2025

Optimizing the optimizer increasing performance efficiency of modern compilers

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

OPTIMIZING THE OPTIMIZER

INCREASING PERFORMANCE EFFICIENCY OF MODERN COMPILERS

by

HAFSAH SHAHZAD

B.S., Lahore University of Management Sciences, 2013
M.Sc., Technical University of Munich, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2025

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

Third Reader

Wenchao Li, PhD
Associate Professor of Electrical and Computer Engineering
Associate Professor of Systems Engineering

Fourth Reader

Sanjay Arora, PhD
Research Scientist, Red Hat, Inc

Khudi ko kar buland itna ke har taqdeer se pehle
Khuda bande se ye puche bata teri raza kia hai

Iqbal

Acknowledgments

Working on my PhD has been one of the challenging and yet most fulfilling experiences of my life. It provided me an opportunity to learn and grow in ways that I could not have done otherwise. It came with its own highs and lows - the journey has undoubtedly been tough but looking back I feel so fortunate to have undertaken it. Each low has been a lesson in moving on and advancing, and I am grateful to all the wonderful people who supported me in this endeavor.

I want to start by thanking my advisor, Professor Martin Herbordt for providing me this opportunity and accepting me into his research group. His guidance and valuable insights have been a driving force for this journey and I am grateful to him for honing me into a computer engineering student. Besides the technical work, he taught me valuable lessons in making slides and presenting my work that I will always value.

I am also immensely honored to have Ulrich Drepper as my mentor. He not only inspired my research journey but guided and taught me throughout these years. I am also grateful to Sanjay Arora and Ahmed Sanaullah for mentoring me and spending numerous hours discussing and debugging complex problems. I started off as a hardware engineer with little interest in computer science and hardly any knowledge of compilers. I am ending this journey as a hardware and software engineer with passion for both fields. I owe this miraculous transition to my amazing mentors.

I would also like to thank my other committee members, Professor Manual Egele and Professor Wenchao Li for their valuable feedback and guidance. I remember hitting a road block particularly in one of the research directions and I am grateful to Professor Egele for teaching me to appreciate my high level goal and regain my focus. I would also like to express my gratitude to friends and colleagues from CAAD lab.

Special thanks to Pouya, Anqi, Sahan, Robert, and Chunshu for guiding me with their experience and helping me flourish in the research environment.

Finally, I want to thank my wonderful family for being my pillars of strength and my friends for always motivating me and celebrating my successes. To my father, Shahzad, for always believing in me and pushing me to reach for the stars and my mother, Arifa, for teaching me the value of hard work and self-efficacy: I am grateful for your unwavering love and support. To my brother, Danish for standing by my side in every high and low. To my parents-in-law, Ammi and Baba, for their consistent affection and motivation. To my kids, Amena and Abdullah, for undertaking this journey with me and being my source of comfort and joy. And foremost to my spouse, Ahmed, for walking this path with me; for motivating me to aim higher and for being my biggest support. I am fortunate and blessed to have you all by my side.

OPTIMIZING THE OPTIMIZER
INCREASING PERFORMANCE EFFICIENCY OF MODERN COMPILERS

HAFSAH SHAHZAD

Boston University, College of Engineering, 2025

Major Professor: Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

A long-standing goal, which is increasingly important in the post-Moore era, is to augment system performance by building more intelligent compilers. One of our motivating hypotheses is that much of the capability needed to advance compiler optimization is already present: state-of-the-art compilers not only provide a large set of code transformations, but also (by-and-large) correctly apply them to preserve the semantics, syntax, and functionality of the code. The challenge lies in getting the compiler to select an appropriate sequence and number of these transformations so as to generate the highest possible performance code based on the developer goals, such as size, speed, or energy.

In this thesis, novel approaches towards automatically generating performant code are developed. In particular we showcase the use of deep learning in building a next generation “smart" compilation pipeline. Deep Learning can fathom complex relationships between code and compilation heuristics, hence providing an excellent tool for optimization tasks. First, we use it to predict a minimum set of optimization options that increase performance on a per-application basis. This is especially useful

for frequently used applications and kernels. For these, developers are willing to spend hours to obtain a few percent performance improvement. Manually developing such heuristics is nearly impossible given the complexity and vastness of the optimization space. This led to two research thrusts. The first is optimizing how transformations and heuristics within a CPU compiler, such as GCC and LLVM, are applied. We note that doing so has the potential to benefit not only code targeting CPUs, but also code which targets hardware, e.g., FPGAs. This is because of the proliferation of High Level Synthesis (HLS), i.e., the use of languages, tools, and techniques that facilitate conversion of a CPU program into a custom hardware design. An efficient and performant CPU code and its underlying intermediate representation is likely to be well suited for translation to a Hardware Description Language (HDL) that programs an FPGA. The second research thrust is automating the pre-processing of the application code, e.g., through the application of directives or pragmas targeting different compilers (GCC, Vitis HLS, Intel HLS) and architectures (CPU and FPGA).

A limitation of above approaches, per-application tuning of compiler heuristics, is that although they guarantee performance improvement, they are time-consuming and lack generality. We therefore use deep learning to find a generalized solution that outperforms the present solutions. First, we develop a neural net based cost function that can accurately predict binary code size for GCC-based compilation. This provides a means to circumvent the cost-computation bottleneck of invoking a downstream compiler to get performance values. This cost function is especially valuable in training models that require a reward in terms of the impact of transformations applied and thus invoke the compiler. Second, we develop pre-trained deep learning models that surpass GCC's default -O_z in more accurately predicting optimal compiler transformations for a given application. Our approach is sufficiently practical to be integrated into the compiler as an -O_{mL} option.

Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application Specific Integrated Circuit
BitW	Bump in the Wire
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DFG	Data Flow Graph
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSL	Domain Specific Language
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
IR	Intermediate Representation
LSTM	Long Short-Term Memory
MLP	Multi Layer Perceptron
MMM	Matrix Matrix Multiplication
NIC	Network Interface Card
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect express
PME	Particle Mesh Ewald
RTL	Register Transfer Language
SIMD	Single Instruction Multiple Data
SpMV	Sparse Matrix Dense Vector Multiplication
ToR	Top of Rack

Chapter 1

Introduction

Demand for high-performance computing (HPC) resources continues to grow across various domains, ranging from scientific research and engineering simulations to data analytics and machine learning. As HPC applications become ever more complex and data-intensive, optimizing their performance becomes increasingly crucial. Even small improvements in efficiency can lead to significant benefits, allowing for more scientific computations, lower resource utilization, and faster time-to-solution. Thus, developers spend significant amount of time optimizing and tuning applications.

An optimized implementation of an HPC application can be an order of magnitude faster than an unoptimized implementation. Generating high performance code, however, requires both expertise and significant programmer effort; improving code optimization methods and support tools is one of the longstanding pursuits in science and engineering. An enduring goal is to automate the optimization process by enhancing already existing compilers, ideally, to enable the compiler to generate efficient code based on the developer goals, which may be code size, speed, or energy consumption. Modern compilers offer default optimization levels (e.g. -O_z, -O_3, and -O_{fast}) based on these goals. However the compiler transformations applied in these default levels are largely "one-size-fit-all" sequences that are often sub-optimal [15, 23, 252]: what works best for one application may not work for another. Despite decades of efforts, there remains much room for improvement [23, 109, 119, 150, 163, 176, 232, 249].

Compilers typically generate sub-optimal code, in part, because (i) they must

make safe decisions, (ii) there are mismatches in programming models, e.g., between that provided to the coder and the target architecture, (iii) the compiler itself lacks knowledge about the target architecture, and (iv) the compilation process to date has inherent limitations, e.g., that it cannot be expected to restructure code to recognize and switch an algorithm. These challenges are exacerbated by the industry-wide trend towards heterogeneous architectures. This requires efficiency across a range of compilers, targeted not only for CPUs, but also other target architectures such as FPGAs that are becoming increasingly prevalent all the way from the cloud to the edge. The space of possible optimization decisions is massive and their impact depends not only on the target application but also the target hardware. This makes the optimization problem a conundrum for naive and expert developers alike: the former to obtain any performance improvement at all; the latter to avoid the immense efforts often required.

Nonetheless, state-of-the-art compilers are remarkably sophisticated, providing a large set of code transformations such as vectorization, dead-code elimination, inlining, and loop optimizations; and also correctly apply these transformations preserving the semantics, syntax, and functionality of the code. In all, a compiler may contain hundreds of transformation passes, many of which are executed numerous times. One of our motivating hypotheses is that much of the capability needed to advance compiler optimization is already present in the compilers themselves: The challenge lies in getting the compiler to select an appropriate sequence and number of transformations that generate high performance code. As we describe, this may be possible through methods that are either internal or external to the compiler.

In this thesis, we ask: *Given an input code for a hardware target, can we automate the prediction of a minimum set of optimization options that increase performance?* Before describing methods for answering this question we first circumscribe the set

of compilers to be targeted. In particular, in this research we specifically target CPU compilers such as GCC and LLVM. We note that so doing has the potential to benefit not only code targeting CPUs, but also code which targets hardware, e.g., FPGAs. This is because of the proliferation of High Level Synthesis (HLS), i.e., the use of languages, tools, and techniques that facilitate conversion of a CPU program into custom hardware design. It has been found that an efficient and performant CPU code and its underlying intermediate representation are likely to be well-suited not only for the CPU, but also for translation to Hardware Description Languages (HDL) that program an FPGA [216].

We pursue four research directions. **The first research direction** is to automate pre-processing of the application code, e.g., through the application of directives or pragmas targeting different compilers (GCC, Vitis HLS, Intel HLS) and architectures (CPU and FPGA). Since the compiler already has immense optimization capability, **the second research direction** is to optimize how optimizations within a CPU compiler (such as GCC and LLVM) are applied; in other words, to *optimize the optimizer*. Both research directions require smart optimization decisions. Manually developing such heuristics, however, is nearly impossible given the complexity and vastness of the optimization space.

Central to these two research directions, therefore, is to augment optimization strategies with appropriate methods in machine learning (ML). In selecting the ML method, we find that supervised ML is not ideal since it requires labeled data mapping workloads to create optimal (or close to optimal) heuristics, which is computationally prohibitive. Unsupervised ML techniques are also problematic as they do not take into account the requirement that a quantity representing performance needs to be maximized. Reinforcement learning (RL), however, may be appropriate: RL represents the problem of maximizing long-term rewards without requiring labeled data

and has been used for such planning problems before [94, 132, 169, 254]. We use RL to determine the optimal heuristics directly for individual codes.

These two research directions deal with automatic tuning, i.e., finding the best possible optimization options on a per-application basis. While this ensures a performance benefit, it lacks generality and is time-consuming. **The third and fourth research directions** extend this work by creating generalized models for optimal heuristic prediction applicable to a broad range of applications. Before we can delve into this process, however, we must address a major limitation: the availability of rich features representative of the underlying code. To this end, we first developed CodeXplorer: a code feature extraction framework for GCC.

Feature extraction for GCC code is not straight-forward: the last major work to extract static features from GCC codes was Milepost [109] and their framework does not work beyond GCC 6+. We tested other alternatives such as a Python plugin by GCC developers [100]. However the limitation of most prior work is that they have not been updated to recent GCC versions and hence do not support vectorization, inter-procedural optimizations, or new architectures. For LLVM, there are many efforts that allow feature extraction including a custom pass within LLVM. There is a need to develop similar feature extractors at different abstraction levels for GCC's latest versions (11-13). Quality features are important since these provide meaningful information for training effective ML models. We have therefore developed a GIMPLE (Intermediate Representation for GCC) parser that allows users to extract more than a 100 function-level static features. In addition, users can choose among types of feature extractions such as (i) After which GCC pass the features should be extracted? (ii) Which subset of features are important? and (iii) What are user target architecture, initial compile flags and target optimization level? The output is a vector of numbers that can be fed as embeddings into a deep learning model.

Building a generalized model requires two things: Efficient features to characterize the given code and performance values to quantify the impact of any optimizations applied. The latter involves invoking the downstream compiler to measure, e.g., the size of the binary, the number of instructions in the object code, or the execution time. **The third research direction** uses the derived features to develop a cost function for GCC. This deep neural net based cost function can reasonably predict code binary size for GCC. Having a cost function speeds up the training time by manifolds and maintains accuracy.

The fourth research direction extends this work by creating a *pre-trained deep learning model* that can (i) surpass compiler default levels such as -Oz in more accurately predicting optimal compiler transformations for a given application and (ii) is sufficiently practical to be integrated into the compiler as an -O_{mL} option. A major limitation of present-day ML models is that there are still many functions whose performance degrades when compared to -O_z. We therefore propose a classifier to be used in tandem with the pre-trained model to reduce such performance regressions. We show that it is possible to reduce such negative regressions by 99%. Our results are sufficiently practical for inclusion of our approach in production compilers.

The thesis is organized as follows:

Chapter 2 gives the broader significance of our work and an overview of the research direction. It also gives an explanation of ML methods used including a deep dive into Reinforcement Learning and background into compiler optimization.

Chapter 3 gives context for the fraction of the target hardware and applications that are not well-known by the broader compiler community, i.e., by looking at various FPGA cloud architectures and analyzing what future clouds will look like. This is part of our publication [76, 222].

Chapter 4 presents the first research direction: our approach to pre-process appli-

cation codes and apply directives and pragmas using RL methods. We also compare hand-optimized solutions and make important observations. This is part of our publication [225]

Chapter 5 combines work from Chapter 4 and integrates it into a framework targeting different compilers and back-ends.

Chapter 6 presents the second research direction discussed above: Applying RL to optimize programs for an HLS back-end. This is part of our publication [223].

Chapter 7 presents the third research thrust we discussed above. We first show how we develop a feature extraction framework to extract code embeddings for training ML models. We then develop a cost function for GCC that is trained using our derived code features and can accurately predict binary size.This is part of our publication [224]

Chapter 8 is the final culmination of our research where we train ML based framework that can outperform GCC default -Oz. We also present a classifier to reduce performance regressors - functions whose performance is better on GCC default -Oz than using our proposed -OzmL.

Finally Chapter 9 gives a summary and future work.

Chapter 2

Background

This chapter serves two purposes. First, it discusses the high level problem addressed in this thesis and narrows it down to the four research thrusts. It also presents FPGAs as a special case study, elaborating on High Level Synthesis that is also one of the focal points in research thrusts one and two. Secondly, it presents an overview of compilers and reinforcement learning, two key principles guiding the research in this dissertation.

2.1 Achieving the 3Ps

Advancements in semiconductor technology have largely been driving the exponential growth in computing power. However, in the past decade, Dennard's law has reached its plateau: scaling benefits derived from silicon no longer apply. Inorder to continue improving efficiency, we need to increase performance from software largely through optimizations. However, 'best-performing' code is subject to interpretation. It might be code with minimal latency. It can be code with highest throughput or energy minimization can be the goal. Each goal requires different optimizations. An optimized code can significantly save effort, time, money and resources.

One of the longstanding goals in software community is to achieve optimized code such that it is not only performant but also programmable and portable. Portability in this case is defined as allowing the user to code in a single language with a single compiler regardless of the target architecture [137]. In this thesis we fix portability

by ensuring all research thrusts require a standard environment (C language). Every research thrust is programmable in that it is automated with minimal efforts required. Programmability refers to minimal code modifications needed in addition to minimal programming effort used. Performance is generally achieving the developer goal: the 'best-performing' code through optimizations.

There are multiple approaches to optimizing code. Manual efforts have been used by skilled developers to target specific inefficiencies in the code. These can apply best practices such as inlining, loop unrolling, dead code elimination and strength reduction to improve code performance. Often users can manually improve memory access patterns or exploit parallelization opportunities such as multi-threading and vectorization. Similar source code restructuring has also been used in the space of FPGAs [216, 304]. In the past developers have experimented with different optimization flags/passes/heuristics either manually or systematically through iterative search to get a 'best optimized' version of the given code. Such techniques target performance at the level of the compilation pipeline. There is a large body of work in this space - including approaches such as design space exploration [106, 159, 162, 297], genetic algorithms [14, 69, 238], random [10] and greedy approaches [77, 140, 197]. There are also dynamic libraries such as SPIRAL and ATLAS [206, 271], designed to optimize for specific applications such as linear algebra and signal processing. Similarly Domain Specific Languages (DSLs) exist that are tailored for achieving performance in specific domains. Examples include parallel frameworks such as OpenMP, MPI and CUDA to extract performance in parallel computing and Tensorflow, Pytorch and Halide for machine learning operations.

In this thesis we focus on achieving performance through the process of efficient translation: source code pre-processing and compiler based optimization. Figure 2·1 highlights these two key ingredients in code translation. Pre-processing is a way to

alter the source code such that it becomes more performant. It includes techniques such as loop unrolling, inlining and architecture specific optimizations using pragmas/annotations. This is the focus of research thrust one and is covered in detail in Section 2.3. Compiler based optimizations are covered in research thrust two in Section 2.4. In section 2.4.1 we introduce compilers in general and how the thesis extracts performance from them.

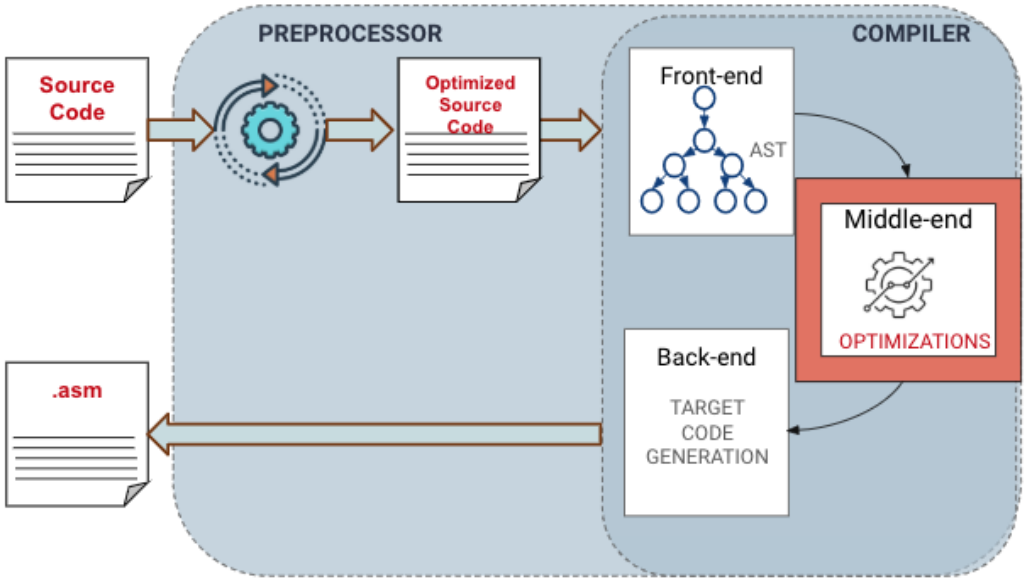

Figure 2·1: Translation from the context of this Dissertation

2.2 FPGAs as Special Case Study

In this section we start with FPGAs as a growing compute resource in the data centers and the cloud. FPGAs are flexible, cost-effective, and reprogrammable which makes them well-suited for high-speed data processing applications.

2.2.1 FPGAs in the Cloud

In the last five years, FPGA presence in the cloud has gone from near zero (except for deeply embedded devices) to a large fraction of all high-end FPGAs sold. This

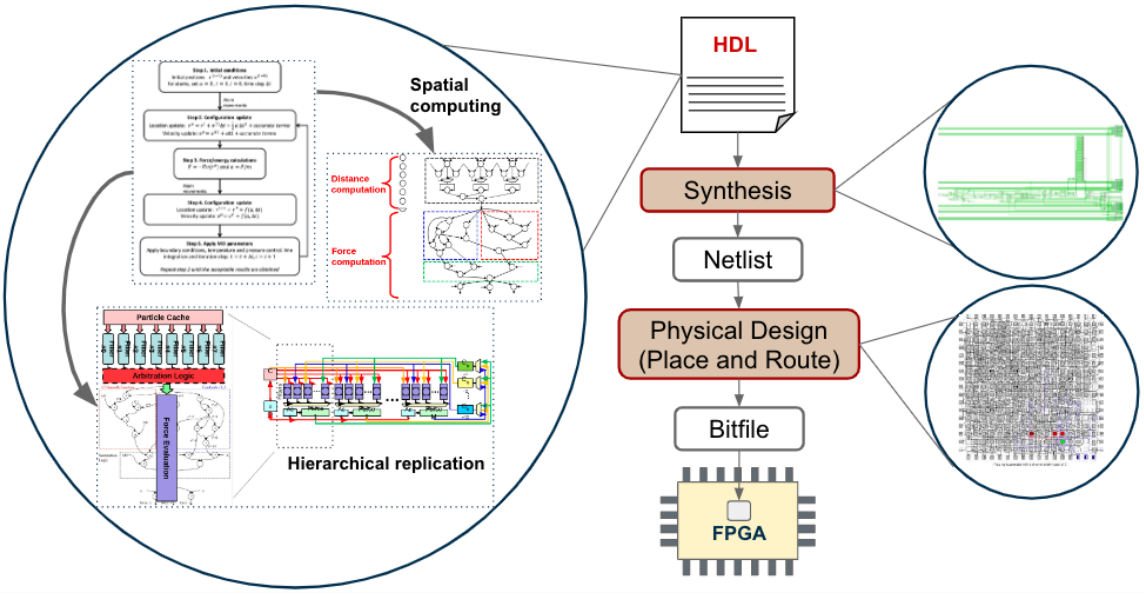

Figure 2·2: Programming FPGA - Workflow

is because FPGAs offer uniquely the performance, power, and flexibility needed to support the diversity and dynamicity of cloud workloads. We begin by observing that, although FPGAs are widespread, they cannot be randomly deployed as part of cloud infrastructure. Any FPGA cloud architecture must satisfy a number of constraints placed by the cloud provider. As a result, FPGA use in the cloud is non-uniformly distributed and motivated by the specific advantages and limitations that each unique architecture offers. We started our research exploring and analyzing trends in existing cloud FPGA architectures that highlight this complex relationship between architectures and system requirements. This allowed us to identify novel architectures that are likely to offer substantial benefits for cloud workloads. The details are in Chapter 3.

