check mark. As can be seen from the table, no RL algorithm dominates our case studies - the best algorithm depends on the optimization task. However, the Supersonic-chosen RL algorithm generalizes well to input test programs of an optimization task.

4.6 Hardware and Software Platforms

For case studies 1, 2 and 4, we use two multi-core servers to evaluate the resulting code. The first server has 2x 26-core Intel Xeon 8179M CPU running at 2.40GHz, and the second server has 2x 32-core AMD EPYC 7532 CPU at 2.4GHz. Both servers have 128GB of RAM and run Ubuntu 20.04 with Linux kernel v5.4. In our evaluation, we run the SuperSonic meta-optimizer on the AMD server and use the chosen client RL to optimize the target task on both machines. We run the relevant deep learning models on an NVIDIA RTX 2080 Ti GPU. We use LLVM v10 as the backend compiler to generate the executable binary in our evaluation. For superoptimizaton, we also test our approach on GCC v11.2.

4.7 Performance Report

Cross-validation. We use 3-fold cross-validation to evaluate our approach. This means we partition the benchmarks into three groups (folds). We perform client RL search and tuning on two folds of the benchmarks and then test the tuned RL architecture on the remaining benchmarks. We repeat this procedure three times to test each of the three folds in turn. Unless stated otherwise, we exclude the time spent on client RL search and tuning from the overhead of deployment-time performance optimization, because client RL search and tuning is a *one-off* cost performed offline and the tuned RL architecture can be applied to many new programs without incurring this tuning-search overhead. However, *we give the same amount of search time/iterations for all methods when optimizing a test program.*

Runtime measurement. To measure the runtime of the resulting binary, we run each benchmark at least 100 times on an unloaded machine. For each benchmark, we also compute the 95% confidence interval bound and increase the

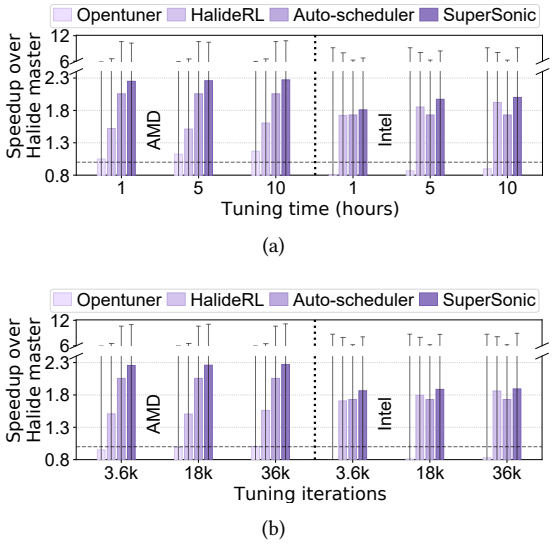


Figure 3. Performance to expert-tuned Halide schedules under different search time (a) and iteration (b) constraints. SuperSonic gives the overall best performance than other auto-tuning methods.

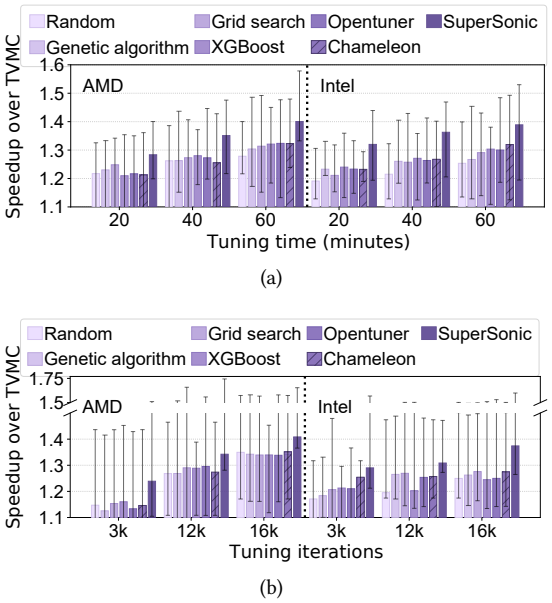


Figure 4. Speedup for DNN code generation under different search time (a) and iteration (b) constraints. The min-max bar shows the range across benchmarks. SuperSonic outperforms all competing methods on both platforms.

number of profiling runs if the interval is greater than 2%. We report the geometric mean across runs. We also show the performance variances across benchmarks, compilers and cross-validation settings as min-max bars on the diagram.