2.2.2 FPGAs are Great but Hard to Program!

Despite the proliferation of Field Programmable Gate Arrays (FPGAs) in both the cloud and edge, the complexity of hardware development has limited its accessibility

to developers. FPGAs are exclusively programmed by a cadre of experts. This is because programming hardware is different than programming software. FPGAs are typically programmed using Hardware Description Language(HDL) and each HDL code can take months to develop. Figure 2·2 shows the programming flow. We know CPU code is sequential in nature whereby each instruction is executed one after the other probably in a pipeline with some degree of parallelism. However, FPGA execution model is spatial. An FPGA executes all instructions at the same time by mapping it to hardware spatially hence allowing for far greater levels of parallelism. An FPGA can replicate function units as long as there is room on the device. This goes through another layer of abstraction to Synthesis where behavioral representation is converted to gates and logic. As a next step it undergoes Physical Design where hundreds of thousands of LE are connected with millions of wires depending upon the resources of the FPGA. Depending upon the complexity of the code, a good design can take months to develop.

Some research tangential to the present work explores another difficulty of deploying FPGAs: the need to be able to automatically generate a hardware operating system appropriate for target FPGA and application [62, 63, 65, 213, 244].

2.2.3 High Level Synthesis and its Challenges

To address the programmability issue, one solution is to add another layer of abstraction on top of HDL which is easy to program, portable and familiar.

This is illustrated in Figure 2·3. The attempts to get from sequential code to hardware is called HLS. This offers a considerable advantage by enabling complex hardware designs using procedural languages. In Figure 2·6 we briefly look at how HLS works. At front end it takes a C code and converts it to IR, hence allowing all subsequent steps to use language independent format. We apply optimizations on top of this IR in order to optimize for area, latency, parallelism, reduce power/energy.

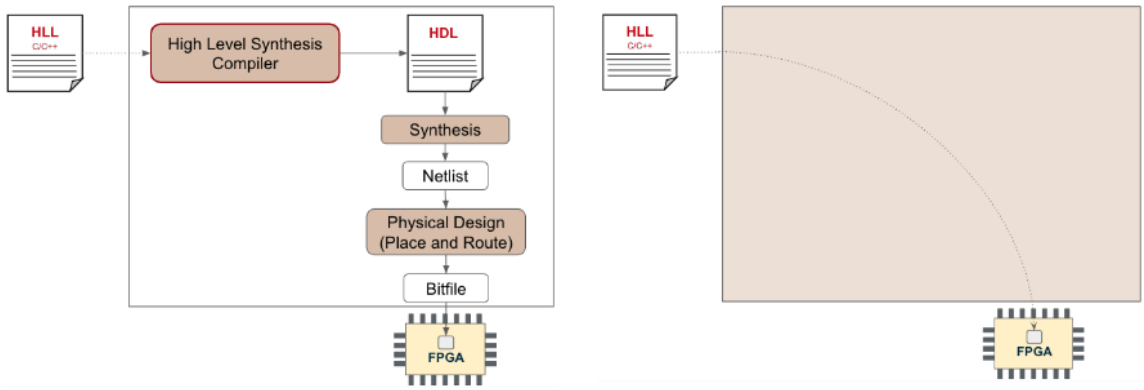

Figure 2·3: High Level Synthesis as a Blackbox

At the back-end operations are mapped and scheduled and resources are assigned according to the particular FPGA chip. This looks like a compilation and indeed HLS is like a compiler.

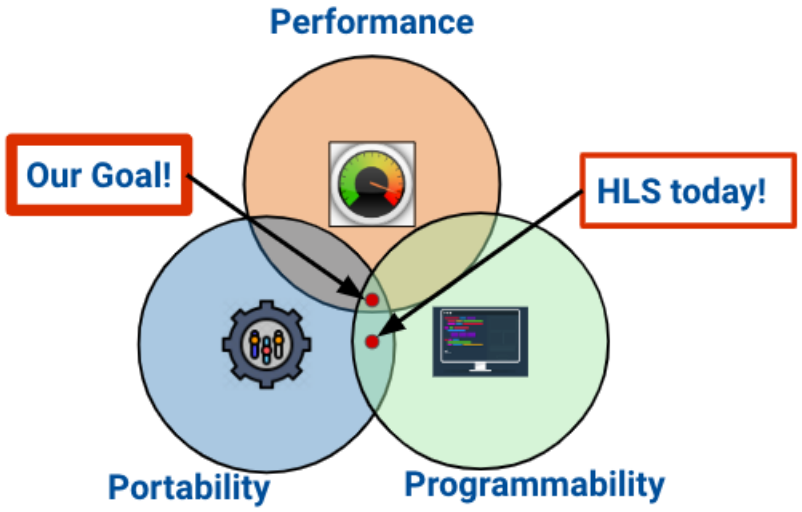

Figure 2·4: High Level Synthesis and the 3Ps Dilemma

However, while High Level Synthesis (HLS) offers a possible solution by automatically compiling CPU codes to custom circuits, it currently delivers far lower hardware quality than circuits written using Hardware Description Languages (HDLs). We illustrate this in Figure 2·4. One of the longstanding goals in the software community

is to achieve all 3Ps that is achieve performance, portability and programmability. While HLS does deliver Programmability and Portability, the Performance is relatively bad. In prior work from our group [214, 288] we have observed that OpenCL used for coding nearest neighbor computation on FPGAs can deliver 6-64× worse performance at the expense of up to 7× more resource consumption. Similar observations are also made by other researchers [17]. Other work from our group showed how to enhance existing vendor tools to help automate these methods [215, 217].

Our findings at CAAD lab BU have shown that you can get good performance through HLS using source code transformations: that is vary C code to match the code optimization strategy of the compiler. In [216], authors observed that we can vary the way code is written in order to suit the style of the specialized back-end. Source code tuning has also been explored in other work [102]. However, a drawback of this strategy is that it is not portable. These methods are empirically guided and compiler dependent. Moreover, it requires significant developer expertise in hardware. One way in which we have proposed to solve this challenging problem is that instead of changing the source code to match the compiler strategy how about we tame our compiler according to the application. Our observation is that compilers can work we just have to find the magic spell that enables optimizations to run optimally.

2.3 Pre-processing Source Code

Pre-processing source code is an approach to guide code generation by introducing constraints or additional information. These tell the compiler about possible optimization opportunities in the source code. However, using annotations/pragmas is challenging. We ask: Can we achieve performance but at same time not lose programmability and portability? This is the focus of research thrust one and addresses both FPGAs and CPUs. Below we present this problem of pre-processing source code

as a special case study looking at HLS and extend this to CPU codes.

2.3.1 Modifying the HLL code to become Target-Specific

An approach that we have employed to improve the state of HLS is to generate a target specific HLL: write code in a familiar high level language like C but restructure for the specific target. This can be done by inserting annotations/compiler hints in the existing HLL code at appropriate places. We have explored this for FPGA back-ends on state of art Xilinx Vitis and Intel HLS tools in Chapter 4. We have also extended this work to CPU backends and come up with a generic framework that handles these annotations end-to-end: the user feeds in input HLL code and at the output gets an annotation-inserted optimized HLL code. This is given in Chapter 5. Below we discuss the space of this research direction, what ha been typically done in the past and how our work fits in.

CPU code annotation Traditionally annotation libraries have been used to grant parallel semantics to code syntax that is originally written to run sequentially; examples include OpenMP, OpenACC, OpenHMPP, and OpenSs. A number of existing tools [21, 25, 99, 182, 209, 237] target automatically inserting OpenMP pragmas into source code. These tools have their own caveats: the output generated varies in format, i.e. source to source versus binary; pointer aliasing hinders parallelization; and additional checks are needed to ensure dependencies are avoided. Loop parallelization strategies also differ: Cetus [99] deals with multi-core parallelization while DawnCC [182] targets GPU threading. Modern compilers support many more annotations in addition to standardized ones supported by these tools. These additional annotations also address some challenges faced by these tools and are explored in this work.

There are also frameworks exploring the space of loop transformations (on Clang employing LLVM's polyhedral loop nest optimizer Polly [159, 162, 280]) and optimiz-

ing loop vectorization parameters inside Clang's intrinsic pragmas [132]. Our work is novel since it explores a broader set of annotations, provides options to turn on certain compiler optimization flags [114], and supports instruction set extensions [87].

FPGA code annotation

Much research in annotations for HLS has focused on design space exploration. [106, 236, 241, 279, 292, 297] introduced heuristics and model-based approaches to sample the design space. We extend this work by also supporting automatic insertion of annotations into the source code, thereby creating an end-to-end flow. Also, supervised and reinforcement learning techniques are promising since these may reduce the search cost by using past data to drive optimization decisions [94]. Several studies identify regions of interest, or predict optimal factors for loop-based HLS directives, such as unrolling or pipelining. A major contribution of our work lies in creating a massive search space for multiple state-of-the-art HLS compilers and deriving performance also by leveraging HLS directives for optimizing functions and arrays.

Authors in [19] have explored code annotations for data-flow programming in streaming applications that have multiple kernels within a single code. The source-to-source compiler, SpecHLS [120] targets a subset of HLS kernels that can benefit from speculative pipelining strategies. [235] present an optimizer for design space exploration within the Merlin compiler. Others, such as AutoScaleDSE [152], use their own quality-of-results (QoR) estimator to provide performance estimates and optimize for loop tiling and array partitioning.

In short, several studies have explored automated code annotation; they are advanced in our work through extensibility and generality, which also results in improved performance. Our work as discussed in Chapters 4 and 5 supports additional compilers, types of computational hardware, different ML-based optimizers such as Reinforcement Learning (RL) and Bayesian Optimization (BO) for code annotation,

and is able to explore a massive search space of possible options. These features combine to yield codes with substantially improved performance, including over previously optimized code.

2.4 Modifying Compiler Strategy

In this section we explore the second part of effective translation: modifying teh compiler strategy to make it more performant for the developer goal. We start with a brief introduction of compilers and extend it to our second research thrust.

2.4.1 Compilers: A brief introduction

A compiler is a specialized software program that translates a source code written in high level language (HLL) into a semantically equivalent and functionally correct program written in a target, lower-level language that a computer can understand and execute on the underlying hardware, such as machine code or assembly language. Figure 2·5 gives the overview of its compilation process. Compilers typically have three modules: front-end, middle-end, and back-end. This type of hierarchical structure enables modularity: the front-end and back-end can be quickly adapted to incorporate newer languages and architectures while the middle end that is language and hardware independent can be reused.

The Front-End analyzes and validates the input code. In the start there is the lexical analysis or scanning that breaks the source code into tokens (such as keywords, symbols, variables etc.). Next the parser runs the syntactic analyses to generate the abstract syntax tree(AST) that adheres to the language's grammar rules. It makes sure the code is in valid 'form' and 'format'. After that the semantic routines ensure the code logic is correct and that the operations defined within it make sense. After that the intermediate representation of the code is generated that is all input

languages and platforms independent. For GCC this is the *GIMPLE* code that we have analyzed both to generate features defining the code and also for optimizations.

The Middle-End of the compiler optimizes the intermediate representation of the code according to the defined flags/criteria/goals/optimizations. The Back-End maps and translates the optimized intermediate representation of the code into hardware-specific machine code, often in the form of object code or binary code..

We are proposing changes within the middle end of the compiler to enhance performance of the code so it can execute better on the target hardware and also include 'smart decision-making'. Compiler optimizations are passes/flags each focusing on specific aspects of code improvement. Some examples include dead code elimination, loop unrolling, vectorization etc. LLVM allows around 150 transformation passes while GCC has more than 250 flags. It is quite possible that a certain optimization enhances performance on a certain code and worsens on another. The ordering of applied optimizations also impacts performance. Hence LLVM has an optimization space of 150^{L} where L is the pass sequence length while GCC has a space of 2^{250}.

Exploring such a massive optimization space is a major challenge. We have addressed this in this thesis using tools from machine learning, in particular Reinforcement Learning.

2.4.2 Optimizing the Compiler Strategy

As seen in Figure 2·6, we can optimize codes for HLS by applying appropriate optimization passes/flags at the middle end of a CPU compiler. Compiler optimization problem space for HLS is vast. Hardware quality is affected not only by the high level program but also by several other factors such as pass ordering, the number of passes and their parameter values. CPU compilers such as GCC and LLVM offer default optimization levels such as -Oz, -O3 to optimize code for specific goals. However, optimization strategy defined by -Ox levels is rather static and does not take

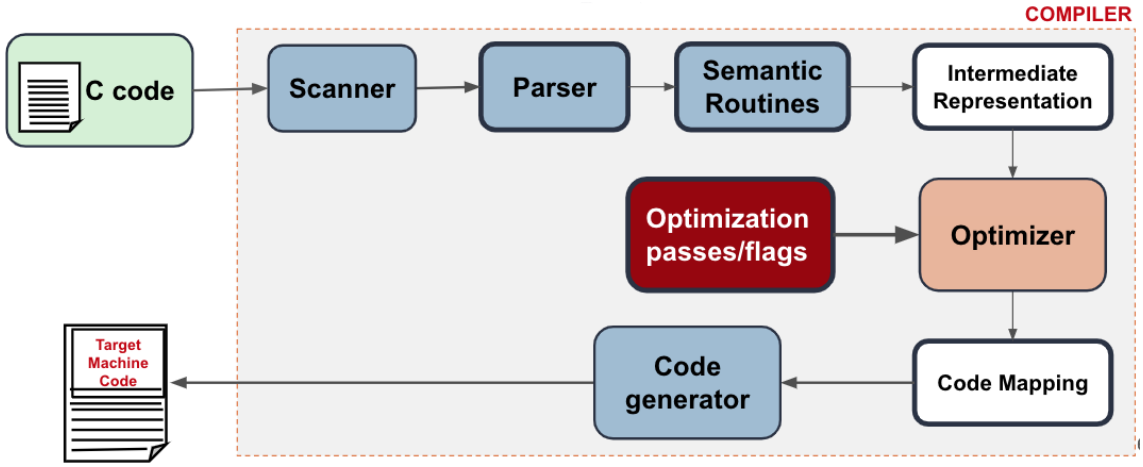

Figure 2·5: Compilers - an Overview

the problem nor really the variety of the target hardware into account. Hence it is not optimal for other specialized back ends like FPGAs. For simple applications like matrix multiply on an FPGA back-end, we have seen that with appropriate number of passes and their ordering, more than 60% improvement in performance over the standard -O3 is possible when evaluated using Legup[168].

In order to bridge the gap between hand tuned and automatically generated hardware, it is thus important to determine the optimal pass ordering for HLS compilations, which could vary substantially across different workloads. Since there are dozens of possible passes and virtually infinite combinations of them, manually discovering the optimal pass ordering is not practical. Instead, we will use reinforcement learning to automatically learn how to best optimize a given workload (or a class of workloads) for FPGAs. Specifically, we investigate the use of reinforcement learning to discover the optimal set of optimization passes (including their ordering and frequency of application) for LLVM based HLS - a technique for compiler tuning that has been shown to be effective for CPU workloads. Among all the work on compiler tuning, [141] is the only state-of-art framework, to the best of our knowledge, that addresses the problem particularly using High Level Synthesis for specialized back-

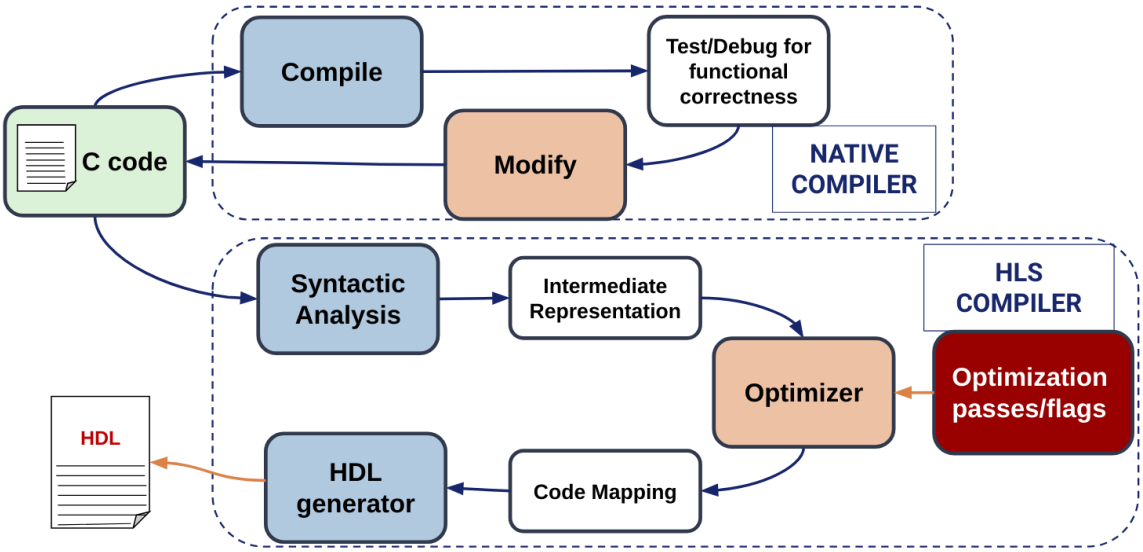

Figure 2·6: Native versus HLS Compiler

ends such as FPGAs. Our works builds upon it to improve the existing approach and explore further possibilities.

Existing efforts in RL based compiler tuning have: i) only explored a small number of learning strategies, and ii) evaluated the impact of these strategies using a single metric for learning quality i.e. speedup over *-O3*. This is a significant drawback since learning goals can vary substantially across workloads and developer requirements. For example, developers can prioritize lower turnaround times over largest speed up values. In such cases, a different learning strategy would be needed which is able to trade off achieved speed up for a faster learning rate. Thus, similar to a uniform optimization strategy, a generic learning strategy is also inefficient. We have addressed these limitations by exploring in our work and beyond, how compiler tuning is impacted by different strategies in reinforcement learning for HLS. We study how selection of observation space, action space, training agent and the reward computation frequency impacts the overall learning rate and the final reward. This work is discussed in Chapter 6.

2.5 Smart Optimization - Building a Generic Strategy

In this section we discuss our third and forth research thrust - building a generic strategy that can predict performance heuristics across a broad range of applications and functions.

Compilers offer default optimization levels (such as -O_z, -O_3) that specialize in generating high performance code based on various developer goals such as size, speed, or energy. As prior work has shown, however, the code generated by these default levels is often suboptimal leaving substantial room for improvement. We ask: Given an input code for a target hardware, can we predict a minimum set of optimization options that improve its performance beyond these default optimization levels? We have explored this earlier in section 2.4.2 by employing optimizations on a per-application basis. While this guarantees performance, it is time consuming and lacks generality. A preferred solution is one where a *pre-trained deep learning model* can (i) surpass compiler default levels in more accurately predicting optimal compiler transformations for a given application and (ii) is sufficiently practical to be integrated into the compiler as an -O_{mL} option. Achieving this solution is the final culmination of this thesis.

In our initial work for research thrust four, we set up a basic framework to train an RL based model across multiple single function applications and use it to predict heuristics for a limited subset of test applications. We established that performance gains over -Oz are possible: On a set of 78 applications, it is possible to decrease binary size above -Oz by a mean of 20.5%, number of instructions by 14% and number of GCC passes applied by 19%. As final work done for this dissertation, we expand along these lines and increase the size of training dataset and expand it to include multi-function, popular benchmark suites. We evaluate various techniques such as reward shaping, normalization, action space pruning to increase training accuracy of

the model. We also incorporate application classification to filter out functions whose performance improvement seems improbable. Our final goal is to achieve a single model that can (largely) predict performance improving optimizations across a broad range of source code functions. We present this work in Chapter 8. Our results show that it is possible to optimize more than 60% of the functions and achieve an average of 8.37% reduction on top of GCC 13.2's -Oz. Once results are achieved on a broader application set, such a pre-trained model can be integrated into GCC as an -O_{mL} option and predict flag options for developers that give a far smaller memory footprint than output by GCC's present -O_z option.

One limitation of training models for AI is the training time particularly caused by invoking the downstream compiler in evaluating performance values. This is the motivation behind our research thrust three and covered in Chapter 7.

2.6 Our not-so-secret Recipe: Artificial Intelligence

Developing effective translation process requires an intelligent strategy. This work explores utilizing artificial intelligence(AI) in driving optimal choices. AI is a popular choice in the research space. We drive our differentiation by asking: Can we increase performance over the state-of-art without compromising portability and performance? Is our approach practical and flexible? Is it extensible and modular? The specific novelty of each research thrust and how we place our work in the overall landscape is covered in each relevant chapter. Here we discuss background of the approach used and introduce Markov Decision Process and its implementation as Reinforcement Learning.

2.6.1 Markov Decision Process

In this thesis we have framed the problems of pre-processing source codes and modifying compiler strategy in the context of Markov Decision Process (MDP): a stochastic

process that model decision making in fully observable environments. It is defined as a tuple (State, Action, Transition Probability, Reward function, Discount Factor). In a typical MDP, an agent is the 'learner' that learns through its decisions and interactions. An environment is a demonstration of the 'problem' in which the agent interacts. State defines the set of possible situations/configurations/positions of the agent within the environment. Actions are the decisions that the agent makes and through which it perturbs the environment to learn. Transition Probability is the probability or likelihood that the agent transitions from one state to another. Reward is the value that the agent gets after performing a certain action in some state of the environment. Discount factor is a means to qualify how much importance has to be placed on immediate versus future rewards. MDPs are based on the Markov Property which states that the future state depends only on the current state and action, and not on the past states' sequence. Solving an MDP essentially means to find a policy or strategy that chooses an action based on the current state and optimizes for the overall reward/outcome.

2.6.2 Reinforcement Learning as an MDP

Our approach for solving MDPs is utilizing Reinforcement Learning (RL): an approach in which an agent learns through its interactions with the environment. It obeys the Markov property in that the current state is defined to capture information of the past states. Intuitively meaning that the current state and action are only required to define the future state.

The ultimate goal in RL is computing a mapping between the environment and actions: a policy that is maximizing the expected reward. In doing so, it relies on a trial-and-error mechanism in order to learn, as opposed to labeled data in supervised learning. Figure 2·7 summarizes its workflow. In the context of RL formulated as an MDP, there are two major components to the overall framework:

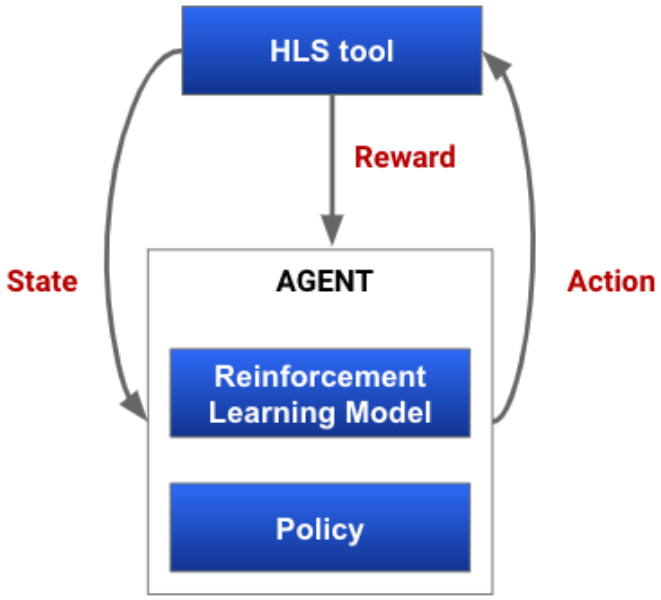

Figure 2·7: An Overview of Reinforcement Learning

i) *Environment*: the problem that we are trying to learn to solve, and

ii) *Agent*: used to perturb the environment and learn based on feedback.

The smallest unit of *Environment-Agent* interaction is typically a *Step*. At each *Step*, the *Agent* predicts an *Action* that the *Environment* should take e.g. the next move in a game of chess. After taking the action, the *Environment* returns a *Reward* indicating the impact of the *Action*, as well as an updated *State* which represents the change in the *Environment* as a result of taking the *Action*. The next *Step* then starts and a new *Action* is predicted.

The above process continues till the *Environment* indicates that a conclusion has been reached e.g. the game of chess has ended. This collection of *Steps*, from the first predicted action *Action* to the *Action* that causes the *Environment* to reach a conclusion, is referred to as an *Episode*. At the end of an *Episode*, the *Environment* is reset and a new *Episode* starts. A collection of such episodes is referred to as an Itera-

tion. The agent is updated once per iteration. The number of *Episodes* per *Iteration* can vary significantly based on a number of factors. At the end of each *Iteration*, the learning portion of a RL framework updates a *Policy* (deterministic/stochastic strategy) about which *Actions* cause *Agents* to maximize their long-term, cumulative rewards. RL assumes that the environment is Markov i.e. that the updated state also depends on the previous state and the action taken. It also assumes that the action taken is only dependent on the current state.

There are three types of RL implementations: policy-based, value-based, and model-based. Policy-based RL involves coming up with a *Policy* to maximize the cumulative *Reward*. Value-based RL attempts to maximize an arbitrary value function, V(s). Model-based RL is based on creating a virtual model for a certain *Environment* and the *Agent* learns to perform within the constraints of the *Environment*.

In each subsequent chapter, we discuss in detail how the RL framework is set up as an MDP. We also discuss various components involved in the decision making process and how the process is validated to ensure optimal performance.

Chapter 3

Survey and Future Trends for FPGA Cloud Architectures

3.1 Motivation

The demand for data center services is likely to continue growing exponentially [177]. As Moore's Law has slowed and the computational overhead and complexity of cloud workloads continue to rise and evolve, FPGAs offer a promising mechanism to address the paradox of simultaneously combining performance, power, and flexibility. They can be deployed almost anywhere in the data center in order to accelerate compute, storage, and network, as illustrated in Figure 3·1. Due to the immense benefits of FPGAs, their use in data centers is expected to grow at a Compound Annual Growth Rate (CAGR) of 48% between 2020 and 2027 [117].

While FPGAs offer a number of benefits in the data center, the choice of which particular architecture to leverage is complex and non-trivial; we simply cannot place any number and size of these devices anywhere in the cloud. The extreme need for cost-effectiveness leads to emphasis on size, power, cooling, compatibility, automation, and in-place upgradability, all while ensuring that specific needs of cloud workloads are met in terms of performance, memory, server capability, reliability, security, and network connectivity. For example, as illustrated in Figure 3·2, architectures can offer a different set of key benefits because of FPGA placement and connectivity. Thus, based on a system's requirements, certain FPGA cloud architectures can be

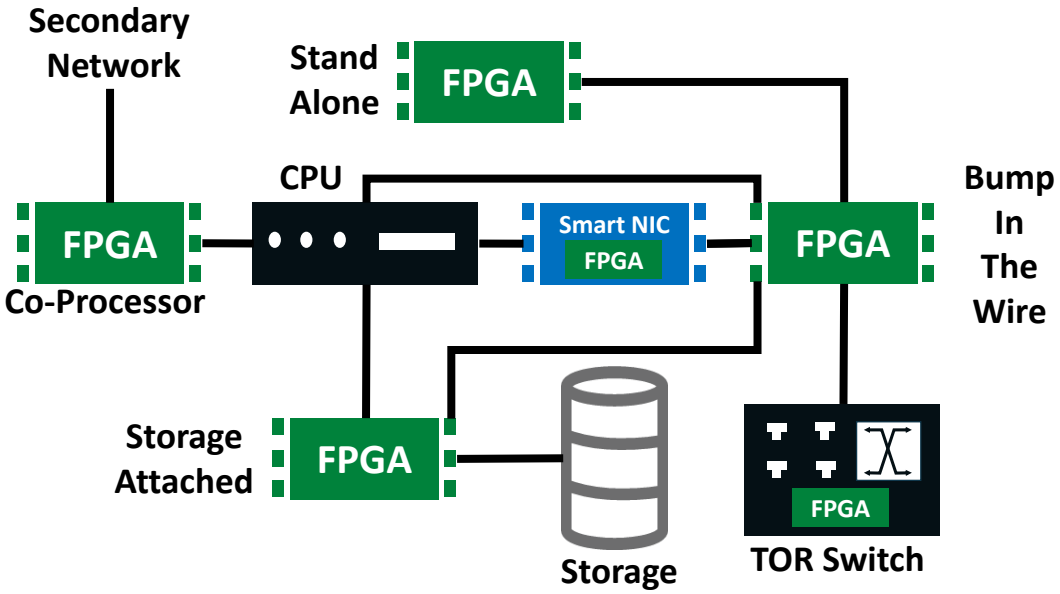

Figure 3·1: Deployment versatility of FPGAs in the data center

substantially more advantageous than others.

Given the non-trivial relationship between system requirements and architectures, surveys of FPGA cloud architectures can play an immensely important role in helping deliver on the promise of FPGAs in the cloud. Specifically, these surveys can highlight the key advantages and limitations of individual architectures, which in turn make it easier to describe and compare systems without getting bogged down by low level implementation details. These surveys can also provide an 'innovation guide' to the vendors, including FPGA chip/board manufacturers and FPGA cloud system architects, about the features and limitations critical to their customers. Moreover, if a new system is to be brought online, these types of surveys can help reverse-engineer the best suited architecture based on a given set of requirements and constraints.

While there are several surveys that analyze different facets of cloud FPGAs [75, 86, 90, 153, 156, 160, 179, 187, 247, 250, 253, 256, 261, 267, 291, 293, 294, 296], currently there is less work that addresses the architecture space. Relevant prior

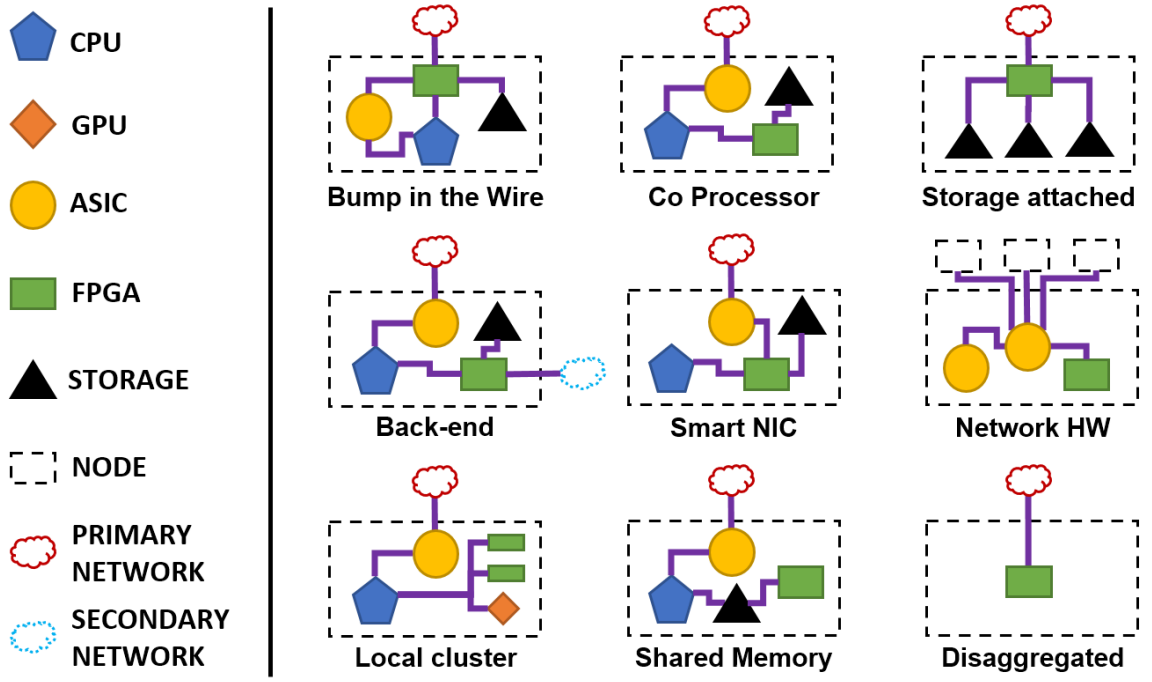

Figure 3·2: Examples of common FPGA architectures. Potential benefits for each architecture include: i) **Bump-in-the-Wire:** Large scale compute, network and storage acceleration, ii) **Co Processor:** Local compute acceleration, iii) **Storage attached:** Local storage acceleration, iv) **Back-end cluster:** Ultra low latency, rack scale FPGA-FPGA communication, v) **Smart NIC:** Local network acceleration, vi) **Network HW:** Flexible routing/switching protocols, vii) **Local Cluster:** Multi-accelerator system, viii) **Shared Memory:** Cache coherent acceleration, and ix) **Disaggregated:** High infrastructure utilization.

work is primarily discussions in support of specific technical contributions [55, 80, 234, 268, 303]. These are generally brief and based on broad categories and assumptions that do not capture all of the unique set of advantages and challenges of different architectures. For example, the taxonomy in [80] captures FPGAs as in-node CPU compute accelerators, but does not extend to a number of other configurations, such as computational storage devices [36, 39, 104], stand-alone devices without a CPU in the node, or devices deployed outside of nodes, e.g. in TOR switches [32, 193].

The goal of this paper is to discuss cloud FPGA architectures with sufficient depth such that the relationship between design choices and constraints can be analyzed and

trends can be identified, but while abstracting away low-level implementation details such as specific chips/boards and communication protocols. To do so, we survey existing cloud FPGA deployments in the context of the following central questions:

- **Type of FPGA board:** Are off-the-shelf boards used, or are the requirements specific or strict enough that a custom board is needed?
- **Placement of FPGAs in the system:** What type of components have FPGAs? Are they shared?
- **Network connectivity:** Do FPGAs have direct access to any inter-node networks?
- **Intra-node connectivity:** What significant devices can an FPGA talk to within a node? How?
- **Use cases:** Who is using the FPGAs (provider, user, etc.) and for what workloads?

Since the answer to the above can vary substantially based on the strictness of the constraints, we separate the analysis into production and research architectures. This allows us to both derive meaningful and valuable trends, as well as to plot a possible road map for cloud FPGA architectures, since research systems naturally point to future production systems. We also highlight promising, novel areas of innovation in the cloud FPGA architecture space that are currently not part of any production or research work, but may offer substantial value.

The specific contributions are:

- We survey existing cloud FPGA architectures deployed in both production and research systems.

- We classify these cloud FPGA architectures and use this classification to analyze important trends.
- We highlight several promising areas of future innovation in the cloud FPGA architecture space.

More generally we expect this work to serve different purposes to different communities, with the survey and references providing an introduction to FPGAs-in-the-cloud to non-experts and the taxonomy and predictions being useful (or at least provocative) to practitioners.

The rest of this chapter is organized as follows. Section 3.2 presents a taxonomy for classifying cloud FPGA architecture. Based on this taxonomy, Section 3.3 discusses existing production cloud FPGA architectures. Then, Section 3.4 extends this discussion to cover existing research systems. Section 3.5 highlights potential future innovation derived from the taxonomy. Finally, Section 3.6 gives the conclusion.

3.2 Taxonomy

The taxonomy is based on the critical questions highlighted in the previous section: **A) type of FPGA boards**, **B) placement of FPGAs in the system**, **C) network connectivity**, **D) intra-node connectivity**, and **E) use cases**. We now discuss this taxonomy in more detail and highlight both the above categories, as well as the different sub-categories that can exist within each. Note that these sub-categories are neither mutually exclusive nor comprehensive: it is possible for system to have multiple sub-categories, and for new sub-categories to be added later to encapsulate future innovation.

3.2.1 Type of FPGA boards

Given that cloud providers are not currently creating their own FPGAs, the smallest unit of differentiation is the FPGA board; both **a) Off-the-shelf** and **b) Custom** are widely used. Economic advantages depend on scale of deployment and provider development infrastructure. Given the latter, custom boards still have higher start-up and upgrade costs, but may be cheaper in large quantities. But the advantage of scale also affects off-the-shelf economics as the provider has the market power to affect price and board features.

With custom FPGA boards just about any attribute can be varied, such as number/types/bandwidths of I/O ports, FPGA family, off-chip memory type and size, form factor, and other on-board devices. This ensures that the boards closely match the specifications/requirements of the target system, from computation to cooling. None-the-less, off-the-shelf boards are available for every (currently) sizable usage domain, including SoCs [33, 40], node-level networking [53, 54, 58, 147, 180, 192, 231], NoCs [8], data center switches [31, 32, 193, 220] and storage [28, 36, 39, 48, 104].

3.2.2 Placement of FPGAs

FPGAs can be placed in either a **a) distributed** or **b) centralized** manner. Having a distributed FPGA placement means that compute/storage nodes have their own FPGAs, and thus do not have to compete for the resource. This leads to more offload capability, greater reliability (since FPGA failure does not affect on other compute/storage nodes), and reduces security concerns since offloads for different nodes can be isolated. It is also possible to place FPGAs in a centralized manner, typically inside the networking nodes (e.g. in switches as ASIC-FPGA, CPU-FPGA, or FPGA only circuits). Substantially fewer FPGAs are needed for such a deployment; this typically translates to easier management, lower power consumption, lower TCO,

smaller average node sizes, and potentially higher performance since expensive high-end FPGAs can be used (and upgraded more frequently).

3.2.3 Network connectivity

Within each node, it is possible for FPGAs to be **a) not connected to any network** or connected to **b) the primary network** and/or **c) a secondary network**. Being connected to the primary data center network enables FPGAs to intercept/accelerate network traffic to the node, as well as achieve data-center-wide scalability for FPGA workloads since multiple FPGAs can directly communicate with each other. However, the circuitry needed to support this FPGA position can consume a significant portion of FPGA resources. This includes circuits to support high resiliency since the FPGAs can be a single point-of-failure: an entire node can become unstable in the case of an FPGA error. In the case of secondary network connectivity, FPGAs can communicate across nodes with significantly more flexibility in the topology used (e.g. mesh, torus, switched), as well as the communication protocol, all of which can lead to ultra low latencies. However, using a custom network configuration means that complex router hardware, routing algorithms, and switch arbitration policies may need to be implemented on each FPGA. Moreover, complex cabling may be required, which can add a significant burden to the overall data center architecture [80].

3.2.4 Intra-node connectivity

FPGAs within a node can be **a) not connected to any other significant device** (i.e. be a *Disaggregated* resource) or connected to one or more devices: **b) CPUs**, e.g. through PCIe and possibly with cache coherence using interconnects such as CCIX [243], CXL [97], or CAPI [195]; **c) Other FPGAs**, e.g. through a PCIe switch and/or using direct and programmable interconnects; **d) GPUs**, e.g. through a PCIe switch; **e) ASICs**, e.g. through multiple potential forms of connectivity

depending on the ASIC and nature of coupling such as NIC or tensor processor; **f) Storage devices** through the device-specific interface, e.g. SPI for flash and DDR for SDRAM.

3.2.5 Use cases

Use cases have a substantial impact on architecture, since cloud providers must ensure workload requirements are met (e.g. performance) without compromising on core aspects (e.g. security, reliability). Common cloud FPGA usage domains are the following. **a) Customer applications:** Customers can develop, simulate, debug and compile their custom FPGA logic, as well as scale their infrastructure and change resources according to their workload demands. A wide pool of applications can be deployed, e.g. in genomics, financial analytics, computational fluid dynamics, video processing, transcoding, and security. Several development environments are available so users do not need to write their own HDL code [66]. **b) Provider Application as a Service (AaaS):** The cloud provider supports a limited set of customer applications by developing the FPGA design themselves: only the necessary APIs and high level design parameters are exposed. This model ensures high performance and resilience at the expense of reducing customer access to the entire FPGA. **c) Provider applications:** Cloud providers use FPGAs to accelerate their internal workloads, e.g. SDN, as well as save CPU resources that can then be rented to the customer.

An especially promising set of use cases leverages the common positioning of the FPGAs within the node (network facing) and the tight coupling of communication and application logic. These include privacy preserving computations [200, 201, 272] and certain computations where strong scaling is especially challenging, such as Molecular Dynamics [64, 273, 274, 275, 276] and Docking [239, 240, 257, 258, 259].

3.3 Production Architectures

In this section we discuss production cloud FPGA systems that are in widespread or large-scale use.

3.3.1 Overview

Perhaps the most widely deployed production system is Microsoft's unique Catapult v2 [80], which has FPGAs in most Azure and Bing SKUs in a *Bump-in-the-Wire* configuration: FPGA sits between the TOR, NIC ASIC and CPU, hence enabling data-center-wide communication within tens of microseconds of latency. These hundreds of thousands of FPGAs (or more) are used for both internal (e.g. network packet processing [186], Bing search [86]) and external workloads (e.g. Machine Learning inference as a service [86]), with HPC workloads also found to be plausible [227, 228, 230].

Another type of a widespread production system is the single node accelerator model, which leverages FPGAs in either a *Coprocessor* configuration, or as a *Local Cluster* where devices are interconnected either via a PCIe switch or using direct FPGA-FPGA interconnects. A number of cloud providers such as AWS [3], Huawei [45], Baidu [27], Tencent [37], Nimbix [34] and Alibaba [26] use this model. These systems are used by customers to run a wide pool of cloud native applications such as genomics, financial analytics, data acquisition, computational fluid dynamics, video processing, image processing, transcoding, security, and AI workloads [41, 50, 51, 52, 56, 251]. There are also examples of these FPGAs being used by providers for their own workloads. Baidu uses FPGAs to accelerate its cloud-based storage, SQL queries, data security, search engine, and AI workloads [41, 47]. FPGA-based AI chips–such as Baidu's Kunlun for AI, Alibaba's Ouroboros for speech recognition, and Alibaba's Hanguang 800 for inference operations–are deployed in their cloud data centers [105]. Alibaba has reported 75% savings in TCO by using FPGAs to oversee product images

on its e-commerce site [283]. In 2018 it reported over $30 billion retail on its website in a single day (compared to $5 billion on all US online and in-store retail on black Friday 2017); this was possible with its data center FPGAs being used to accelerate transactions and provide recommendations to users [107].

There are also systems that are widely deployed, but where there is insufficient publicly available information for analysis. Amazon has announced AQUA (Advanced Query Accelerator) nodes for its Redshift data warehouse, available through the RA3.16XL and RA3.4XL instances. These nodes use FPGAs to accelerate dataset filtering and aggregation [29, 49]. Baidu uses Smart NICs to improve virtualisation and workload performance [59]. OVHCloud also uses Smart NICs, but for network packet processing to mitigate Distributed-Denial-of-service (DDoS) attacks in its cloud traffic [35, 42]. Scaleflux CSD2000 SSDs are deployed by over 40 data centers globally [30]. An example is the Alibaba cloud, which uses Scaleflux CSD20004 in place of traditional SSDs on their storage nodes to accelerate applications such as MySQL, Aerospike, Oracle, and PostGreSQL [46]. Samsung Smart SSDs [39] are deployed at the Nimbix cloud where they accelerate Apache Spark, running queries up to 6x faster when using software from Bigstream [38]. Eideticom's computational storage processor [104] has been implemented in Barreleye G2 servers on Rackspace [205].

3.3.2 Architecture Trends

Figure 3·3 classifies production cloud FPGA architectures into seven taxa (using the taxonomy in Section 3.2). To effectively compare these architectures and highlight trends, we avoid illustrating the taxonomy as a single tree. Overall, there are four major trends: 1) Boards, 2) Placement, 3) The relationship between network connectivity and use cases, and 4) Intra-node connectivity.

a)

Type of Board	
Custom	Off-the-shelf
[A], [B], [C2], [F], [H]	[N], [T]

c)

Network / Use	Consumer	PaaS	Provider
Primary		[C2]	[C2]
Secondary			
None	[A], [B], [F], [H], [T]	[N]	[A], [B], [H]

b)

Placement	
Centralised	Distributed
	[A], [B], [C2], [F], [H], [N], [T]

d)

Intra-node Connectivity				
FPGA	CPU	GPU	ASIC	Storage
[A], [F], [H]	[A], [B], [C2], [F], [H], [N], [T]		[C2]	[A], [B], [C2], [F], [H], [N], [T]

Figure 3·3: Classification for the following production cloud FPGA architectures: Alibaba [A], Baidu [B], Microsoft Catapult v2 [C2], Amazon AWS F1 [F], Huawei [H], Nimbix [N] and Tencent[T].

Boards

Figure 3·3a shows that a majority of vendors have used custom boards in their deployments due to the benefits discussed earlier. For Microsoft in particular, this was necessary since requirements for placing FPGAs in special HPC SKUs "constrained power to 35W, the physical size to roughly a half-height half-length PCIe expansion card (80mm x 140 mm), and tolerance to an inlet air temperature of 70∘C at 160 lfm airflow" [80]. Custom boards are not required, however: Nimbix and Tencent both use off-the-shelf.

Placement

Figure 3·3b shows that all these systems deploy FPGAs in a distributed fashion. This is because: i) FPGA resources are easier to orchestrate, ii) FPGAs can be offered as

bare-metal resources, which simplifies the tooling needed, and iii) FPGA failure affects only local resources, as opposed to potentially millions of nodes.