Client RL search. In our evaluation, we apply MTL to perform RL search for all four optimization tasks simultaneously on a single server, with a total search budget of 100 trials.

5 Experimental Results

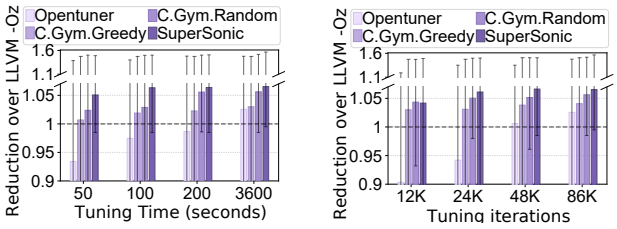
In this section, we first present the case study results, finding that SuperSonic outperforms hand-crafted strategies in each task. We then provide an analysis of SuperSonic's working mechanisms, showing SuperSonic can accelerate the deployment-time search by 1.75x. Note that the tuning time is proportional to the number of tuning iterations, but the tuning time per iteration can vary between evaluated methods. Specifically, the SuperSonic-chosen client RL can perform, on average, 1, 3, 24 and 10 search iterations per second for case studies 1, 2, 3 and 4, respectively, on our evaluation platforms.

5.1 Case Study 1: Optimizing Image Pipelines

Figure 3 reports speedup over the Halide master scheduler [62]. SuperSonic gives noticeable performance improvement on the AMD platform. It also manifests larger advantages when the search time is limited. On certain benchmarks, SuperSonic is able to gives over 11x speedup with a correctly optimized code. The tree-based auto-scheduler gives a high speedup of over 10x for a single benchmark, but it has a lower mean performance improvement across benchmarks compared to SuperSonic. On average, SuperSonic delivers a 1.5x improvement over HalideRL, a manually tuned RL strategy. Note that HalideRL uses PPO as the exploration algorithm and the sequence of already applied schedules as the state function. By contrast, SuperSonic determines that for this task, using MCTS as the exploration algorithm and a hash function of the applied schedules as the state function is better than HalideRL. MCTS compares complete schedules through simulations and looks ahead before making intermediate scheduling optimizations, leading to a better result. Overall, SuperSonic outperforms all alternative search techniques on all but two benchmarks on both platforms.

5.2 Case Study 2: Neural Network Code Generation

Figure 4 reports the the performance improvement over the default schedules. RL based methods (Chameleon and SuperSonic) can generate better code than TVM's evolutionary or predictive modeling based search techniques. While random and grid-based search can significantly improve one benchmark (the top point of their min-max), their performance is not robust and can give poor performance for other benchmarks. By contrast, SuperSonic delivers more robust performance by giving no slowdown on the AMD platform and only minor slowdown over XGBoost on two benchmarks on the Intel platform. SuperSonic improves Chameleon, the second best-performing method, by up to 1.22x, improving the default schedules by up to 1.74x.



(a) (b)

Figure 5. Code size reduction over LLVM -Oz under different search time (a) and iteration (b) constraints. SuperSonic outperforms alternative methods.

Table 4. Important compiler passes for code size reduction and how often a pass is chosen by a search method.

	Opentuner	C.Gym.Greedy	C.Gym.Random	SuperSonic
simplifycfg	0.84%	3.29%	1.23%	**10.78%**
early-cse-memssa	0.78%	0.18%	0.92%	**6.80%**
gvn	0.69%	1.83%	0.97%	**5.98%**
instcombine	0.75%	0.12%	1.17%	**8.99%**
newgvn	0.74%	2.90%	1.02%	**7.66%**
others	**96.21%**	91.67%	94.69%	59.79%

5.3 Case Study 3: Code Size Reduction

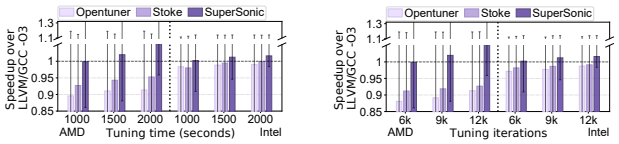
Figure 5 reports the code size reduction conducted on the Intel platform using LLVM. We note that an average code size reduction of 4% is considered to be significant [72–74].SuperSonic improves -Oz for all but two test benchmarks. It delivers the best reduction for 90% of the test benchmarks. For those benchmarks where SuperSonic does not deliver the best reduction, the difference between SuperSonic and the best-performing method is small, less than 1%. On average, SuperSonic gives the highest mean code size reduction, improving LLVM -Oz by up to 1.57x.

Table 4 lists the top-5 most frequently-appeared LLVM passes in the compiler pass list that gives the best code size reduction. Because these passes are often included in a compiler sequence that offers a high code size reduction, they are likely to be important for reducing code size. The table also shows how often a search method chooses a pass as an action when optimizing a benchmark. Compared to other methods[2], SuperSonic picks an important pass more often during the search, suggesting that it learns the importance of optimization passes.

5.4 Case Study 4: Superoptimization

Figure 6 shows the superoptimization results using LLVM and GCC. SuperSonic outperforms Stoke and OpenTuner on most of the test benchmarks with a higher mean speedup. Increasing the tuning iterations (and the search time) during deployment can improve the search performance because it allows the search algorithm to explore the optimization space better and uses feedback to improve its search strategy. Although the STOKE developers note that their mutation engine is not matured enough to outperform LLVM/GCC -O3, we demonstrate that RL can deliver noticeable improvement by better exploring the optimization space (up to 1.34x).

[2]In theory, with a sufficiently larger number of samples, each pass will have a 1/123 chance to be chosen by C.Gym.Random in Table 4.



(a) (b)

Figure 6. Speedup over LLVM/GCC -O3 for superoptimization under different search time (a) and iterations (b).

Figure 7 shows a kernel from STOKE Hacker benchmark dataset, compiled by LLVM -O3, and the best-performing version found by SuperSonic. The SuperSonic code for the `_Z3p23i` kernel is 18 lines shorter, 1.2x faster than -O3. This is one of the many examples where the Supersonic-driven superoptimization generates faster code over -O3.

5.5 Further Analysis

5.5.1 Compare to Other Client RL Search Strategies. This experiment compares SuperSonic's deep RL-based meta-optimizer against four widely used parameter search techniques: grid search, simulated annealing, random search, and genetic algorithms. We set the number of search iterations to 100 for all search algorithms. We also vary the search space by adding more candidate RL components, where the number of RL component combinations vary between 150 and 800. For a chosen RL client, we follow the same parameter fine-tuning process to fine-tune the selected RL components (Section 3.4). Figure 8 compares the performance of the client RL chosen by different search algorithms during the RL client search stage. As can be seen from the diagram, all client RL give an averaged improvement over the baseline. SuperSonic finds a better client RL during the limited client RL search budget, leading to overall better performance across search space sizes and case studies. We note that the limited client RL search budget restricts the performance of finding good client RL with a large search space (e.g., when the number of RL component combinations is 800). Increasing the search budget will allow the meta-optimizer to find a better client RL to improve the resulting performance.