Network Connectivity - Use cases

Figure 3·3c shows two important trends. First, none of the systems uses a secondary network. This is likely because of: i) the cost and complexity of wiring a second network for potentially millions of nodes and additional networking hardware, ii) the potentially limited scalability if direct FPGA-FPGA connectivity is supported, and iii) high chip resource usage for building routers and securing the system. The second important trend is the relationship between network connectivity and use cases. Specifically, due to security and reliability constraints, systems that allow customers to offload their own applications do not support any direct network connectivity. Rather, this connectivity is only available if workloads are either internal, or if the offering is an application where only a limited set of APIs are exposed to the customer.

Intra-node Connectivity

Figure 3·3d shows four major trends. First, in all of the systems FPGAs communicate with the CPU over the PCIe slot. This emphasizes the role of the CPU as being the core computational resource, whereas the FPGA is a complexity offload engine managed by the CPU. Second, all FPGAs are connected to some form of off-chip storage, typically a DDR memory chip on the same board. Third, no production system currently offers instances with FPGA-GPU connectivity. To the best of our knowledge, none of the cloud providers has placed GPUs and FPGAs within the same node. In terms of FPGA-ASIC connectivity, only Microsoft supports this since the FPGA must transparently process packets for the traditional NIC. Fourth, a majority of systems support *local* FPGA clusters through PCIe switches or direct FPGA-FPGA interconnects. This allows for high speed connectivity among a small number FPGAs.

3.4 Research Architectures

We discuss systems that are presently in research and development and represent the most technologically plausible candidates for widespread future deployment.

3.4.1 Overview

One of the most commonly used research architectures is the cluster of *Back-End* tightly coupled FPGAs that deploys a secondary network using direct and programmable interconnects to connect FPGAs across nodes. Microsoft's Catapult V1 was a back-end system that connected multiple nodes in 6x8 tori [207]. It was demonstrated using Microsoft's Bing workloads; it is not clear whether is was ever part of a production cloud. Although this approach can substantially reduce FPGA-FPGA latency, it is difficult to scale beyond a single rack due to wiring requirements; in the general case it also requires each FPGA to implement a router to support the communication. Currently no such example can be found operating in the production cloud. Other research examples include the 2D torus of 64 FPGAs on Maxwell [67], Novo-G# with a 3D torus interconnect among 64 FPGAs [116, 226, 229], the Noctua system at the Paderborn Center for Parallel Computing [57, 196, 202] with point to point connections among its 16 FPGA nodes, and the Albireo nodes of the Cygnus supercomputer system at University of Tsukuba [154] with a 2D 8x8 torus.

Another research area proposed in [108, 157] involves *Channel-over-Ethernet* (CoE), which is a back-end, inter-FPGA Ethernet communication network using the OpenCL kernel programming. Communication is also possible via the host CPU with Infiniband as a primary network. The results demonstrate the feasibility of such a configuration as the system achieves a latency of 0.99µs for inter-FPGA communication via the secondary Ethernet switch as compared to 29.03µs via the host CPU. A drawback is that data are sent as packets so there is additional overhead, such as IP addresses

and flags, that reduce the effective data rate [263].

Other research architectures include systems that support a *Local Cluster*, but where the communication scaling via direct interconnects is limited to a single node. Examples include Novo-G (a former version of Novo-G#) that allowed eight FPGAs wired on the same node communicate directly [115]. Another example is the research systems currently deployed at the IBM SuperVessel Cloud [2] and the IBM Power8+CAPI cluster at the University of Texas, Austin [44] that use a *Shared Memory* cache coherency model.

A different approach is to directly connect FPGAs to the data center network as a standalone resource. Each FPGA can be accessed by a CPU or another FPGA resulting in good scalability. CloudFPGA at the IBM Zurich Research Lab has demonstrated that network-attached disaggregated FPGAs improve network latency and throughput over other configurations, e.g. SW-only, PCIe attached FPGAs, bare metal servers, and virtual machines [269, 270]. The authors have built data center rack scale prototype with 1024 FPGAs [6]. A drawback of such an architecture may be that FPGA-CPU communication is necessarily among separate nodes and has high latency. Another consideration is the increase in the number of TOR connections.

The Open Cloud FPGA Testbed (OCT) is another research system that connects off-the-shelf FPGA boards to the network and also to a host CPU via PCIe. The testbed provides flexibility for cloud researchers to experiment with bare metal nodes, FPGAs' programming, and with FPGAs connected directly to the network and to one another [135]. University of Toronto SAVI testbed connects FPGAs to the primary network [246]. The authors in [245, 247] have demonstrated that virtualising FPGA resources on the SAVI testbed enables multiple regions within an FPGA device to support different designs using APIs such as OpenStack. Enzian [1] at ETH Zurich employs an FPGA as a node connected to the network on one end and coherently

attached to a large server-class SoC on another node. Unlike Microsoft's *Bump-in-the-wire*, this system allows CPUs to either connect directly to the network or via the FPGA. Unlike other cache coherent systems, it allows the FPGA side of the cache coherency protocol to be extended and tailored [16].

a)

Network / Type of Board	Custom	Off-the-shelf
Primary	[E], [I]	[S], [O]
Secondary	[C1], [N#]	[G], [M], [Nc]
None		[Nr], [Nv], [P], [V]

b)

Placement	
Centralised	Distributed
	[C1], [E], [G], [I], [M], [Nc], [Nv], [N#], [O], [P], [Nr], [S], [V]

c)

Intra-node Connectivity				
FPGA	CPU	GPU	ASIC	Storage
[G], [Nc], [Nv], [N#], [M], [P]	[C1], [E], [G], [M], [Nc], [Nr], [Nv], [N#], [O], [P], [S], [V],	[G], [P]		[C1], [E], [G], [I], [M], [Nc], [Nr], [N#], [Nv], [O], [P], [S], [V]

Figure 3·4: Classification for selected research cloud FPGA architectures: Microsoft Catapult v1 [C1], Enzian [E], Cygnus[G], IBM cloudFPGA [I], Maxwell [M], Noctua [Nc], NARC [Nr] [89], Novo-G [Nv], Novo-G# [N#], Open Cloud Testbed [O], Power8+CAPI TACC [P], SAVI [S], IBM SuperVessel [V].

3.4.2 Architecture Trends

Figure 3·4 classifies the research cloud FPGA architectures using the taxonomy defined in Section 3.2.As we can see, with the exception of a few possibilities, research systems have explored different varieties of cloud architecture options. While production systems are bounded by several factors such as total-cost-of-ownership (TCO), power-usage-effectiveness (PUE), performance, resilience, modularity, scalability and

security; research systems tend to enjoy greater degrees of freedom.

Boards

Figure 3·4a shows that custom boards are preferred if the proposed systems are *Disaggregated*, network attached (e.g Enzian and IBM CloudFPGA). Also, for earlier *Back-end* systems like Catapult v1 and Novo-G# a customised board allowed the system to increase transceiver count. However, we can see that recent *Back-end* and *Local Cluster* systems mostly use off-the-shelf boards. Systems with no inter-node communication network almost always use off-the-shelf boards.

Placement

To the best of our knowledge, no research systems are deployed in a centralized manner (Figure 3·4b).

Network Connectivity

Figure 3·4a shows that research systems are distributed evenly across the different network connectivity options. We also note that newer systems almost always have network connectivity, either primary or secondary. This helps scale the application across multiple FPGAs and achieve lower latency.

Intra-node Connectivity

Figure 3·4c shows three major trends. First, none of the research architectures connects an ASIC with an FPGA on the same node. Second, all systems have some form of off-chip storage. Third, GPUs are being employed on the same node as FPGAs, especially for highly parallel, SIMD-like workloads and communicate over a PCIe switch [78].

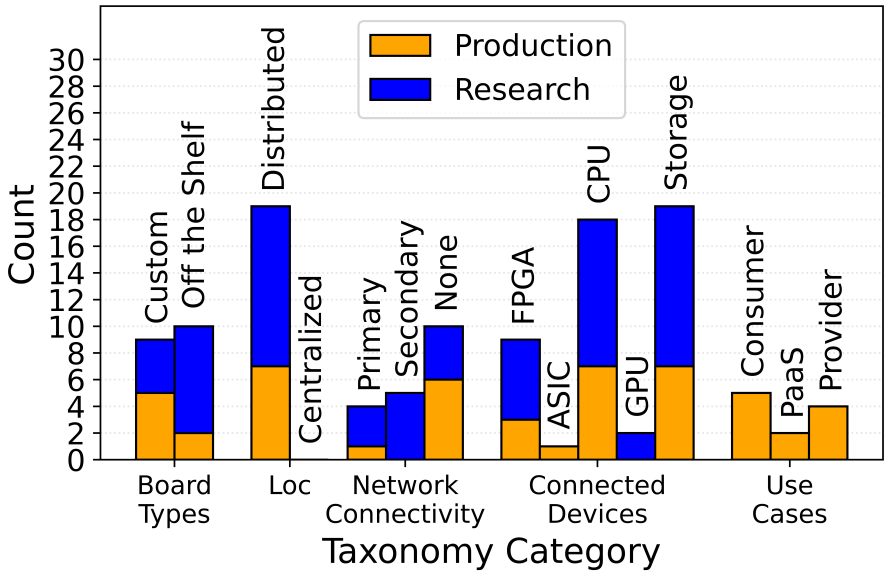

Figure 3·5: A Categorical Comparison between the Production and Research Systems analysed in this paper

3.5 Potential Future Innovation

We identify areas of potential novelty that can be derived by traversing the categories in the taxonomy, and by comparing different sub-categories with what is already present in Figures 3·3 and 3·4.

Type of Boards: Potential novelty here is with modular boards that lie at the intersection of custom and commodity. Similar to what is commonly done with microcontrollers, semi-custom boards can be built by buying and connecting together off-the-shelf modules for different FPGA chips, memory chips, and interfaces (QSFP+, PCIe etc). This would allow providers to tailor boards to their specific requirements, reduce the penalties of designing a custom board (development costs, upgrade costs, probability of failure, time to market), and easily replace specific modules as needed (due to hardware failure or for regular upgrades).

Placement of FPGAs: While FPGAs have been used in high end network switches [31, 43, 60], their role is typically limited to providing the performance and flexibility

needed to support changing protocols. However, there is currently no system that leverages TOR switches where FPGAs are responsible for implementing the entire switch hardware [32, 193].

Supporting such an architecture has a number of benefits. i) Customer offloads: Customers could use these TOR FPGAs to compute in the network e.g. for doing collective operations such MPI All-Reduce and Broadcast. ii) Provider offloads: Providers could leverage these FPGAs to implement services such as metering, accounting, analytics, and packet filtering. iii) Flexible networking: By combining FPGA based TORs with Bump-in-the-Wire FPGAs, a data-center-wide network could be created that does not rely on a standard protocol for communication. As a result, the communication latency could be reduced substantially. Alternatively, it may be possible to dynamically switch between different standard protocols based on the target workload.

Other research has demonstrated the potential benefits of deploying FPGAs as application-level accelerators within the switch [126, 127, 128, 129, 130, 131].

Network Connectivity: A potential novelty here would be to support both Primary and Secondary network connectivity, either within the same FPGA, or through multiple tightly coupled FPGAs within the same node. This would effectively combine key benefits of Microsoft's Catapult v1 and v2, i.e. having ultra low latency for rack scale communications through custom interconnects and still supporting data-center scale FPGA-FPGA connectivity.

Intra-node connectivity and Use cases: The connectivity between FPGAs and CPUs is typically done using the PCIe bus. This is because existing use cases define the role of the FPGA as an offload engine for the CPU. However, a potential novelty here is supporting sufficient low-level electrical coupling, such as the FPGA has read-modify-write access to the CPUs Baseboard Management Controller and

firmware. This would effectively turn the FPGA into a management and security controller for the CPU, and enable new *system administrator* use cases such as CPU firmware attestation. Some recent research using FPGAs to perform intranode functions, including managing distributed caches and orchestrating communication, has been shown to be beneficial in supporting communication [284, 285, 286] and performing certain AI tasks such as processing Graph Neural Networks and Recommender Systems [123, 124, 125, 295].

3.6 Conclusion

We present a survey of cloud FPGA architectures that explores the complex and non-trivial relationship between system requirements and deployment configurations and identifies areas of potential future innovation in this space. To help organize the survey, we use a taxonomy that abstracts away low-level implementation details while still highlighting advantages and limitations of a given architecture. Using this taxonomy, we classify both production and research systems; this in turn is used to demonstrate the major trends in cloud FPGA architecture. Finally, based on the findings of this survey, we identify several potential areas of innovation that are currently not explored in either production or research.

Chapter 4

AutoAnnotate: Reinforcement Learning based Code Annotation for High Level Synthesis

4.1 Introduction

A fundamental challenge in Electronic Design Automation (EDA) is the creation of code that is not only programmable with reasonable effort, but that is also performant. Basic to creating high performance FPGA applications is that they are usually programmed by developers experienced in that domain; moreover, programs for spatial accelerators such as FPGAs often do not follow the optimization principles of a traditional software design. High Level Synthesis (HLS) tools that enable transformation of High-Level Language (HLL) code into an FPGA specific design have the potential to offer a considerable advantage by enabling complex hardware designs using procedural languages. Among the vast number of academic and commercial products in this and related spaces are Electronic System Level (ESL) design tools [18, 142, 168, 184], runtime libraries [282], autotuners [255, 287], and other program development infrastructure [281].

It is often the case, however, that the HLS compiler is unable to output high quality results. Many studies have tackled this problem using pre-processing of the source-code, e.g., [102, 216, 304]. For example, FPGA vendors suggest source code reconstruction focusing on optimization for pipelined registers, predictions, and mem-

ory coalescing, among others. However, improving source code by using manual code rewriting also requires significant expertise in FPGA architecture, including programming for distributed memory resources, deep pipelines, and data-flow routing. Also, this approach is compiler agnostic; while this seems a positive, it turns out to be the opposite: we find that some practices that yield benefits with Intel HLS can result in the opposite with AMD Vitis.

An alternative, source code annotation with pragmas (or directives), aims to make apparent to the compiler certain opportunities in the code, such as potential for parallelism or reducing memory latency. Most existing HLS tools employ user directives to transform code. However, these require that a naive code be properly annotated with *specific combinations* of pragmas in order to improve performance [88]. There are several limitations: Which pragmas should be used to exploit inter- and intra-module parallelism and memory abstractions? And, Which combinations of these pragmas work well together and guarantee maximum performance? In particular, specifying HLS directives and pragmas presents several challenges:

1. Transforming source code requires not only knowledge of hardware micro-architecture, but also familiarity with the proper use of tool-specific optimization directives and pragmas. It also requires facility with the overall coding style, e.g., applying appropriate memory partitioning, which itself is dauntingly dependent on code complexity. In fact attempting to use pragmas may actually exacerbate the programmability problem as the designer now needs expertise with the HLS tool as well as with hardware design. Moreover, the effect of inserting a pragma depends not only on the HLS tool, but also its version and the target FPGA type.

2. Pragmas may need to be added at tool-dependent locations in the code. For example, the loop unrolling pragma for Vitis HLS must be inserted after the *for* loop; this is in contrast to other compilers such as Intel HLS, GCC, Clang, and OpenMP

where loop unrolling must be declared before the loop.

3. Even if the pragmas are inserted correctly, good performance is still not guaranteed. We have observed that incorrect (yet legal) insertion of certain pragmas can result in multiple factors worse performance. Infeasible configurations may increase latency costs, cause deadlock or result in runtime errors. For example, for an array used in a *for* loop: if the array partitioning factor is less than the loop unrolling factor, then the loop may not be unrolled successfully because the visits to the array are limited by the partitioning factor [18]. If the array partitioning factor is greater than the loop unrolling factor, more memory resources are consumed without increasing the parallelism. Compatible directives and factors are needed [241].

These challenges indicate that code rewriting with HLS directives is a hard problem and that there is a need for an intelligent automated toolflow to productively annotate code in a reasonable amount of time. While the objective is still to leverage parallelism and optimize resource usage, our approach is ***to extract maximum performance benefit without any additional programming effort, while using state-of-the-art tools to guarantee universal availability***.

This chapter presents AutoAnnotate, an automatic code annotation framework for HLS. AutoAnnotate reduces the need for developer expertise and effort by using Reinforcement Learning (RL), a ML approach that has proven valuable and superior to human experts in many fields [94, 132, 190, 223, 254]. We find that an RL agent can efficiently capture code characteristics and the HLS tool annotations that work well together, and thus to enable learning a model that predicts improved code insertions for a given application. AutoAnnotate adds value for both less and more experienced developers: the former in developing performant FPGA codes; the latter for generating design alternatives without having to go through hundreds of pages of reference manuals for each target tool. FPGA workloads are application-specific hence

we can train RL agent on the application itself. While code features are important for generating code specific annotations, the agent itself learns pragma combinations best suited for the given application. Moreover, these are dependent on the specific dataset on which they operate on, hence functional verification works.

This chapter makes the following contributions:

- An end-to-end RL-based source code annotation framework that takes C code as input and outputs a performant, pragma-injected C code. Automating insertion of compiler directives/pragmas improves not only usability and portability but also the quality of customized programs on FPGA platforms. If performance enhancement using HLS directives within a fixed number of iterations is not possible, it outputs the unoptimized code and suggests pre-processing using source code rewriting.

- An extensible method for design space identification involving source code profiling. This outputs code-specific pragmas and their insertion points. The RL agent then learns to annotate the C code with the minimum number of pragmas from the design space, minimizing the overall execution cycles within a set number of iterations. Given X as input C code for an FPGA application, construct a design space y= Z(k,X) where k signifies the possible pragma insertion points for X and y gives the list of possible pragmas for X that have been appropriately labelled with variable names and optimal factors. The RL agent then learns to annotate X with the minimum number of pragmas from the exploration space y with the aim to minimize the overall execution cycles within a set number of iterations.

- We demonstrate the framework on common FPGA workloads that have been analyzed in prior work [216] and on state-of-the-art HLS tools. Our results

show that using AutoAnnotate on these baseline codes gives an average of 42× performance improvement for AMD Vitis HLS and 3.42× for Intel HLS.

- We recognize that performance improvements over baseline HLL code can be difficult to interpret. We have therefore hand-optimized these codes using a procedure based on best practices [216] and again applied AutoAnnotate, this time still achieving a 32.3× performance improvement for AMD Vitis HLS and 3.1× for Intel HLS.

- We obtain the unexpected result that, of all of the combinations, the best performance was generally from running AutoAnnotate on the original baseline, rather than either code that has only been hand-optimized or hand-optimized code further optimized with AutoAnnotate.

The rest of this chapter is organized as follows. Section 4.2 discusses some of the closely related work. Section 4.3 gives the motivation and background for using RL to annotate codes for HLS. Section 4.4 presents the proposed framework, AutoAnnotate. Section 4.5 lists the evaluation methods. Section 4.6 evaluates the effectiveness of the approach using mainstream FPGA benchmarks. Section 4.7 gives the conclusion.

4.2 Related Work

Pragmas are inserted into source codes to give appropriate hints to the compiler and guide its process of performance optimization. Source code annotation for performance enhancement has been done several times; however, only a handful of publications have actually looked at the pragma insertion problem for HLS tools. *To the best of our knowledge, no one has yet used Reinforcement Learning to annotate generic C codes with pragmas for HLS workflows on any state-of-art HLS tool.* Our work is also the first in this category to verify that all transformed C codes indeed generate

functionally correct codes.

In other work, Amiri et al. [19] explored code annotations with regards to data-flow programming in streaming applications that have multiple kernels within a single code. Similarly, the source-to-source compiler, SpecHLS [120] focuses on HLS kernels that can benefit from speculative pipelining. Design Space Exploration for HLS [106, 218, 241, 279] focuses on heuristics-based approaches to sample the design space. Other work identifies regions of interest, or predicts optimal factors, but for limited directives such as loop unrolling or loop pipelining. Moreover, most of the work in this domain uses performance estimation instead of actual synthesis with an HLS tool. Since these quality-of-results (QoR) estimators are not the actual HLS tool, their performance values might differ significantly from the correct results. Also, they may not have proper verification of generated codes through regression testing. Some work that does invoke downstream HLS tools has run-times of days. And, to the best of our knowledge, few of these use state-of-art tools for performance estimation. [235] presents an HLS optimizer for design space exploration within Merlin compiler. Others, such as ScaleHLS [152], use their own QoR estimator to provide performance estimates.

4.3 Background

In this section we look at the programmability model for traditional HLS tools and the available "knobs" for users to pre-process their codes using directives/pragmas. We find that the complexity motivates the RL assisted framework. We then discuss an alternate pre-processing approach, namely, source code rewriting, and show that it can be combined with RL. Our overall framework is presented in Section 4.4.

Table 4.1: AMD Vitis HLS Pragmas Explored [18]

Type	**Attribute**	**Additional options**
Function Inlining	pragma HLS inline	off, on
Interface Synthesis	HLS interface	mode= ap_fifo ,s_axilite, m_axi; port; bundle; offset; depth
Pipeline	HLS Pipeline	rewind; enable_flush; II
Loop Unrolling	HLS unroll	factor
Array Optimization	HLS array_partition	variable; dim; factor type=cyclic,complete,block;

4.3.1 Programmability model for HLS tools

While HLS tools have raised the abstraction level for programming FPGAs, extracting performance from the input code is still a challenge. HLL codes are traditionally sequential, and typically written by developers with little knowledge of FPGAs' essential characteristics. Most HLS tools improve performance at their back-end by leveraging data-level parallelism using dependency analysis. However, much parallelism is still hard to predict. For this, tools such as AMD Vitis, Intel HLS, and Microchip's SmartHLS [18, 143, 184] allow users to insert annotations to highlight explicit parallelism, pipelining opportunities, or optimal memory interfacing.

The task of pragma insertion is notoriously demanding. Tables 4.1 and 4.2 list some available pragmas for HLS tools that have been explored in this work. Since applications investigated are single kernels, the scope of this study does not include task level pipelining and structure packing using pragmas such as pragma HLS data flow (along the lines of work presented in [19] for multiple kernels).