5.5.2 Deployment-stage Search Time Relative to the Best-performing Alternative. Table 5 shows the relative search time and iteration count required by SuperSonic-tuned client RL to exceed the results given by the best-performing competitive scheme. The table shows the minimum, average, and maximum search overhead across

```
.L_4004e0_LLVM_O3:                .L_4004e0_SuperSonic:
  movl   $0x60106f,%eax             movl   $0x601191,%
       edi
  movl   $0x601191,%edi             movq   $0x400990,%
       rcx
  movl   $0x400700,%edi             movl   $0x400700,%
       edi
  pushq  %rax                       pushq  %rax
  nopl   0x0(%rax,%rax,1)           nopl   0x0(%rax,%rax
       ,1)
  ...                               ...
  callq  4008f0 <_Z3p23i>           callq  4008f0 <
       _Z3p23i>

._Z3p23i:                         ._Z3p23i:
  movl   %edi,%eax                  popcntl %edi,%ecx
  shrl   %eax                       movq   %rcx,%rax
  andl   $0x55555555,%eax           retq
  subl   %eax,%edi
  movl   %edi,%eax
  andl   $0x33333333,%eax
  shrl   $0x2,%edi
  andl   $0x33333333,%edi
  addl   %eax,%edi
  movl   %edi,%eax
  shrl   $0x4,%eax
  addl   %edi,%eax
  andl   $0xf0f0f0f,%eax
  movl   %eax,%ecx
  shrl   $0x8,%ecx
  addl   %eax,%ecx
  movl   %ecx,%eax
  shrl   $0x10,%eax
  addl   %ecx,%eax
  movzbl %al,%eax
  retq
  nopl   0x0(%rax,%rax,1)
```

Figure 7. SuperSonic finds a better program (right) over LLVM -O3 (left) for a STOKE Hacker kernel (`_Z3p23i`) that counts the number of bits, by using the `popcntl` instruction.

Table 5. Search overhead required by SuperSonic to exceed the performance given the best-performing alternative

Use cases	% of search time (& raw numbers)			% of iterations (& raw numbers)		
	MIN	**GeoMean**	**MAX**	**MIN**	**GeoMean**	**MAX**
Case study 1	3.6 (21 mins)	39.6 (4 hours)	72.1 (6 hours)	3.9 (1,425)	44.0 (15,821)	75.3 (27,141)
Case study 2	1.0 (1 min)	39.1 (24 mins)	74.1 (45 mins)	4.2 (885)	32.3 (3,885)	68.8 (8,256)
Case study 3	1.5 (3 sec)	31.0 (61 sec)	61.1 (2 mins)	2.1 (738)	29.7 (10,714)	83.5 (30,068)
Case study 4	1.1 (22 sec)	32.8 (11 mins)	42.3 (14 mins)	1.2 (1,478)	36.5 (43,745)	46.9 (56,387)

test benchmarks. On average, the RL architecture found by SuperSonic converges faster, requires less than 39.6% of the search time used by the best-performing alternative search algorithm to deliver better results. This means the RL architecture chosen by SuperSonic can start delivering a better optimized code with, on average, 1.75x less search time (up to 100x) compared the best-performing alternative tuning algorithm that runs longer. While search methods like genetic algorithms have no upfront training cost, they must search each time afresh. SuperSonic performs a one-off offline RL search and tuning, but the tuned client RL significantly reduces the search cost for any *new* programs after that.

5.5.3 Chosen RL Components. Table 3 shows the client RL components chosen by SuperSonic varies from one task to another. We also observe that the optimal RL found by SuperSonic is consistent across cross-validation runs for a given task. This means the client RL architecture can generalize across inputs of a given task. We notice that using MCTS with DNNs for state representation gives good performance for three out of the four tasks, but it is less effective for code size optimization compared to PPO. Using MCTS for code reduction gives an average reduction of 0.6% instead of 6.5% delivered by PPO. This perhaps due to a large number of discrete actions (i.e., the compiler passes) in the optimization space, which requires more search time to learn a good value function to efficiently guide the MCTS simulation.

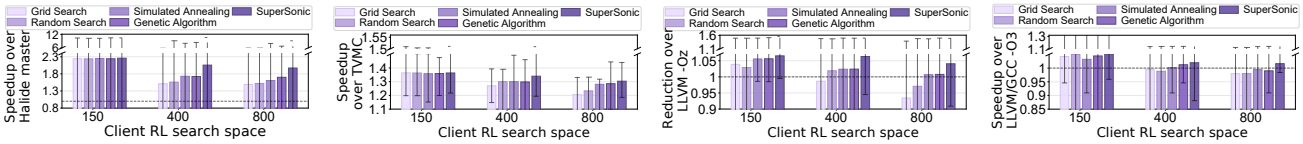
6 Discussions

We have showcased that deep RL can be highly effective in performance tuning, but this depends on having a suitable RL architecture. Deep RL can benefit from knowledge learned from other inputs (or programs). Some of this knowledge can be transferable from one program to another. For example, the client RL algorithm learns applying `-simplifycfg` (Table 4) is always beneficial for code size reduction from training benchmarks. The RL agent thus increases Evolutionary-based search algorithms cannot use prior knowledge and have to start from scratch for every new program [40].

Our experiments show that evolutionary-based techniques can outperform RL on certain test cases (Section 5.1). Because evolutionary algorithms often have a lower runtime overhead than deep RL methods, it is interesting to see if both methods can be combined to explore more optimization points of the search space. For example, one can first employ a deep RL to identify the most promising areas in the search space and then use low-cost evolutionary algorithms to perform finer-grained searches on the identified regions. Another example is to use evolutionary algorithms to combine and dynamically choose between multiple state or reward functions to improve the robustness of the system.

We found the measurement cost often dominates the search time of an autotuner. One way of reducing the measurement overhead is to use active learning to reduce the number of runs when we are confident that the results are statistical sound [63, 64]. Another strategy is to use a predictive model to estimate outcome of runtime measurements to reduce environment interactions.

SuperSonic searches the client RL by trying different combinations of RL components. An interesting approach



(a) Optimizing Image Pipelines (b) Neural Network Code Generation (c) Code size reduction (d) Superoptimization

Figure 8. Comparing the performance of a client RL chosen by different search algorithms during the client RL search stage. We vary the number of client RL combinations by varying the number of candidate RL components. The SuperSonic chosen client RL gives the best overall performance during deployment.

would be to combine online search and predictive modeling, by using the profiling data collected from earlier runs to build a predictive model to directly predict the promising RL component combinations to reduce the search overhead.

Finally, RL often employs machine-learning (ML) models to estimate the reward or to represent the environment state. However, ML is brittle, and small changes in data distribution during deployment can result in incorrect predictions. Therefore, techniques for detecting when the model's estimation can be trusted is useful [71, 91].

7 Related Work

Auto-tuning techniques have been widely used to reduce expert involvement in performance optimization tasks [88]. Early works like ATLAS [89] and FFTW [31] demonstrate the potential for searching domain-specific optimization parameters like the loop tile size. In iterative compilation, prior works have exploited the use of evolutionary algorithms like the genetic algorithm [7, 8] and other search techniques built upon Bayesian optimization [11, 17, 42, 56, 75] and predictive modeling [4, 10, 15, 48, 67]. Recent works also apply auto-tuning techniques to optimize code generation for deep neural networks [18, 34, 61, 92], image processing applications [69], and runtime tuning of operating system and processor parameters [20]. Our work is among these efforts in applying auto-tuning techniques for code optimization.

In recent years, RL (in particular deep RL) has demonstrated impressive results in domains like game-playing [66, 78] and robotics [6]. It has also been employed for various performance optimization tasks, including compiler phase ordering [36], loop optimization [13], choosing vectorization parameters [35], task scheduling [54] and memory placement [45]. These prior works rely on hand-tuned policies or feature engineering to derive a good search strategy for the given application domain. Given the large number and diversity of workloads to be optimized, approaches like ours for automating RL architecture tuning are attractive.