In HLS the constructs that have the highest impact on the final RTL micro-architecture are *functions*, *loops*, and *arrays*. Typically, an HLS tool first converts each function into a specific hardware component. For a given code, its ports and *functions* can be either be inlined or not. Inlining is especially useful for functions that are small and rarely called. These can be inlined into a larger caller function,

Table 4.2: Intel HLS Pragmas Explored [143]

Type	**Attribute**	**Additional options**
Pointer Interface	var_type*	restrict, volatile
Host Interfaces	ihc::mm_host< var >&var	Data Width, Alignment, Address Width, Burst Width, Latency Stable=1,0
Agent Memories	registers,memories	configurations= numbanks writeonly, singlepump, doublepump, bankwidth, volatile
Constant Interfaces	stable_argument	avalon_agent_ register_argument
Loop Unrolling	unroll	factor
Max Concurrency	max_concurrency	1,0
Max Interleaving	max_interleaving	1,0
Other Loop Optimizations	#pragma ivdep loop_coalesce, ii disable_loop_pipelining	factor
Global Optimizations	clang fp contract	fast,on,off
Other Global Optimizations	component_pipelining max_concurrency use_stall_enable_clusters	

allowing operations within the inlined function to be shared and optimized effectively. However, inlining can also worsen performance, specifically when the inlined function needs to be called multiple times within the parent function [18].

Next, *Loops* can be unrolled completely or only partially. Unrolling introduces hardware duplication. The amount of unrolling, however, is constrained by the memory bandwidth. *Loops* can also be pipelined with different initiation intervals. But an inter-iteration loop carried dependency will cause loop pipelining to fail.

Finally, *Arrays* can be mapped to registers or memories of different types. This is critical since interface contention—i.e., a RAM that allows fewer reads/writes than accesses in the same iteration—will cause a bottleneck. Deciding on the optimal number of concurrent ports for memory accesses is important so that if multiple arrays are feeding data to a compute task, these arrays can be mapped to different interface ports so as to access the data in parallel. Also, the data must be stored in different memory banks or the accesses will become serialized. Declaring appropriate

interface pragmas allows users to resolve such hardware contentions.

For designs with large numbers of loops, arrays, and functions, it is impossible to systematically explore all of the different design-space combinations. To come close to an optimal solution, an expert who can adapt rapidly is required. In this work, we have leveraged the particular capabilities of AI so that the expertise and adaptability come from reinforcement learning (RL).

4.3.2 Best Practices Optimizations of Baseline Codes

In prior work using source code transformation [102, 216, 304], researchers benchmarked common FPGA workflows by exploiting best practices of manual source code reconstruction. These best practices come mainly from (i) vendor-specific best practices HLS tool reference manuals, (ii) universally applied code optimization strategies such as use of loop unrolling and loop fusion; and (iii) FPGA-specific good practices, such as allowing concurrent memory accesses, using memory bandwidth effectively, and efficient resource binding. While applying best practices is compiler-agnostic, it does benefit from programmer understanding of typical HLS compiler behavior and of FPGA architecture.

An observation of note to this study is that, in some cases, *source code reconstruction offers benefits not possible with annotations alone*. This observation illustrates limitations of current HLS compilers: not all performance can be extracted using just compiler hints; in some cases programmer involvement is still required. Some examples include:

- reordering of loops for spatiality and for exploiting input caching;
- loop tiling to remove loop carried dependencies;
- rewriting code to remove overlapping array accesses;

- fusing multiple loops to achieve runtime reduction;
- extracting task level parallelism by implementing independent tasks, such as kernels, and connecting them using channels;
- restructuring code to remove conditional accesses and if-else conditions that can create bottlenecks for parallel execution; and
- since some HLS compilers cannot estimate latency if a while loop is present, converting them into for loops.

Accordingly, evaluations of HLS enhancements, such as those proposed in this study, should have multiple baselines: the original code, but also code to which best practices have been applied. The latter is essential since it represents versions of the code that are in states that are (apparently) not accessible to these HLS compilers. As a result our evaluations are with respect to both baseline code (version 1) and code that has undergone programmer-based reconstruction as in Table 4.3. As we see in Section 4.6, however, it turns out that AutoAnnotate on *unoptimized* code most often gives the best results.

Figure 4·1 shows code optimizations performed by AutoAnnotate on matrix multiply. When the baseline code is run through AutoAnnotate, it outputs annotated code that improves the latency by 7.3×. The baseline code is also hand optimized (as outlined in [216]); these results are summarized in Table 4.3. We find the version (#5) that gives the best performance; it is the same for both AMD HLS and Intel HLS. Results are given in Figures 4·3; to summarize, there is a latency improvement of 2.7× versus the baseline code. We then run the best hand optimized code through AutoAnnotate and the annotated code improves the hand optimized code by 1.98×. In this case, the combined benefit of hand-optimization plus AutoAnnotate is a speedup of 5.34× over the original baseline.

Table 4.3: Summary of code versions and optimizations applied

Version	**Optimization**
1	CPU-like single kernel C code
2	Implement task parallel computations in separate kernels connected using channels
	Apply optimizations such as loop unrolling, minimizing variable declaration using temps
	Using constants for problem sizes and data values to reduce off-chip memory access
	Coalesce memory operations
3	Implement all optimizations proposed for version 2 within a single kernel instead
4	Reduce array sizes to infer pipeline registers as registers, instead of BRAMs
5	Perform computations in detail,using temporary variables to store intermediate results
6	Use predication instead of conditional branch statements

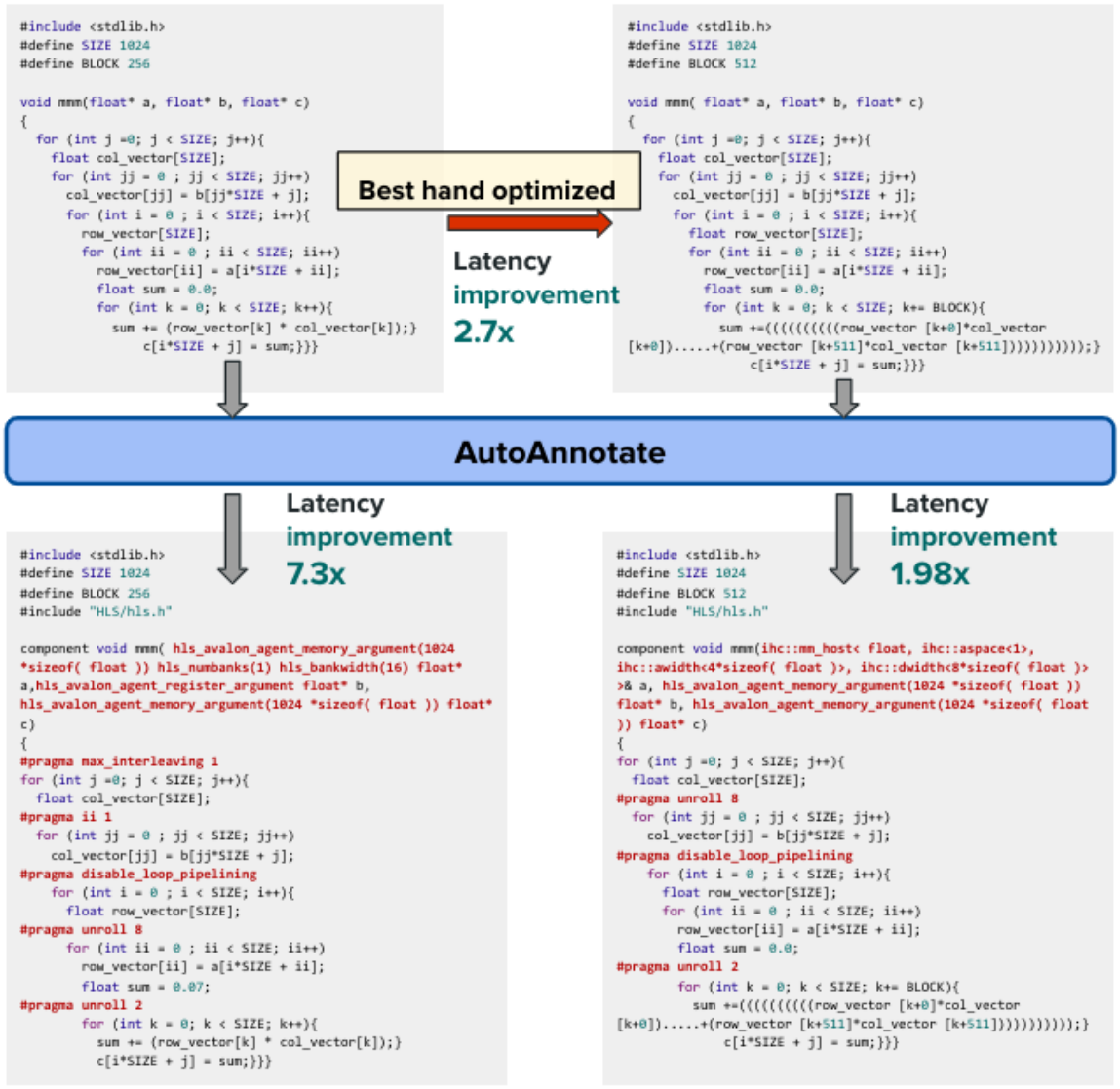

Figure 4·1: Example output from AutoAnnotate for Intel HLS: MatrixMultiply baseline code can be hand optimized using version 5 in Table 4.3 to increase performance by 2.7x. AutoAnnotate inserts pragmas into each of the baseline and hand optimized versions to increase performance by 7.3x and 5.34x with respect to the baseline code.

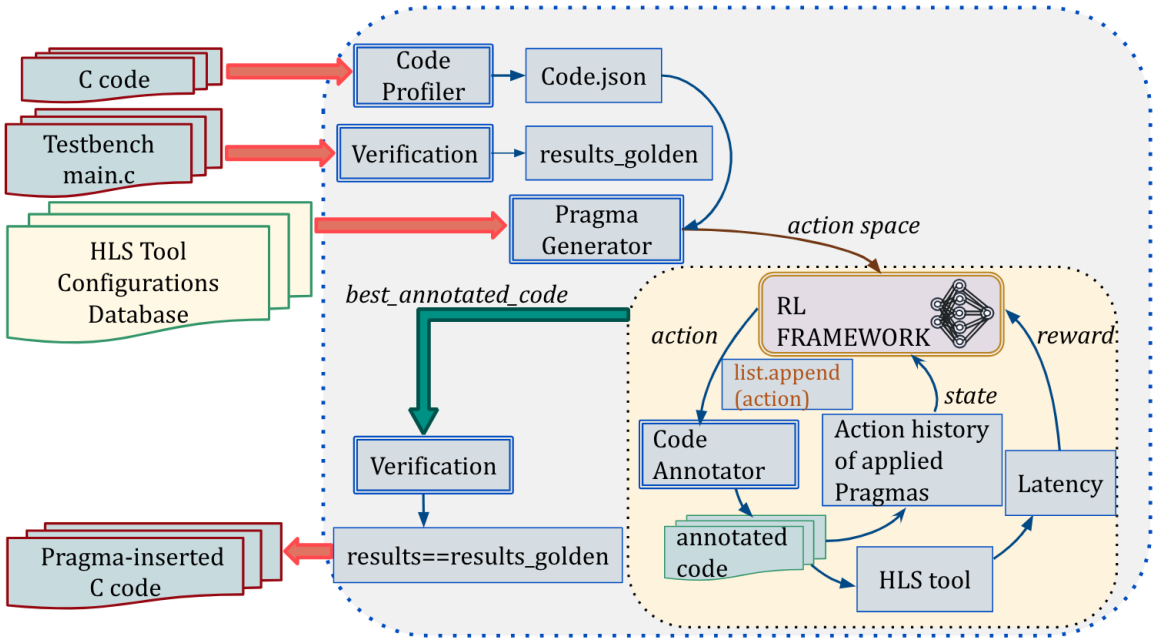

Figure 4·2: AutoAnnotate: Proposed framework for automatic HLS annotations using RL

4.4 The AutoAnnotate Framework

4.4.1 AutoAnnotate Design

The proposed framework for automated source code annotation for HLS is illustrated in Figure 4·2. The C code is fed into the framework together with HLS tool specific *pool* of pragmas. Tables 4.1 and 4.2 list the pragmas used in AutoAnnotate and their various configuration options. Most of these pragmas feature placeholders that must be correctly labeled with variable names, port designations, bundle names, and offsets. *Pragma-Generator* addresses this by accurately labeling placeholders based on variables extracted by the *Code-Profiler*. The *labeled-pool-of-pragmas* is passed as *action space* to the RL framework. The RL agent selects pragmas and the *Code-Annotator* places them at designated line numbers. The annotated code undergoes latency evaluation, enabling the RL agent to learn which pragmas yield the best rewards over time. Pragmas resulting in minimal latency after RL training are integrated back into the original C code to output the best-annotated version, subject to validation through co-simulation. The workflow of AutoAnnotate is detailed in Sections A,B,C and D below.

4.4.2 Code Profiler

Input code is first fed into a code profiler that outputs code characteristics that are required for HLS annotation. These include information about (i) various functions in the code and their declaration points; (ii) variables in each function and their names and attributes (e.g., pointer, read-only); and (iii) loops in each function, their declaration points and nesting levels.

4.4.3 Pragma Generator

The HLS tool-specific configurations are fed separately into the toolflow. Each configuration contains a directory of HLS tool-specific information such as (i) type/version of HLS tool, (ii) built-ins that are supported after function declaration, and (iii) pragma optimizing loops. Pragmas supported are given in Tables 4.1 and 4.2. Each of these pragmas is categorized into a pool of function and loop pragmas. Function pragmas are inserted either after function declaration (AMD HLS) or within the function arguments (Intel HLS). Similarly, loop pragmas are inserted either after for loop (AMD HLS) or before it (Intel HLS). This information, together with the code profiler output, is fed into a Pragma Generator.

The HLS Tool Configuration Database (see Figure 4·2) is the same regardless of the source code and depends on the HLS tool and its version. This is like a registry of possible pragmas supported by the tool. Most of these pragmas have placeholders that need to be annotated with appropriate variables from the function and other configuration values. The Pragma Generator labels the placeholders within each pragma with all possible variable names and configuration options and outputs a list of labeled pragmas and their insertion points for the specific source code. This list of (pragma, insertion point) is fed as an action space to the RL framework.

4.4.4 RL based Environment

Reinforcement learning (RL) is an ML approach where an agent learns by continually interacting with its environment. We propose that an RL agent can learn code characteristics and HLS tool annotations that work well together and so predict the best code insertions for a given application. In contrast to supervised ML, the RL agent can be tuned to optimize multiple objectives with large search spaces and does not need pre-labeled data for training. An RL framework has two major compo-

nents: i) the *environment*, or the problem to be solved, and ii) the *agent*, which is used to perturb the environment and learn based on feedback. The smallest unit of environment-agent interaction is typically a *step*. At each step, the agent predicts an *action* that the environment should take. After taking the action, the environment returns a *reward*, indicating the impact of the action, and an updated *state*, which represents the change in the environment resulting from taking the action. The next step then begins and a new action is predicted.

The above process continues until the environment indicates that a conclusion has been reached. This collection of steps, from the first predicted action to the action that causes the environment to reach a conclusion, is referred to as an *episode*. At the end of an episode, the environment is reset and a new episode starts. A collection of such episodes is referred to as an *iteration*. The agent is updated once per iteration. At the end of each iteration, the learning portion of the RL framework updates a *policy* (a deterministic or stochastic strategy) about which actions cause agents to maximize their long-term, cumulative rewards. RL assumes that the environment is Markovian, i.e., that the updated state depends only on the previous state and the action taken. It also assumes that the action taken is only dependent on the current state.

The RL framework for AutoAnnotate is set up as follows:

- Environment: The environment is composed of the HLS tool and the specific application. It outputs the number of clock cycles needed to execute the target application.

- Agent: Proximal Policy Optimization (PPO) is used as the reinforcement learning agent [219]. It is stable, easy-to-use, and gives good optimization decisions for source code annotations. Based on its good performance, it is also the default policy at OpenAI [194].

- Action: Each action represents a single annotation decision. This could be either one of the annotations generated by the pragma generator or a null action specifying that the agent do nothing. These decisions are appended to an ordered list of actions for the episode.

- Code Annotator: The list of actions suggested by the PPO agent is passed to a code annotator after every episode. This includes the pragma and its insertion point in the C code. The code annotator inserts all the actions recommended by the RL agent into appropriate locations in the source code and outputs an annotated code.

- State: In RL, a state represents the current environment that the agent is in. In this case, a memory-based state that stores information about previous actions taken within an episode is configured as the state. This is important since it allows the agent to plan its next actions based on the current situation and goals. This is referred to as an *action history* and is a list equal to the length of the action space (output by the pragma generator). Each element of the list is incremented and updated when a pragma corresponding to that list element is applied. PPO learns to map the distribution of applied pragmas to the next pragma that should be applied while maximizing the cumulative rewards across time-steps in an episode. Since the possible action can also be a null pragma, the agent is simultaneously forced to learn both the optimal combination and the optimal number of pragmas to insert.

- Reward: The reward is defined as the difference in latency between the unoptimized, unannotated code and the current step; lower latency thus results in a higher reward value. The reward is set to a default value of 0 for each step, except for the last step in the episode in which the actual latency is obtained

by running through the HLS tool. *Maximum Reward* is defined as the highest reward value obtained by any episode during training.

4.4.5 HLS Tool

In the current study we use both AMD Vitis HLS and Intel HLS. Moreover, adding support for any HLS compiler is straightforward. The annotated code after each episode is fed into the HLS tool and the post-synthesis reports generated are parsed by the toolflow to output the latency values. HLS tools typically try to generate scheduling algorithms based on the operating frequency. The frequency target and FPGA type can also be altered according to user preferences. While the goal here is reduction in latency, the system can also be set to optimize for area, or other measure of resource utilization, or for both latency and resource utilization. These are specified by reading the appropriate performance value from the tool and setting it as the reward function for the RL agent.

4.5 Evaluation Methods

4.5.1 Benchmarks

The seven benchmark codes in [216] are used for evaluation in this work: Needleman Wunsch (NW), Fast Fourier Transform (FFT), Range Limited Molecular Dynamics (RL), Particle Mesh Ewald (PME), Matrix Multiplication (MMM), Sparse Matrix Dense Vector Multiplication (SpMV), and Cyclic Redundancy Check (CRC). These were derived from mainstream FPGA benchmarks including Rhodinia [82] and OpenDwarfs [72, 161], including an optimization in [288]. In the present work, we have first validated their approach by porting their C code to Vitis HLS and Intel HLS. Here, and at all stages of the evaluation, we have verified correct program execution.

To obtain the baseline, the applications are all transformed into CPU-like single kernel C code, the standard starting best practice (Version 1 in Table 4.3). Because of the often orthogonal capabilities of programmer optimization versus code annotations (as described in Section 4.3.2) we also apply AutoAnnotate to codes that were transformed using the methods in Table 4.3.

4.5.2 Experimental Setup

The reinforcement learning framework is set up using Open AI Gym (0.21.0)'s environment interface, Ray (1.7.0) unified API, and Ray's RLlib library to provide the agent interface. Keras (2.6.0) is used to provide the neural net API for Python (3.9). Tensorflow (2.6.0) is used for machine learning. Each test is run using *a single worker* on a standard multi-core CPU, the same initial *seed* value for random number generation, and a total training time of *300 iterations*.

The HLS tools used are AMD Vitis 2021.1 and Intel HLS 23.3 [18, 143]. We set the target frequency to 300MHz. Artix-7 is used as the default FPGA for AMD Vitis HLS and Stratix 10 for Intel HLS. We believe that these methods for learning effective code annotations are general and applicable to different versions of these tools as well as to other HLS tools. Moreover, different FPGA types can also be chosen.

4.5.3 Applying Code Optimizations

Some details are as follows. First, Version 2 from Table 4.3 is omitted. It involves leveraging task-level parallelism using multiple kernels; the focus here is on single kernel optimization. Moreover, our premise is that the HLS compiler is still able to automatically infer task parallel pipelines [216]; also, having a single kernel offers several advantages such as reduced computation overhead and simple algorithmic flow [145].

Figure 4·3 show the results of individual optimizations proposed by [216] per-

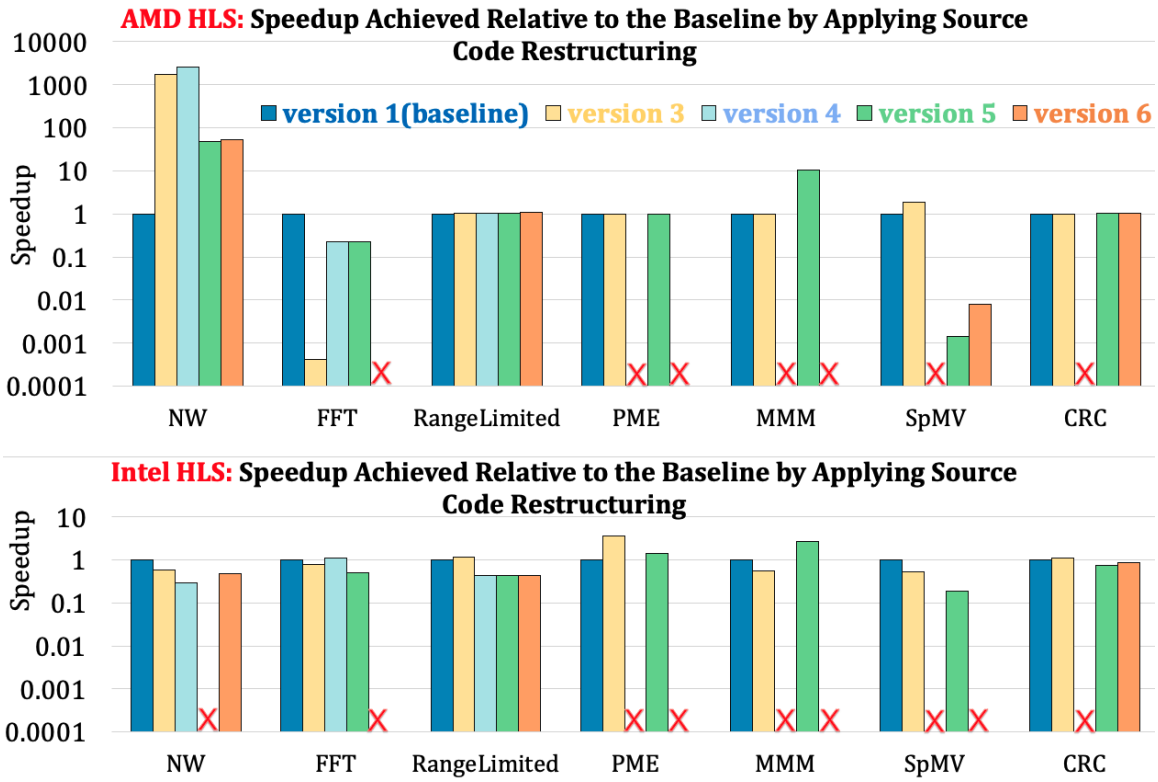

Figure 4·3: Systematic optimizations performed in [10] are applied for AMD Vitis HLS and Intel HLS. 'X' indicates that either the version was not created since the corresponding optimizations did not exist for the specific benchmark or the tool run into a deadlock/compilation error.

formed on top of the baseline code (Version 1) as evaluated using the AMD Vitis HLS and Intel HLS tools. In [216] the authors observed that each subsequent version of the source code restructuring typically increases performance. We find here that source code rewriting does not give much benefit at all for multiple benchmarks including FFT, RangeLimited, and CRC. For AMD HLS, the benefit is seen largely in the systolic-array-based Needleman Wunsch benchmark, where each version gives better performance than the baseline. Even here, however, Versions 4 and 5 unexpectedly give poor results when compared to the prior work. For Intel HLS, in the case of PME, Version 3 optimizations give best results. In the case of MMM, for both Intel and AMD, Version 5 gives best performance.