CompilerGym [23] is an machine-learning platform for compiler optimization. It provides an OpenAI Gym-like programming environment [65]. SuperSonic utilizes CompilerGym's API for problem definition, but complements to CompilerGym by providing ways to automate RL component searching and tuning. We plan to integrate SuperSonic into the mainstream release of CompilerGym.

In a relevant research area, neural architecture search [28] aims to find the right neural network structure and parameters to reduce the model size and execution time or to improve accuracy [82, 93]. Our work draws aspiration from these past foundations to find a good RL architecture for performance optimization.

8 Conclusions

We have presented SuperSonic, a framework for building RL-based performance tuners. SuperSonic provides the capability to automate the process of designing and tuning RL algorithm structures. We evaluate SuperSonic by applying it to four different optimization tasks. Experimental results show that the RL architecture found by SuperSonic delivers better overall performance than alternative search techniques, including hand-tuned RL strategies.

SuperSonic supports mainstream and emerging RL programming environments like RLlib and CompilerGym. It is designed to provide customizable interfaces to allow developers to introduce new algorithms and methods to be used in the RL policy architecture. As the community provides more models and methods, SuperSonic will be able to explore a more comprehensive policy search space. As a result, there will be less of a need to create domain-specific methods for each project. In the long-term, we hope that SuperSonic will work out-of-the-box for most performance developers once they have defined their optimization tasks. We hope the findings of this paper and the release of SuperSonic can encourage the adoption of RL in many other code optimization problems.

Acknowledgments

This work was supported in part by the National Science Foundation of China (NSFC) under grant agreements 61972314, 61872294 and 61902170, the International Cooperation Project of Shaanxi Province under grant agreements 2021KW-04, 2020KWZ-013 and 2021KW-15, and a Meta (Facebook) research award. For any correspondence, please contact Zhanyong Tang (Email: zytang@nwu.edu.cn) and Zheng Wang (Email: z.wang5@leeds.ac.uk).

Appendix: Artifacts Evaluation Instructions

A.1 Overview

Our research artefact enables the reproduction of our approach and figures from our experimental results. This document consists of instructions for performing AE on pre-configured notebooks (Section A.2) or Python scripts within a docker image (Section A.3).

Main Results

Our AE enables a reduced-size evaluation for the main results of our work, i.e., Figures 3-6 in the paper. The results compare the performance of the client RL found by our tool against prior search-based techniques. We also provide a small-scale experiment to showcase how the developed techniques can be used to search for a client RL architecture.

Artifact Check-list (Meta-information)

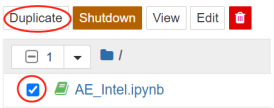
- **Compilation:** LLVM 9.0, gcc 7.5 and gcc 4.9 included
- **Experiments:** The experiments can be run with the included scripts.
- **How much disk space required (approximately)?:** 40 GB
- **How much time is needed to prepare workflow (approximately)? :** Several hours for a small-scale evaluation and one week for a full-scale evaluation
- **Publicly available? :** Yes, code and data are available at: https://github.com/HuantWang/SUPERSONIC
- **Code licenses:** CC-BY-4.0 License
- **Archived:** 10.5281/zenodo.6313660

A.2 Instructions for Interactive Notebooks

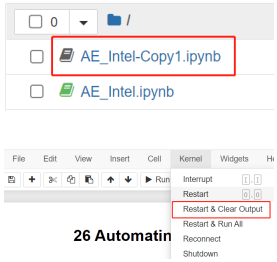
For convenience, we have provided a pre-configured Python Jupyter Notebook within a pre-configured docker image - see https://github.com/HuantWang/SUPERSONIC/blob/master/AE.md for how to access the docker image.

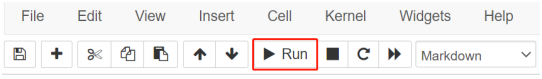
Experiment Workflow

1. Access the Jupyter Notebook using the method described above.
2. From the Jupyter landing page, select the checkbox next to the notebook, i.e., `AE_Intel.ipynb`; then click "Duplicate".



3. Click the name of the newly created Jupyter Notebook, e.g. AE_Intel-Copy1.ipynb.
4. Select "Kernel"> "Restart & Clear Out" from the menu to reset the results.
5. Repeatedly press the *play* button (the tooltip is "run cell, select below") to step through each cell of the notebook. Alternatively, select each cell in turn and use "Cell" > "Run Cell" from the menu to run specific cells. Note that some cells depend on previous cells being executed. If any errors occur ensure all previous cells have been executed.





Evaluation and Expected Results

Code cells within the Jupyter Notebook display their output inline. Note that some cells can take a few minutes to hours to complete; please wait for the results until step to the next cell. High load can also lead to a long wait for results. This may occur if multiple reviewers are simultaneously trying to generate results.

Customisation

The experiments are fully customisable, the code provided in the Jupyter Notebook can be edited on the spot. Simply type your changes into the code blocks and re-run using "Cell" > "Run Cells" from the menu.

A.3 Instructions for Docker Image

For a step-by-step instruction to replicate our results using a docker image on your machine locally without the interactive notebook, please refer to our GitHub repository (https://github.com/HuantWang/SUPERSONIC/blob/master/AE.md).

A.4 Evaluation and Expected Result

The supplied notebook or scripts will automatically produce the results or diagrams after the experiments. The AE also includes the data used in the paper submission. The results may be different if the experiment is performed on hardware that differs from the ones used in the paper.

A.5 Experiment Customization

Our evaluation scripts contain customizable parameters to change things like the number of search iterations and the training and testing datasets used. Please follow the instructions given at https://github.com/HuantWang/SUPERSONIC/blob/master/AE.md.

References

[1] [n. d.]. Collective Benchmark. https://ctuning.org/wiki/index.php/CTools:CBench.

[2] [n. d.]. llvm-test-suite. https://github.com/llvm/llvm-test-suite.

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[4] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 11–pp.

[5] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2019. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation, In International Conference on Learning Representations. *ICLR'20*.

[6] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. 2019. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113* (2019).

[7] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences. *ACM SIGPLAN Notices* 39, 7 (2004), 231–239.

[8] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 38–49.

[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[10] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* 14, 3 (2017).

[11] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (jun 2016), 25 pages. https://doi.org/10.1145/2928270

[12] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 3588–3600. http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf

[13] Alexander Brauckmann, Andrés Goens, and Jerónimo Castrillón. 2021. A Reinforcement Learning Environment for Polyhedral Optimizations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[14] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

[15] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 185–197.

[16] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 12–23.

[17] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1198–1209.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[19] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. 3393–3404.

[20] Concertio. [n. d.]. Concertio: Autonomous Optimization Platform.

[21] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[22] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE.

[23] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research.

[24] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 378–390.

[25] JW Davidson, Gary S Tyson, DB Whalley, and PA Kulkarni. 2007. Evaluating heuristic optimization phase order search algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 157–169.

[26] Christian Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *19th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[27] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman P. Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015).

[28] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.

[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[30] Alvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michele Sebag. 2010. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence* 60, 1 (2010), 25–64.

[31] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[32] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[33] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. 2017. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 3389–3396.

[34] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. 2019. Wootz: A Compiler-Based Framework for Fast CNN Pruning via Composability *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 717–730. https://doi.org/10.1145/3314221.3314652

[35] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[36] Ameer Haj-Ali, Qijing (Jenny) Huang, William S. Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems 2020, MLSys*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.).

[37] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[38] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. 2019. Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3796–3803.

[39] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[40] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. arXiv:2003.00671 [cs.DC]

[41] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[42] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. 2021. *Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3458744.3473355

[43] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*. 304–314.