These findings highlights a crucial observation: *the benefits of programmer-applied optimizations, e.g., based on best practices, are highly compiler dependent.* For example, pipeline stages are inferred differently by both HLS tools. Performing computations in detail by storing intermediate results in pipeline registers has little to no impact for AMD HLS, but results in improvements in some cases for Intel HLS. This result is also heavily dependent on the application: dissolving branch statements into conditional assignments gives performance improvement for MMM; for other applications, however, there is little benefit.

4.6 Results

4.6.1 AMD HLS

Speedup achieved using AutoAnnotate

Figure 4·4 demonstrates the effectiveness of AutoAnnotate when compared to source code restructuring proposed in [216]. We postulate that modern HLS compilers can do a decent job at optimizing C codes without additional source code restructuring and using HLS tool specific annotations only. The results in Figure 4·4 support this

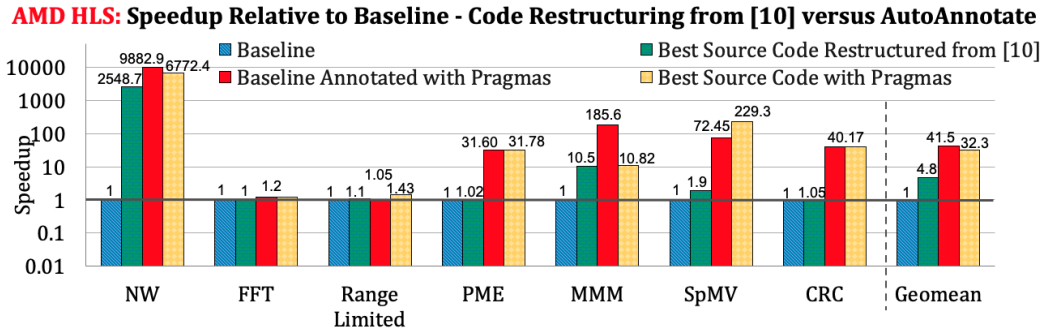

Figure 4·4: The performance of proposed annotation framework, AutoAnnotate on baseline source codes compared to source code restructuring proposed in [216]

claim for AMD Vitis HLS. Two versions are compared with the baseline. In the first, the baseline code for each of the seven FPGA benchmarks is optimized using source code rewriting; the code version that gives the maximum speedup is selected. In the second, AutoAnnotate applied to the baseline code. The results show that the RL agent does a good job at annotating code to enhance performance. On average, AutoAnnotate improves the baseline codes by 42× versus source code restructuring that improves it by 4.8×. At worst, AutoAnnotate gives performance comparable to source code rewriting. This also answers one of the big questions in HLS: Is it possible to perform better than source code restructuring using compiler annotations? The answer, as backed by the results is Yes. For some applications such as NW, PME, MMM, SpMV and CRC this is by a large margin. For the balance, FFT and Range Limited, it is still possible to give slightly better (by 20%) or, at least, comparable performance (within 5%).

Do random annotations also improve performance?

As discussed in Section 4.4 AutoAnnotate uses Proximal Policy Optimization (PPO) as the RL agent. PPO updates its existing policy at each *step* in order to minimize the cost function without deviating too much from the previous policy. Our goal here is to understand how much of the performance gain (in Figure 4·4) is because of RL

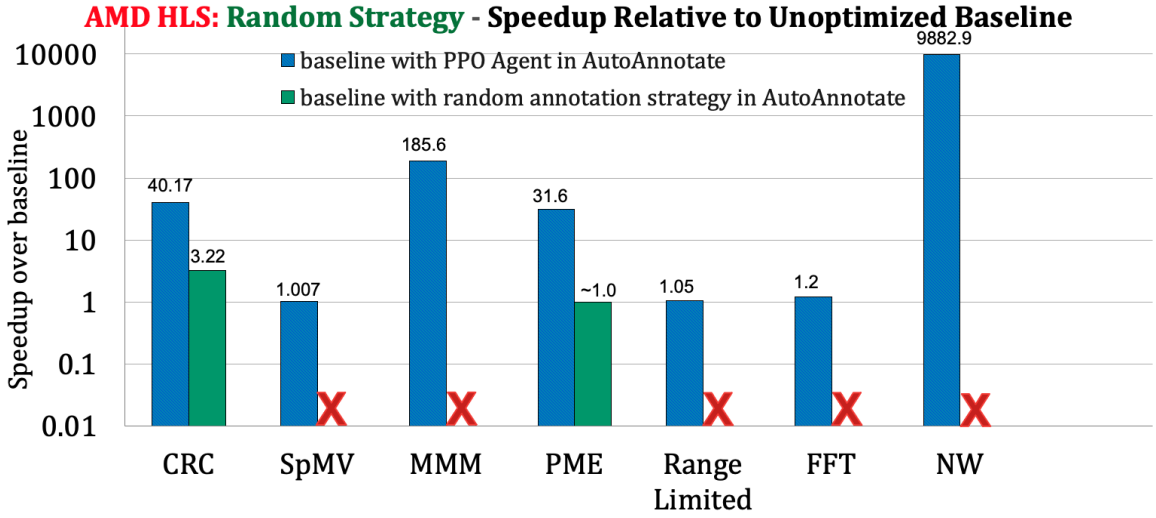

Figure 4·5: Relative Speedup achieved by using base configuration (with PPO agent) in AutoAnnotate versus using a random strategy to annotate code for equivalent number of iterations. X is used to denote the case when not even a single episode of random annotations output compilable code; the proposed annotations were outputting compilable errors on Vitis HLS

(PPO in this case); to do this we compare with a baseline *random* strategy. For the random strategy, we replaced the PPO agent with simple nested loops that ran for an equivalent number of steps, episodes, and iterations, and each time picked random actions of length equal to the list of pragmas (as set up for PPO-RL toolflow as well).

The results are given in Figure 4·5. For five out of seven benchmarks, the random strategy could not get even a single random occurrence of annotations without compilation errors. This is because for Vitis HLS, only a small subset of pragmas/annotations can work well together. And since the random strategy does not learn over time, for every iteration it ran into errors. For CRC and PME it was possible to get legal annotations even with random strategy, perhaps because the action space of effective pragmas is so small that it is possible to pick the acceptable ones just by chance. In both cases, however, the resulting performance was indistinguishable from the original baseline code.

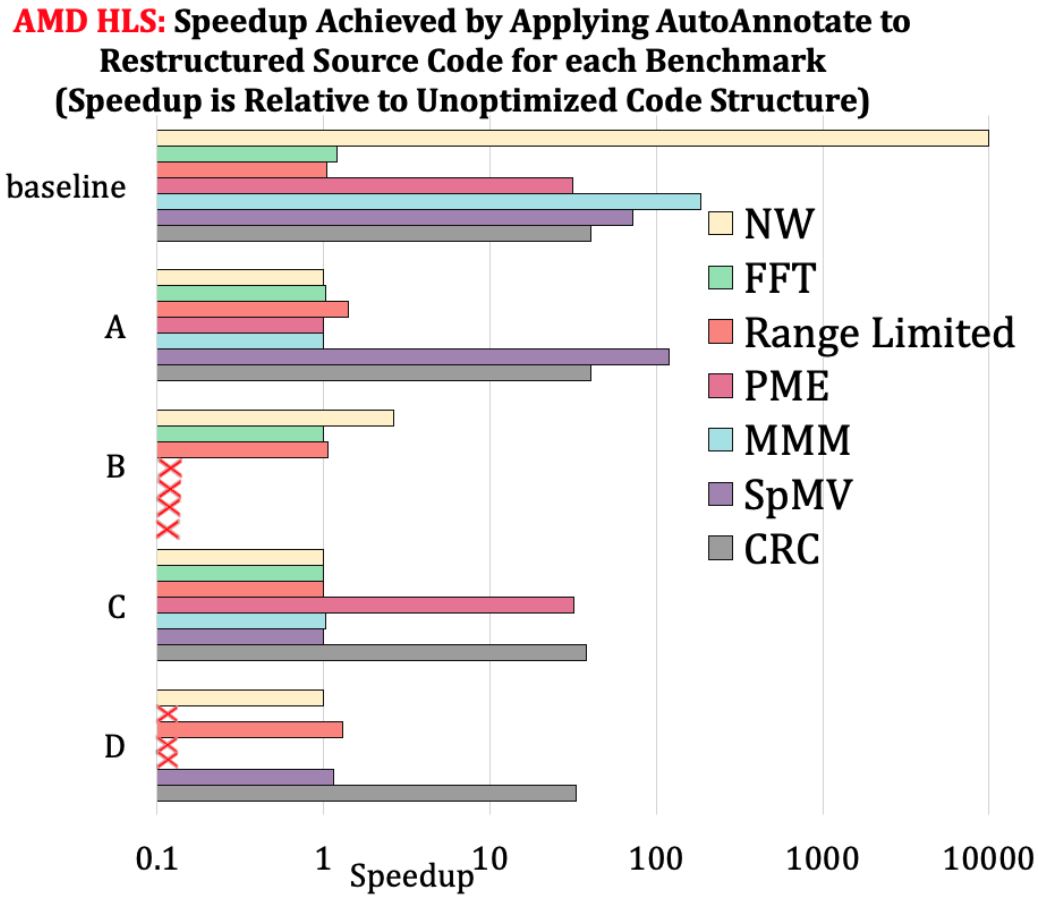

Figure 4·6: Impact of applying AutoAnnotate to each starting code version. 'X' indicates that code structure was not created since the corresponding optimizations did not exist for the specific benchmark.The speedup >1 shows performance was enhanced by applying strategy of AutoAnnotate. Even for the worst case, performance is equal to the unoptimized code. Baseline is the same as Version 1. Code structure A refers to version 3, code structure B refers to version 4, code structure C refers to version 5, code structure D refers to version 6 from Table 4.3.

Combining code restructuring with AutoAnnotate

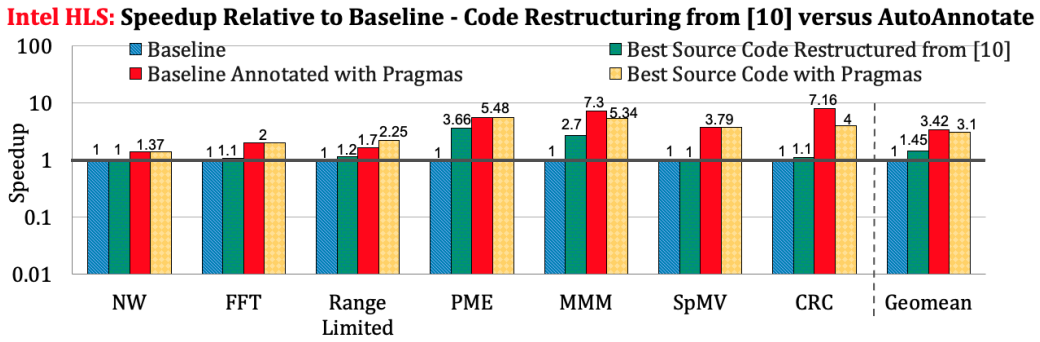

Figure 4·7: Speedup achieved by using AutoAnnotate to insert Intel HLS pragmas for both baseline and best source code restructured versions

Figure 4·6 shows the results of combining source code rewriting (as proposed in [216]) with AutoAnnotate. For the established FPGA benchmarks, we choose different code versions as proposed in Table 4.3 as our starting points. Code structures A-D refer to versions 3-6. Our goal is to understand just how much performance potential is achievable when AutoAnnotate is applied to each code structure. The results show:

- Figure 4·3 gives the best source code version for each application, e.g., for NW, it is version 4. Applying annotations on top of version 4 (code structure B) gives a speedup of 2.67× as can be seen in Figure 4·6. For SpMV, we can improve the best-case hand optimized code (code structure A-version 2 from fig 4·3) using AutoAnnotate by 120×. On average, AutoAnnotate improves the best hand optimized source codes by around 32.3× with respect to the unoptimized baseline.

- Results in Figure 4·6 show that pre-processing using AutoAnnotate gives the maximum performance benefit when applied to baseline code without any source code restructuring. Exceptions include Range Limited where AutoAnnotate

applied to code structures A and D gives slightly better performance than when it is applied to baseline codes; and SpMV where AutoAnnotate applied to code structure A gives 3.16× better performance than when it is applied to the baseline code. AutoAnnotate improves baseline codes by 42×, code structure A by 3.54×, code structure B by 1.42×, code structure C by 2.77× and code structure D by 2.65×.

- From the results we can also conclude that the HLS compiler in general, does well at inferring pipeline stages and leveraging parallelism; further benefit can be achieved by applying tool-specific annotations only (something that AutoAnnotate does) without requiring to manually rewrite the code structure. This is especially evident in the case of CRC. AutoAnnotate applied to every code structure results in nearly identical performance improvement. It is also evident in PME where AutoAnnotate applied to the baseline and to code structure C give the same performance improvement.

4.6.2 Intel HLS

Combining code restructuring with AutoAnnotate

For Intel HLS we combined strategies evaluated in Sections 4.6.1 and 4.6.1. Hand optimized versions as proposed in Table 4.3 were first run on the tool. The results are given in Figure 4·3. Out of these, the maximum performance improvement possible using hand optimizations (source code restructuring) for each workload is selected and its performance advantage is displayed as green bar in Figure 4·7. Next, the baseline codes for each workload are annotated using AutoAnnotate and the performance improvement is given by the red bars. Lastly, the best source code restructured codes are run through AutoAnnotate and the performance improvement possible is given by the yellow bars. We note that in most cases, annotating the baseline codes gives

performance improvement either better than, or at least comparable to annotating the best source code restructured codes. AutoAnnotate improves the performance of baseline codes by a geomean of 3.42× and best source code restructured by 3.1×.

4.7 Conclusion and Future Directions

We have developed an automatic source code annotation framework that replaces developer expertise and effort with Reinforcement Learning. We found that there is a substantial benefit to using AutoAnnotate, not only in automating the challenging task of pragma insertion, but also in obtaining performance that is generally better than that of applying programmer-directed best practices. In fact, this performance benefit (of AutoAnnotate on the baseline code) extended even to the application of AutoAnnotate to the hand optimized code. This last observation needs to be studied further. It could be that these hand optimizations have somehow constrained the degrees of freedom available to AutoAnnotate.

Going into this study we conjectured that a PPO agent could give good performance. We found (see Sections 4.5 and 4.6) that the agent does indeed learn, not only which annotations work well together, but also the minimum set of annotations to give that performance. In the future we will explore replacing it with an even "smarter" agent.

Also in this chapter we passed the history of applied annotations as a state for the agent. This works since FPGA workloads are generally application-specific. This is also important since only a handful of pragmas work well in combination with each other for each application. Hence the RL agent learns the minimum number of annotations that work well for that specific application. The information about code features is captured during code profiling in order to generate code specific pragmas for the application case. In the future we can incorporate the code profiling block

into the RL framework. We can encode the state of the RL framework so as to include code profiling information. This will drive the agent to learn not only the best annotations but also the best insertion points in the code.

Finally, in this work, we ran each training case for 300 iterations so as to limit the time in which we can output a pragma inserted C code. In the future, the agent could be trained to run over an optimal number of iterations until performant pragma inserted code has been achieved.

Chapter 5

AnnotationGym: A Generic Framework for Automatic Source Code Annotation

5.1 Introduction

Most popular compilers support annotations/pragmas as a way for programmers to convey knowledge that can help the compiler to better optimize code [18, 87, 143, 280]. This includes constraints or hints about compute and memory operations that cannot be easily expressed (if at all) due to the semantic and syntactic limitations of a high level programming language. However, inserting annotations correctly and effectively is challenging. Programmers need to know which annotations to use from a large pool, where to insert them, and how to correctly define them to ensure code correctness and performance. Moreover, this expertise may not port to every different compiler, or even to different versions of the same compiler.

In this chapter, we present AnnotationGym, a framework for automatic code annotation that addresses these challenges. The goal is to supply developers with an extensible tool that handles code annotations for a broad set of applications, compilers, hardware, programming languages, developer expertise, and target goals. Given a combination of target code and compiler, AnnotationGym leverages automated design space exploration techniques, including Reinforcement Learning (RL) and Bayesian Optimization (BO), to determine a performant set of annotations. RL has been used for various aspects of performance optimization such as robotics[13, 122], game-playing[73, 233] and compiler auto-tuning tasks[94, 132, 254, 264]. We ask: Can these

optimization algorithms also efficiently annotate source codes?

Specifically, this chapter makes the following contributions:

- An automated and portable approach to source code annotation that substantially reduces requirements for developer effort and expertise;
- An extensible framework with well-defined interfaces that achieves generality across compilers, benchmarks, computational hardware, and optimization algorithms;
- A validation methodology incorporated within the framework to verify the functional and semantic correctness of the generated annotation sets;
- Evaluations across multiple representative benchmarks, compilers, computational platforms, and optimization strategies that show the effectiveness of this approach. For the evaluated benchmarks we obtain, on average, a 57× performance improvement for Vitis HLS, 5.6× for Intel HLS, and 1.8× for GCC. This benefit persists when tested against otherwise optimized baselines.

The significance is as follows: (i) Makes a unique scientific contribution by being the first work to apply ML and learning based algorithms to automating the insertion of pragmas; (ii) Provides an end-to-end, effective tool for insertion of pragmas. Gets excellent results using standard evaluation methods, e.g., post Place-and-Route metrics of frequency and resource usage. Also, automates a standard code validation method. (iii) Provides benefits across the spectrum of hardware design experience: less experienced developers find potentially unfamiliar optimization strategies are automatically explored; experienced developers can port known hardware optimization strategies without going through hundreds of pages of documentation to determine compiler specific annotations.

The rest of this chapter is organized as follows. Section 5.2 discusses related work in automated code annotation. Section 5.3 presents the proposed framework. Section 5.4 describes the evaluation methods. Section 5.5 evaluates the approach. Finally, Section 5.6 gives a conclusion and future directions.

5.2 Related Work

Several studies have explored automated code annotation; they are advanced here though extensibility and generality, which also results in improved performance, including over previously optimized code. AnnotationGym addresses limitations of previous work both for CPU and FPGA code annotations.

5.2.1 CPU code annotation

Traditionally annotation libraries have been used to grant parallel semantics to code syntax that is originally written to run sequentially; examples include OpenMP, OpenACC, OpenHMPP, and OpenSs. A number of existing tools [21, 25, 99, 182, 209, 237] target automatically inserting OpenMP pragmas into source code. These tools have their own caveats: the output generated varies in format, i.e., source-to-source versus binary; pointer aliasing hinders parallelization; and additional checks are needed to ensure that dependencies are avoided. Loop parallelization strategies also differ: Cetus [99] deals with multi-core parallelization, while DawnCC [182] targets GPU threading. Modern compilers support many more annotations in addition to standard ones supported by these tools.

Other frameworks explore the space of loop transformations, e.g., in Clang by employing LLVM's polyhedral loop nest optimizer (Polly [159, 162, 280]), and optimizing loop vectorization parameters inside Clang's intrinsic pragmas [132]. AnnotationGym explores a broader set of annotations not tapped by existing work, e.g., by providing options to turn on certain compiler optimization flags [114] and by supporting

instruction set extensions [87].

5.2.2 FPGA code annotation

Much research in annotations for HLS has focused on design space exploration [106, 236, 241, 292, 297]. Such traditional heuristic based approaches such as simulated annealing and genetic search are not efficient in searching the design space for HLS codes. There is a need for highly efficient learner to analyze the search space and problem explored. In our work we use a data-driven approach utilizing a reinforcement learning agent. We strongly believe this is a promising approach for navigating future HLS landscape. IronMan [279] is one work in this space that utilizes RL. However they have explored use of RL to identify optimal resource allocations between DSP and LUT. We utilize the power of RL in not only identifying resource allocation, but also optimizing loop transformations, identifying appropriate interfaces specific to the HLS tool and optimizing the pipelines. In addition, IronMan uses RL learners like Actor-Critic and REINFORCE. We explore Proximal Policy Optimizer that is state-of-art and widely considered one of the best RL algorithms currently available [194]. A limitation of policy gradient methods, including one used by IronMan is that they require a substantial dataset. Our work utilizes PPO that is considerably sample efficient and is robust and stable, making it a good choice to learn across variety of HLS workloads. We also extend all prior works by also supporting automatic insertion of annotations into the source code, thereby creating an end-to-end flow. AnnotationGym leverages AI to automate the insertion of pragmas. It also demonstrates the applicability of reinforcement learning (RL): this is significant since RL reduces the search cost by using past data to drive optimization decisions [94].

Several studies identify regions of interest, or predict optimal factors for loop-based HLS directives, such as unrolling or pipelining. A major contribution of this work lies in creating a massive search space for multiple state-of-the-art HLS compilers and

deriving performance also by leveraging HLS directives for optimizing functions and arrays. For example, authors in [19] have explored code annotations for data-flow programming in streaming applications that have multiple kernels within a single code. The source-to-source compiler, SpecHLS [120] targets a subset of HLS kernels that can benefit from speculative pipelining strategies. [235] present an optimizer for design space exploration within the Merlin compiler. Others, such as AutoScaleDSE [152], use their own quality-of-results (QoR) estimator to provide performance estimates and optimize for loop tiling and array partitioning.

Other contributions of AnnotationGym are as follows. First, to the best of our knowledge, AnnotationGym is the first to explore HLS code annotation for Intel HLS. Second, to improve training accuracy, AnnotationGym evaluates all performance results using downstream HLS tools. Third, it also supports complete verification. The optimized codes are tested not only to ensure functional correctness, but also implemented on the target hardware to evaluate run time, resource usage, and frequency after place and route. And, finally, it demonstrates how annotated baseline codes compare with respect to their hand-optimized versions.

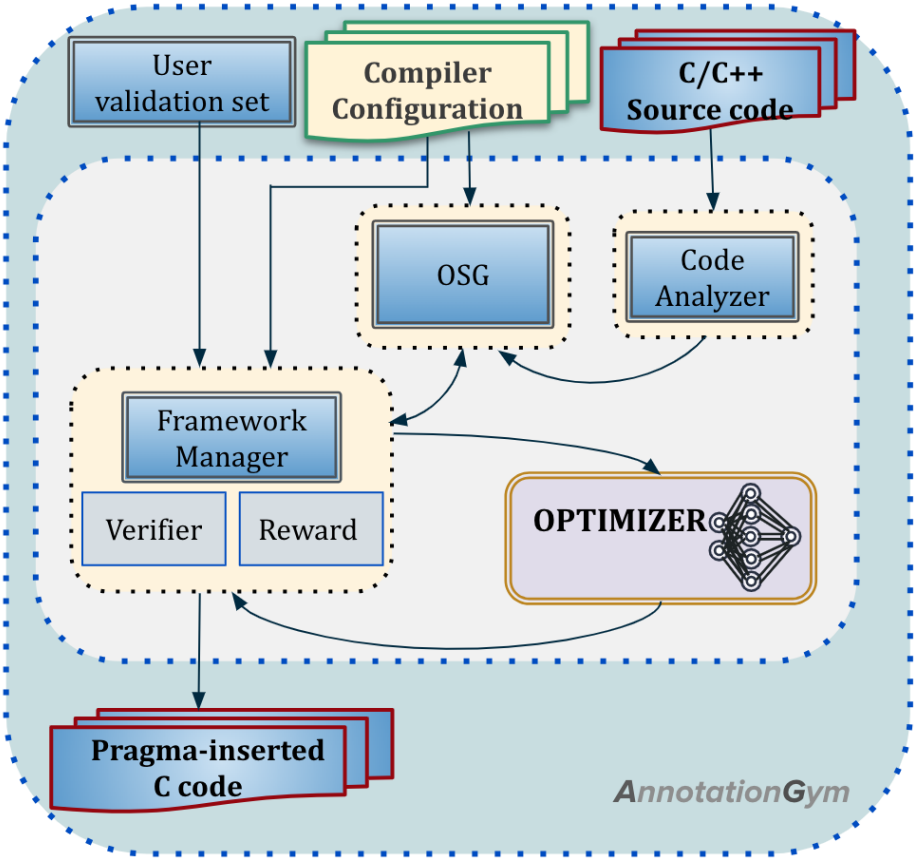

Figure 5·1: AnnotationGym: Framework Overview

5.3 Design of the AnnotationGym Framework

Among the design requirements for AnnotationGym are that it be extensible so that new features can be added to the work flow; modular, so that different blocks can be separately customized based on user requirements; and that it support validation and verification. This means that AnnotationGym can be used for virtually any combination of workload, compiler and design space exploration algorithm. The proposed framework is illustrated in Fig. 5·1 and detailed below.

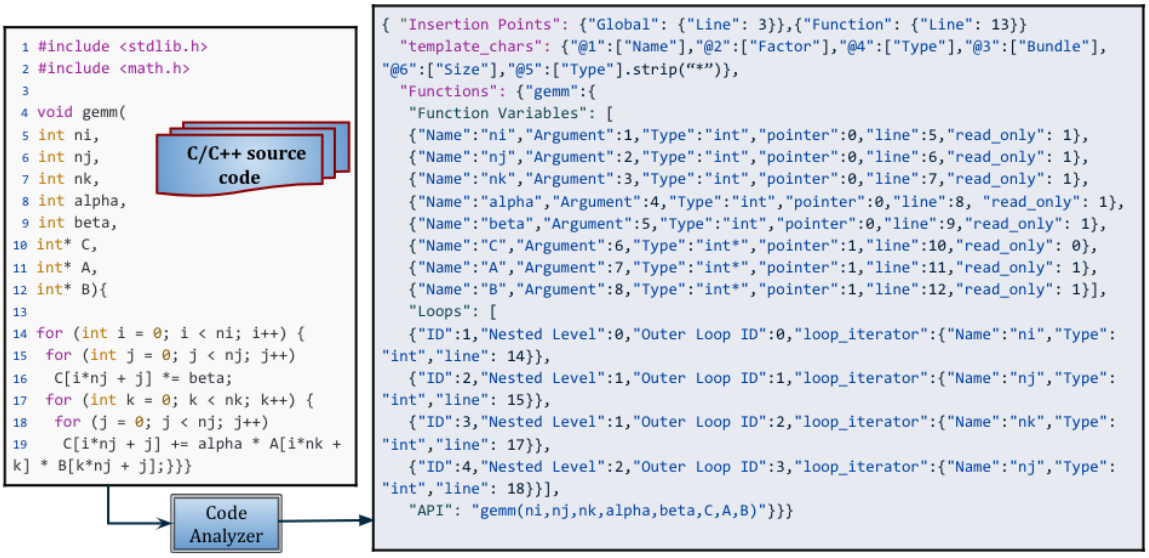

Figure 5·2: Code Analyzer: Source code is parsed and analyzed to create structured representation

5.3.1 Code Analyzer

The target source code (C/C++) is first parsed and analyzed to extract its annotation potential, i.e., the unique ways that the code can be annotated, and also details necessary for further processing. These include: (i) locations in the source code where annotations applicable to the entire code (global scope) can be inserted; (ii) functions in the code, including their declaration points, lists of function arguments and local variables, and lists of loops; (iii) arguments and variables within each function, including names and types (e.g. pointer and read-only); and (iv) loops within each function, including declaration points, unique ID, structural details (loop nesting levels), and loop iterator information (name and type). Once the required details have been extracted, a structured representation of the code is generated. Fig. 5·2 shows the code analyzer output for gemm (general matrix multiply).

The output from the Code Analyzer is same regardless of the compiler. This output is used by the Optimization Space Generator (OSG) to determine where annotations can be inserted, and provides context for annotations that target specific

code variables/arguments.

5.3.2 Compiler Configurations

Compiler configurations are specific to a compiler and provide the information needed by the framework for generating annotations, compiling code, performance measurement, and validating results. As an example, the compiler configuration script for Intel HLS is shown in Fig. 5·3. Annotations for loops, data interfaces, and functions as a whole (global) are registered separately. Similar scripts are also generated for other compilers such as Xilinx HLS and GCC. If a user wants to register an annotation/directive into the framework, they can simply append their information into the compiler configuration script for the specific compiler. The configuration file provides details such as:

Compiler specific commands for i) compiling and running, ii) profiling and evaluating with respect to the target metric, and iii) validating the annotated source code. As a default, the evaluation metric is configured as the latency in cycles. Users can also optimize for other metrics, such as area, or a combination thereof. These commands can be arbitrarily complex; and any additional files needed to support the execution of these commands can be linked during compile and/or run-time. Thus virtually any combination of tooling can be interfaced by the framework.

A list of annotations supported by the compiler is organized into three categories: Global, Function, and Loop. Global annotations include high level optimization options and target hardware specific options. Function annotations typically convey information about function variables, such as memory type, alignment, and memory arguments. These attributes can also be specific to the target hardware, such as array partitioning, type of memory protocol, interface bundles, latency, and, for HLS [18], alignment byte size. Loop annotations are applicable to individual loops, e.g. unroll, flatten, merge, and pipeline. For some annotations, context from the code

is needed to properly apply the annotations, e.g. variable names, types, and factor values.

5.3.3 Optimization Space Generator

The Optimization Space Generator (OSG) abstracts the code and compiler details from the rest of the framework, specifically, through the following.

(i) **Annotation Generation**: The OSG takes its input from the code analyzer and compiler configuration, and outputs a list of possible annotations and their corresponding insertion points for the specific source code. This is done by traversing the different IDs assigned to global, function, and loop-level annotations. It then shortlists the names and types of variables in each function and generates annotations with details in the appropriate namespaces. These details include variable names, types, and sizes; interface-specific details such as bundle sizes; and optimal factors for unrolling.

(ii) **Code Annotation**: The Framework Manager provides the OSG with a list of annotations to be applied. The OSG then parses the source code and inserts each annotation at the appropriate location in source code. Finally it outputs this annotated version of the source code.

5.3.4 Optimizer

AnnotationGym is not tied to any specific learning algorithm or optimization strategy. Instead it allows the user to choose the optimization technique best suited to the problem. The Optimizer has the following inputs:

(i) **Number of exploration rounds**: The number of source code annotations that will be tested by the Optimizer to reach a final best_case. Typically, the greater the number of trials, the more likely the Optimizer is to learn a best set of annotations. However, this also increases the training time and must itself be optimized according

to the application and the learning rate of the Optimizer algorithm.

(ii) **Optimization Space** is represented as a list of all annotations, as well as a range of values that can be selected for each annotation. In the simplest case, this is a 0 (apply) or 1 (skip). The exact range for each annotation and the significance for each value in the range are both handled outside the Optimizer. This allows greater control over the design space exploration (e.g., adding biases) without modifying or restricting the Optimizer itself. The Optimizer is also provided with a reference to the reward function, which takes as input a list of annotations and their corresponding values as determined by the algorithm.

5.3.5 Framework Manager

The Framework Manager coordinates the OSG, the Optimizer, and any calls to tools outside the framework (such as compilers and profilers). It begins by initializing the OSG object. The OSG object returns a list of all possible annotations. The Framework Manager uses this list, as well as any user- or compiler-specific constraints, to create an effective action space for the Optimizer. Next, this space is passed to the Optimizer along with the user-specified number of rounds of optimization exploration and a reference to the reward function. The Framework Manager receives the output of the Optimizer and the list of Optimizer-suggested annotations, together with their insertion points is passed to the OSG.

The OSG, in turn, annotates the code. Using information given by the compiler configuration, the Framework Manager can launch compilation, profiling, and validation runs for this annotated code. To support these operations, the Framework Manager contains two modules, which are described next.

Verifier

The Verifier has two primary responsibilities.

(i) **Pruning**: This step applies to CPU codes only. The list of annotations from the OSG can be pruned to remove illegal annotations and reduce the search space. Currently, pruning is done by applying each annotation individually to the source code and validating the resulting code using the compile command of the specific compiler. Thus pruning is an optional step that can improve results for a subset of compiler-code pairings. It is especially useful for code annotations within GCC and LLVM when alignment-specific information is provided by the *User Validation Set* (as seen in Fig. 5·1) and memory locations of specific arrays overlap. In this case applying the pragma __builtin_assume_aligned might produce wrong results and hence will be pruned.

(ii) **Validation**: To validate an annotated code, the framework evaluates correctness using a user-specified validation set. For HLS compilers, this is a typical testbench file used to ensure that the RTL generated is functionally equivalent to the source code. For CPU compilers, this step ensures that the output is same as for the unoptimized source code. To improve turnaround time, validation is not done at every exploration round, but only for annotated code whose performance exceeds the current best result. If validation succeeds, the best case result is updated. Otherwise, the set of annotations is discarded; this action is indicated to the Reward Function and the reward is updated accordingly. Fig. 5·4 shows an example of optimal annotated code output for three compilers.

Reward Function

The Reward Function is responsible for running the compilation and profiling steps on the OSG code output and determining the resulting reward values. It is also

responsible for shaping the reward if both the measurement decreases with increasing code performance and the validation fails. The reward function also keeps track of i) the best reward and its corresponding annotations and ii) the mapping of the Optimizer action space to the actual design space.

Table 5.1: GCC Pragmas Explored [113]

Builtin	**Additional Options**	**Additional Details**
`if (n==0) || n%x != 0) __builtin_unreachable`	• `x = 8, 15, 32` • `n = loop limit`	Tells the compiler that the loop limit size will always be nonzero or a multiple of `x`.
`__builtin_assume(block_size < n && n% block_size == 0)`	Used for nested loops	• `n = loop limit`
`var_x = (var_x_type*) __builtin_assume_aligned (var_x, n)`	• `n = 8, 32, 64` • `type = int/double/float`	Tells the compiler to assume certain alignments for pointers.
`#pragma GCC ivdep`	Compile with `-ftree-vectorize`	Tells the compiler to vectorize a loop even if it is not vectorized due to possible aliasing.
`#pragma omp simd`	Compile with `-fopenmp-simd`	OpenMP optimization supported outside of GCC annotations.
`#pragma GCC optimize` *x*	• `x = unroll-loops, Ofast, O3, fast-math, inline`	Global optimization options.
`#pragma GCC target` *x*	• `x = avx, avx2, fma, bmi, bmi2, lzcnt, popcnt, tune=native`	Global optimization options.

```
{"COMPILE_COMMAND":"i++ code.c -march=Stratix10 -o output --simulator none --fpga-only --clock
300MHz",
"VALIDATE_COMMAND":"i++ -march=Stratix10 -ghdl main.c -o main && ./main",
"REWARD_EVALUATE":{"LATENCY"},
"ANNOTATION TYPE":{
"builtins for pointer interfaces": [
{"ID" :"Pointer_Interface","Name":"@4 "},
{"ID" :"Pointer_Interface", "Name":"@4 __restrict "},
{"ID" :"Pointer_Interface","Name":"ihc::mm_host< @5, ihc::aspace<1>, ihc::awidth<4*sizeof( @5 )>,
ihc::dwidth<8*sizeof( @5 )>, ihc::align<8*sizeof( @5 )> >& "},
{"ID":"Pointer_Interface","Name":ihc::mm_host< @5, ihc::aspace<1>, ihc::awidth<4*sizeof( @5 )>,
ihc::dwidth<8*sizeof( @5 )> >& "},
{"ID":"Pointer_Interface","Name":"ihc::mm_host< @5, ihc::aspace<2>, ihc::dwidth<8*8*sizeof( @5 )>,
ihc::latency<0>, ihc::maxburst<8>, ihc::waitrequest<true> >& "},
{"ID":"Pointer_Interface","Name":"ihc::mm_host< @5, ihc::aspace<2>, ihc::dwidth<8*8*sizeof( @5 )>,
ihc::latency<0>, ihc::maxburst<8>, ihc::align<8*sizeof( @5 )>, ihc::waitrequest<true> >& "},
{"ID":"Pointer_Interface","Name":"hls_avalon_agent_memory_argument( @6 *sizeof( @5 )) @4 "},
{"ID":"Pointer_Interface","Name":"hls_avalon_agent_memory_argument( @6 *sizeof( @5 )) hls_numbanks(4)
hls_bankwidth(4) @4 "},
{"ID":"Pointer_Interface","Name":"hls_avalon_agent_memory_argument( @6 *sizeof( @5 )) hls_numbanks(1)
hls_bankwidth(16) @4 "},
{"ID":"Pointer_Interface","Name":"hls_doublepump hls_avalon_agent_memory_argument( @6 *sizeof( @5 ))
volatile @4 "},
{"ID":"Pointer_Interface","Name":"hls_numbanks(1) hls_singlepump hls_avalon_agent_memory_argument(@6
*sizeof( @5 )) hls_readwrite_mode(“writeonly”) volatile @4 "},
{"ID":"Pointer_Interface","Name":"hls_avalon_agent_register_argument @4 "}],
"builtins for constants": [{"ID":"Constant_Interface","Name":"@4 "
},{"ID":"Constant_Interface","Name":"hls_stable_argument @4 "
},{"ID":"Constant_Interface","Name":"hls_avalon_agent_register_argument @4 "}],
"pragmas before for loop": [{"ID":"Loops","Name":""},{"ID":"Loops","Name":"#pragma unroll
@2"},{"ID":"Loops", "Name" : "#pragma max_concurrency 1"},{"ID":"Loops", "Name":"#pragma
max_interleaving 1"},{"ID":"Loops", "Name" : "#pragma max_interleaving
0"},{"ID":"Loops","Name":"#pragma disable_loop_pipelining"}, {"ID":"Loops","Name":"#pragma ii
1"},{"ID":"Loops","Name":"#pragma ivdep"},{"ID":"Loops", "Name":"#pragma loop_coalesce"}],
"global": [{"ID":"Global","Name":"#pragma clang fp contract(fast)"}, {"ID":"Global", "Name":"#pragma
clang fp contract(off)"}, {"ID":"Global", "Name":"#pragma clang fp contract(on)"}, {"ID":"Global",
"Name":"#pragma clang fp reassociate(on)"}, {"ID":"Global", "Name":"#pragma clang fp
reassociate(off)"}, {"ID":"Global","Name":"hls_disable_component_pipelining"},
{"ID":"Global","Name":"hls_max_concurrency(0)"},
{"ID":"Global","Name":"hls_use_stall_enable_clusters"}]}}
```


Figure 5·3: Example: Compiler Configuration Script Intel HLS

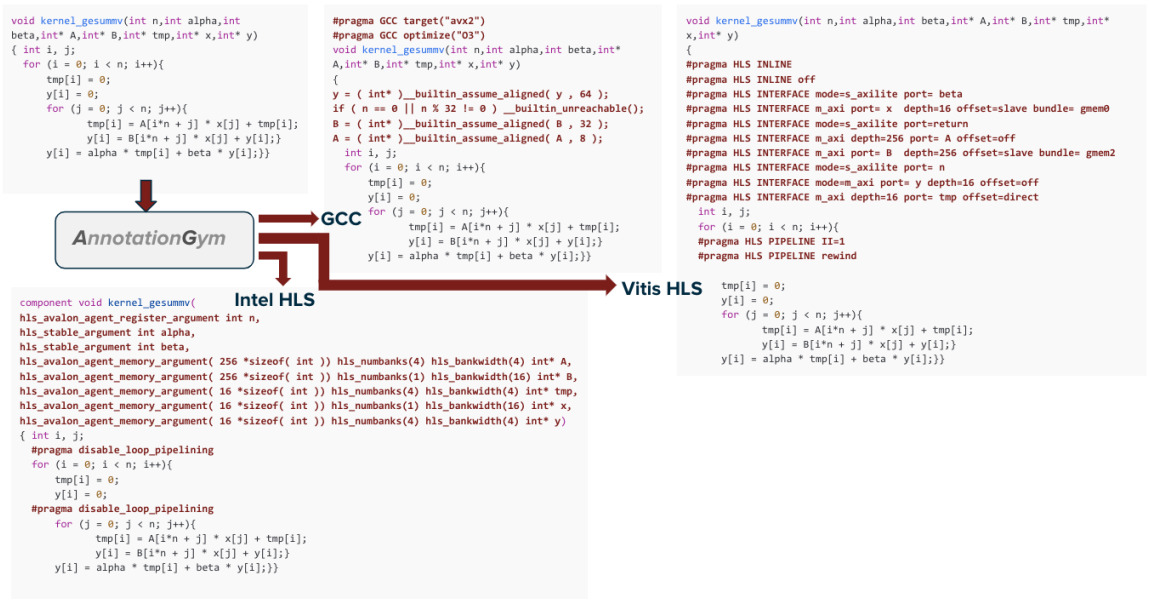

Figure 5·4: Example of automatically pragma-injected code output by AnnotationGym for GCC, Vitis HLS, and Intel HLS.

Table 5.2: Xilinx Vitis HLS Pragmas Explored [18]

Type	**Attribute**	**Additional options**
Function Inlining	pragma HLS inline	off, on
Interface Synthesis	pragma HLS interface	mode= ap_fifo ,s_axilite, m_axi; port; bundle; offset; depth
Pipeline	pragma HLS Pipeline	rewind; enable_flush; II
Loop Unrolling	pragma HLS unroll	factor
Array Optimization	pragma HLS array_partition	variable; dim; factor type=cyclic,complete,block;

5.4 Evaluation Methods

5.4.1 Benchmarks

The framework is evaluated using the Polybench/C benchmark [204] (version 4.2.1). This benchmark is chosen because: (i) the majority of the work in code optimization (CPU,FPGA) has been evaluated using this [11, 71, 132, 138, 151, 152, 235, 278, 279, 290, 298, 299] and (ii) it contains a wide range of practical applications and so is appropriate for general evaluation. For FPGA backends, we have also evaluated other popular FPGA workloads such as (1) Needleman Wunsch (NW) (16K×16K) (2) Particle Mesh Ewald (PME) (100,000 particles and $32 \times 32 \times 32$ grid, and (3) Sparse Linear Algebra (SpMV) (1K×1K matrix). These are derived from mainstream FPGA benchmarks including Rodinia [82] and OpenDwarfs [72, 161].

We have edited the benchmarks such that all arrays are referenced as pointers. This is done to bring the code into more common representation. Some languages (as opposed to libraries) do not have an array data type. Pointers allow for a more general representation of the data. However, languages sometimes do not convey much knowledge of the extent of the pointer. Annotations provide one way to guide compilers about overlapping memory and alignment of the pointers.

Table 5.3: Intel HLS Pragmas Explored [143]

Type	**Attribute**	**Additional options**
Pointer Interface	var_type*	restrict, volatile
Host Interfaces	ihc::mm_host< var >&var	configurations=Data Width, Stable=1,0, Alignment, Address Width, Burst Width, Latency
Agent Memories	agent registers, agent memories	configurations= hls_numbanks hls_writeonly, hls_singlepump, hls_doublepump, hls_bankwidth, volatile
Constant Interfaces	hls_stable_argument	hls_avalon_agent_register_argument
Loop Unrolling	#pragma unroll	factor
Max Concurrency	#pragma max_concurrency	1,0
Max Interleaving	#pragma max_interleaving	1,0
Other Loop Optimizations	#pragma ivdep #pragma loop_coalesce #pragma ii #pragma disable_loop_pipelining	 factor
Global Optimizations	#pragma clang fp contract	fast,on,off
Other Global Optimizations	hls_disable_component_pipelining hls_max_concurrency hls_use_stall_enable_clusters	 -

5.4.2 Target Hardware and Compilers

We evaluated the framework for both CPU and FPGA targets. For **CPU** optimization, we used GCC-11.3.1-3. The codes are compiled using the *-O2 -std=c99 -ftree-vectorize -mcpu=znver3* flags; the assembly codes are then piped directly into llvm-mca [175] for performance analysis. Additional compilation flags that are required by certain annotations are encoded by the OSG during the Annotation Generation phase. For example, for GCC, #pragma omp simd allows users to take advantage of OpenMP's thread parallelization, especially for loops that do not auto-vectorize when the *-O3* flag is applied. This pragma requires an additional flag -fopenmp-simd in order to work. The OSG ensures that the right flags are supplied together with the annotation.