[44] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[45] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. 2021. Optimizing Memory Placement using Evolutionary Graph Reinforcement Learning. *ICLR*.

[46] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[47] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast searches for effective optimization phase sequences. *ACM SIGPLAN Notices* 39, 6 (2004), 171–182.

[48] Sameer Kulkarni and John Cavazos. 2012. 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*. 147–162.

[49] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Bejing, China, 1188–1196.

[50] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. 2020. AdaTune: Adaptive Tensor Program Compilation Made Efficient. *Advances in Neural Information Processing Systems* 33 (2020).

[51] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*. PMLR, 3053–3062.

[52] Yang Liu, Wissam M Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W Demmel, and Xiaoye S Li. 2021. GPTune: multitask learning for autotuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–246.

[53] Rahim Mammadli, Ali Jannesari, and Felix Wolf. 2020. Static Neural Compiler Optimization via Deep Reinforcement Learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 1–11.

[54] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. 270–288.

[55] Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.

[56] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. 2020. Auto-tuning parameter choices in HPC applications using Bayesian optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 831–840.

[57] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1301.3781

[58] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL]

[59] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[61] Rik Mulder, Valentin Radu, and Christophe Dubach. 2021. Fast Optimisation of Convolutional Neural Network Inference Using System Performance Models. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 104–110. https://doi.org/10.1145/3437984.3458840

[62] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide

image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[63] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2014. Fast automatic heuristic construction using active learning. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 146–160.

[64] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 245–256.

[65] OpenAI. 2020. Gym: a toolkit for developing and comparing reinforcement learning algorithms. https://gym.openai.com/.

[66] Jakub Pachocki, Greg Brockman, Jonathan Raiman, Susan Zhang, Henrique Pondé, Jie Tang, Filip Wolski, Christy Dennison, Rafal Jozefowicz, Przemyslaw Debiak, et al. 2018. Openai five, 2018. *URL https://blog. openai. com/openai-five* (2018).

[67] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. 2013. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming* 41, 5 (2013), 704–750.

[68] Marcelo Pecenin, André Murbach Maidl, and Daniel Weingaertner. 2019. Optimization of Halide Image Processing Schedules with Reinforcement Learning. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*. SBC, 37–48.

[69] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[70] Prashant Singh Rawat, Aravind Sukumaran-Rajam, Atanas Rountev, Fabrice Rastello, Louis-Noël Pouchet, and P. Sadayappan. 2018. Associative Instruction Reordering to Alleviate Register Pressure. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 46, 13 pages.

[71] Jie Ren, Lu Yuan, Petteri Nurmi, Xiaoming Wang, Miao Ma, Ling Gao, Zhanyong Tang, Jie Zheng, and Zheng Wang. 2020. Camel: Smart, adaptive energy optimization for mobile web interactions. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 119–128.

[72] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 110–121.

[73] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 149–163.

[74] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 854–868.

[75] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2021. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1280–1295.

[76] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 305–316.

[77] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[78] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[79] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404

[80] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices* 38, 5 (2003), 77–90.

[81] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[82] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[83] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *arXiv preprint arXiv:2101.04808* (2021).

[84] Hado P van Hasselt, Arthur Guez, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. 2016. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29.

[85] Farui Wang, Weizhe Zhang, Shichao Lai, Meng Hao, and Zheng Wang. 2021. Dynamic GPU Energy Optimization for Machine Learning Training Workloads. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[86] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2020. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security* 16 (2020), 1943–1958.

[87] Yiming Wang, Weizhe Zhang, Meng Hao, and Zheng Wang. 2021. Online Power Management for Multi-cores: A Reinforcement Learning Based Approach. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 751–764.

[88] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[89] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[90] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep program structure modeling through multi-relational graph-based learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 111–123.

[91] Shuangjiao Zhai, Zhanyong Tang, Petteri Nurmi, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2021. RISE: Robust wireless sensing using probabilistic and statistical assessments. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 309–322.

[92] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.

[93] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).