The compiler configurations explored are given in Table 5.1. Alias analysis is an important part of an optimizing compiler since the C language, in general, does not prohibit overlapping memory regions. This information can be conveyed by inserting the pragma __builtin_assume_aligned at the appropriate location and labelled with appropriate pointer type and variable names. Similarly #pragma GCC ivdep can be used by the programmer to assert that there are no loop carried dependencies and,

hence, that the instructions can be executed in parallel. GCC allows some global annotations that can be inserted at the beginning of the code and applied to all the functions that follow. These include #pragma GCC optimize(*options*) and #pragma GCC target(*options*). Another, #pragma GCC optimize (Ofast), tells the compiler to assume associativity and hints about auto-vectorization. #pragma GCC target allows the compiler to make use of certain instruction set extensions. These can favor execution of other pragmas or generate SIMD instructions. However, while these annotations are used to increase application performance and exploit parallelization and vectorization, they can also cause a program to crash or make the code slower.

For **FPGA** optimizations, Xilinx Vitis HLS (version 2022.1) and Intel HLS (version 23.3) are used. The Xilinx target device is an xc7a100tfgg676-2L; the Intel target devices are Stratix 10 and Cyclone 10GX. The target frequency is 300 MHz for both tools. The pragmas explored for Xilinx HLS are given in Table 5.2; those for Intel HLS in Table 5.3.

We note that the scope of this study does not include task level pipelining and structure packing using pragmas such as pragma HLS dataflow. #pragma HLS inline allows the user to inline a function and its ports. However, inlining can also worsen performance, specifically when the inlined function needs to be called multiple times within the parent function [18]. Similarly, appropriate interfaces and their options must be defined for each function variable since interface contention will cause a bottleneck: e.g., for a RAM that allows fewer reads/writes in the same iteration. #pragma HLS unroll allows users to unroll loops. Unrolling introduces hardware duplication. However, the amount of unrolling is constrained by the memory bandwidth. Hence this can speed up the computation, but can also cause deadlock.

For Intel HLS, one way to mitigate poor performance from the Avalon Memory Mapped Host Interfaces is to configure its properties such as data width, max burst,

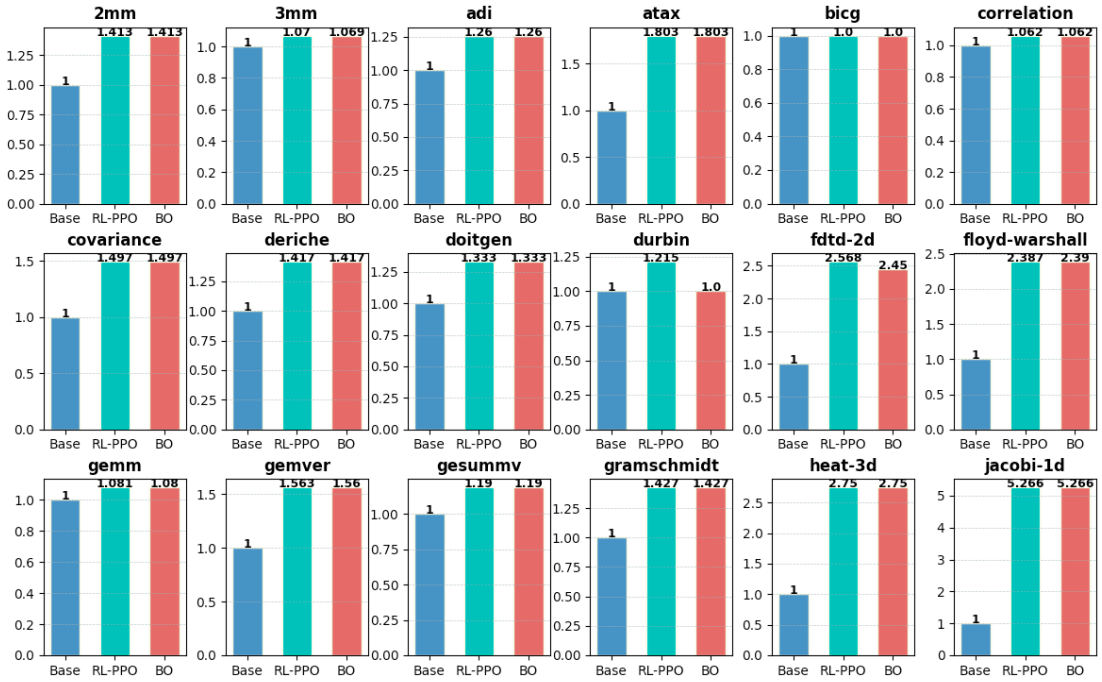

Figure 5·5: Relative Speedup achieved by using AnnotationGym for GCC with 2 different Optimizers: RL-PPO and BO

alignment, and read latency. If Avalon Memory Mapped Agent Memories are created for the specific variable, then their corresponding memory attributes—such as number of memory banks, bank width, readwrite_mode, and single versus double pump—can be optimized. Similarly, the HLS compiler allows the use of the *volatile* keyword to allow concurrent agent memory accesses and the *restrict* keyword to prevent memory dependencies. When possible, loops can be coalesced to reduce area and latency. The #pragma disable_loop_pipelining allows users to mitigate loop-carried dependencies by generating simple sequential datapaths.

To summarize: for both CPU and FPGA, an intelligent strategy is required to learn the list of annotations most suited to a code pattern. AnnotationGym provides both strategy and framework to annotate and validate the source code.

5.4.3 Evaluated Optimizers

AnnotationGym evaluates using two different optimizers. Reinforcement Learning and Bayesian.

Reinforcement Learning (RL)

RL is an ML method where an agent learns by interacting with its environment. RL assumes that the environment is Markovian, i.e., that the updated state depends on the previous state and the action taken. It also assumes that the action taken is only dependent on the current state. Proximal Policy Optimization (PPO) [219] is an on-policy, model-free RL technique with the goal of learning a policy, i.e., a state-to-action mapping that maximizes rewards accumulated over time. We propose that an RL agent can capture code characteristics and compiler annotations that work well together and so enable the learning of a model that predicts best code insertions for a given application.

RL is set up using Open AI Gym (0.21.0) to provide the unified environment interface, Ray (1.7.0) to provide the unified API for reinforcement learning, and its RLlib library to provide the agent interface. Tensorflow (2.6.0) is used for ML tasks, together with Keras (2.6.0) to provide the neural net API for Python (3.9). To ensure standardization, each test is run using *a single worker* on a standard multi-core CPU, a total training time of *300 iterations*, and the same initial *seed* value for random number generation. These details can be modified by the user. Other details for the RL Optimizer are as follows.

Action: Each action represents a single annotation decision. This could be either one of the annotations generated by the pragma generator or a null action specifying that the agent do nothing. These decisions are appended to an ordered list of actions for the episode. The list of actions suggested by the PPO agent is passed to the OSG after every episode. This inserts the actions into appropriate locations in the source

code and outputs an annotated code.

State: A histogram of applied annotations is used to represent the state; this is referred to as an *action histogram*. To implement this histogram, a list is initialized of length equal to that of the optimization space generated by the OSG; each element of the list is incremented and updated when a pragma corresponding to the pragma number is applied. The agent learns to map the distribution of the number of pragmas applied to the next pragma that should be applied. The learning can be seen from the overall increase in the average sum of rewards across episodes. Since a possible action can also be a null pragma, the agent is simultaneously forced to not only learn the optimal combination, but also an optimal number of pragmas to insert.

Reward is defined as the difference in latency between the unoptimized/unannotated code and the current step; lower latency thus results in a higher reward value. The reward is set to a default value of 0 for each step, except for the last step in the episode in which the actual latency is obtained from the HLS tool. *Maximum Reward* is defined as the highest reward obtained by any episode during training.

Episode size: The episode size is fixed to the length of possible pragmas for the code, i.e., the length of the action space. This also bounds the size of the action histogram.

Bayesian Optimization (BO)

BO is a framework for sample-efficient gradient-free optimization of functions. At a high level, the framework operates by alternating between using a surrogate model of the function to pick a new point to sample, and updating the surrogate model. The surrogate model is usually a Gaussian process model. The search space/domain consists of all possible combinations of pragmas with their associated factors and options inserted into the code. BO is setup using BoTorch 0.8.0 and ax-platform 0.2.10.

5.5 Results

5.5.1 CPU Optimizations

Table 5.4 shows speedups obtained by annotating benchmarks from Polybench/C using AnnotationGym. The results were evaluated by running 100 iterations of each kernel using the tool llvm-mca [175] and comparing the latency. The base codes are the unoptimized source code. The speedups show how much gain is possible using the different Optimizers within AnnotationGym. We observe a 1.8× average increase in performance when using simple annotations listed in Table 5.1.

For most benchmarks RL-PPO and BO give the same speedup. However, for fdtd-2d and durbin, RL-PPO outperforms BO. BO allows users to set optimizer-specific configurations such as *bias*. Similarly, RL-PPO allows users to configure hyperparameters. Tuning these Optimizer-specific configurations per benchmark can improve the learning rate.

AnnotationGym annotates the code with the discovered pragmas and ensures through robust validation that the pragmas preserve codes' semantics and function. Our results have shown, while an intelligible annotation can enhance performance by a geomean of 1.8×, a bad annotation set can degrade the performance by an average of 2.8× (up to 5.48×) across the evaluated benchmarks.

Fig. 5·6 shows the impact of applying a wrong set of annotations for the evaluated benchmarks. We note that up to 5.48× performance degradation can occur if a bad sequence of annotations is inserted into the source code. This further highlights the importance of AnnotationGym.

5.5.2 FPGA Optimizations

In the current study we have used both Xilinx Vitis HLS and Intel HLS. The pragmas included in the compiler configuration for each HLS compiler are given in Tables 5.2 and 5.3. After every episode the annotated code is fed into the HLS tool and the

Table 5.4: Speedups factors achieved for GCC for two optimizers

Application	RL-PPO	RL-BO
2mm	1.41	1.41
3mm	1.07	1.07
adi	1.26	1.26
atax	1.80	1.80
bicg	1.00	1.00
correlation	1.06	1.06
covariance	1.45	1.50
deriche	1.42	1.42
doitgen	1.33	1.33
durbin	1.21	1.00
fdtd-2d	2.57	2.45
floyd-warshall	2.39	2.39
gemm	1.08	1.08
gemver	1.56	1.56
gesummv	1.19	1.19
gramschmidt	1.43	1.43
heat-3d	2.75	2.75
jacobi-1d	5.27	5.27
jacobi-2d	2.28	2.28
syrk	1.49	1.49

Table 5.5: Comparison of sizes of annotation spaces explored by AnnotationGym with respect to prior work [297]

Application	AnnotationGym	Comba[297]	Difference
atax	7.21E+16	1.31E+08	5.50E+08
bicg	1.44E+17	5.76E+08	2.50E+08
gemm	5.63E+14	1.05E+10	5.36E+04
gesummv	2.25E+15	8.39E+08	2.68E+06
syr2k	5.63E+14	1.05E+10	5.36E+04
syrk	4.40E+12	1.64E+08	2.68E+04

post-synthesis reports generated are parsed by the toolflow to output the latency values.

Xilinx HLS

Table 5.5 compares the search space explored by AnnotationGym for Xilinx Vitis HLS with work proposed in [297] for the same benchmarks. AnnotationGym explores design spaces that are exponentially larger (on average 1.34E+8×) than in the previous work. This is because the pragmas considered include both typical pragmas, such as loop unroll and array partitioning, and also interface-related pragmas and their parameters (offset, bundle, port, depth). These parameters specify how RTL ports are

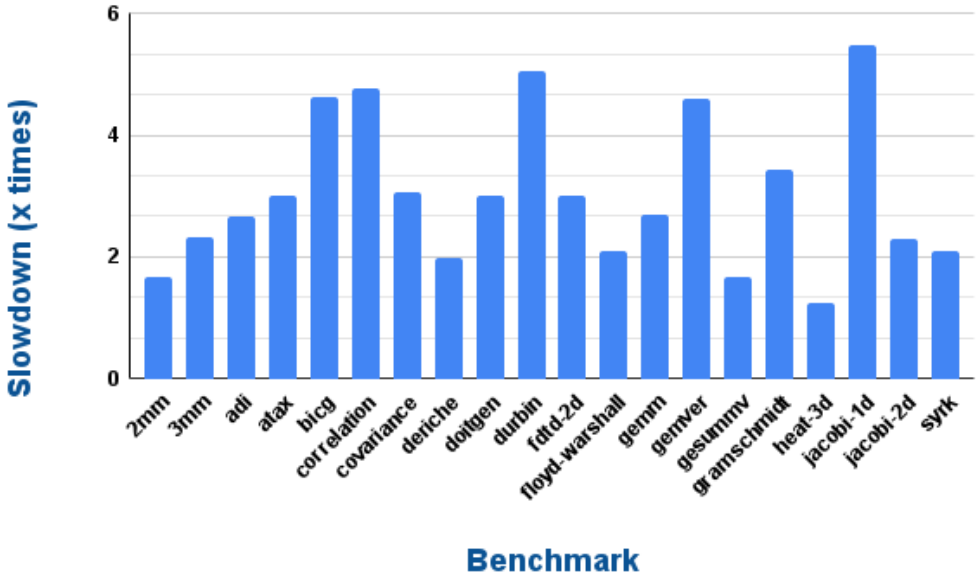

Figure 5·6: Legal annotations: Slowdown varies from 1.24× to 5.48×

Table 5.6: Importance of validation: Better speedups are possible, but some annotations are rejected by AnnotationGym since they fail validation

Application	Largest Failed Validation Speedup	Largest Successful Validation Speedup
atax	248	247
bicg	442	578
gemm	1867	1783
gesummv	1.5	1.5
syr2k	40.1	30.8
syrk	144	2.9

created from the function's arguments. Similarly, pipelining can also have multiple options. The pragmas are elaborated in Table 5.2. Fig. 5·7 shows the speedup of different Optimizers within AnnotationGym and a comparison to previous work [297] that also used the same benchmark[204]. *Base* is defined by the unoptimized benchmark. The speedup is measured with respect to the base. For some applications—such as bicg, gesummv, and syr2k—the Optimizer RL-PPO within AnnotationGym gives the best results. For others—atax, gemm and syrk—bayesian optimization (BO) gives the best results. AnnotationGym speeds up all applications by a geomean average of

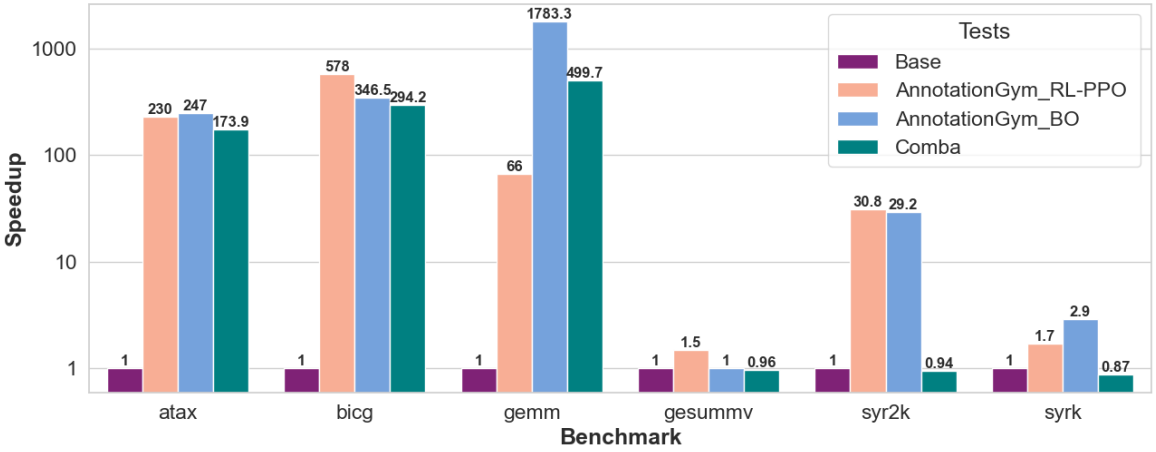

Figure 5·7: Speedup achieved by using AnnotationGym for Xilinx HLS with 2 different Optimizers, RL-PPO and BO, and comparison to previous work [297]

Table 5.7: Post Place and Route Results (xc7a100tfgg676-2L)

Application	**Runtime** (μs)	**Freq** (MHz)	**Latency** (cycles)	**LUTs**	**BRAMs**	**FF**	**DSPs**
atax_Unoptimized	251	173	43472	35660	31	81294	96
atax_Optimized	0.9	215	190	6805	47	15850	96
bicg_Unoptimized	-	-	163564	-	-	-	-
bicg_Optimized	2.3	174	397	13045	92	29194	192
gemm_Unoptimized	5223	130	677635	35846	31	82431	51
gemm_Optimized	208.6	214	44657	16037	31	44792	51
gesummv_Unoptimized	1.1	239	263	4884	31	11971	102
gesummv_Optimized	0.9	195	174	9898	77	21766	102
syr2k_Unoptimized	3374	213	720433	8878	31	24435	9
syr2k_Optimized	297	233	69296	5581	32	16554	9
syrk_Unoptimized	3567	202	720481	7989	31	21820	6
syrk_Optimized	2999	230	689696	6538	17	13598	24

a 56.9×. This is a factor of 2.72× performance improvement over previous work.

Table 5.6 shows the importance of AnnotationGym in ensuring all annotated codes are successfully co-simulated; i.e., that generated RTL designs perform as expected and give results functionally equivalent to C simulation. In particular, we find that for most benchmarks it is possible to get better speedup values (than reported here) by using annotations that are compiled correctly into the Vitis HLS flow, but that fail when co-simulated against the testbench.

Place and Route: Table 5.7 gives the results when each of the unoptimized and AnnotationGym-optimized-codes are implemented on the target device. We make

several observations. (i) The runtimes of the actual circuits improve on average by 64×, hence validating our performance goal of making the circuits run faster. (ii) Generally the resource utilization improves and frequency of the optimized circuits after place and route is better than the unoptimized circuits. (iii) For gesummv, the frequency of the optimized circuit decreases and resources such as number of LUTs, BRAMs, and FFs also increase. Yet the runtime is reduced by 1.24× and latency by 1.5×. This shows that the overall goal of achieving speedup has been achieved in the way circuits have been optimized. (iv) For bicg, the unoptimized code fails to generate hardware in the export-ip phase. This is in spite of it being able to pass co-simulation run in Xilinx HLS. This further highlights the importance of annotations for HLS: it is sometimes impossible to generate circuits without *legal* pragmas.

Intel HLS

Fig. 5·8 shows the speedups achieved by annotating the various codes for Intel HLS using AnnotationGym. Results are obtained for both Stratix 10 and Cyclone10GX. For the Stratix 10 performance is improved by a geomean of 3.21×; for the Cyclone 10GX by 5.6×. An important observation is that the annotations are different across applications. For example, for atax, optimal performance is achieved by declaring some pointer variables as host memory and others as agent memory. For bicg, performance is enhanced by leveraging the volatile keyword and memory coalescing on agent memory. For gemm and syrk, loop coalescing and instantiation of Avalon register arguments enhances performance. Since, to the best of our knowledge, this is the only work exploring code optimization using pragmas for Intel HLS no comparison is possible.

Place and Route All optimized results were verified by generating bitstreams and running the design on the actual hardware (for Cyclone 10 GX). Results are given in Table 5.8. We observe the following. (i) In all cases, AnnotationGym achieves its

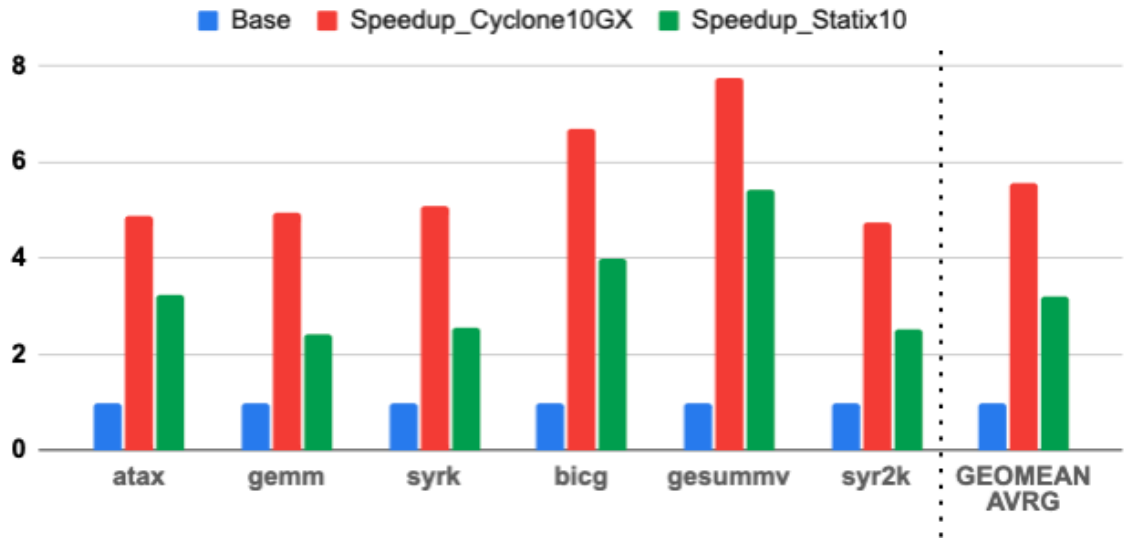

Figure 5·8: Speedups using AnnotationGym for Intel HLS

performance target of a decrease in latency (in cycles) as shown in Fig. 5·8. (ii) The overall runtime (in μs) decreases for all cases. Frequency, however, does decrease in some cases due to longer critical paths. However, since the reduction in latency is significantly larger than the decrease in frequency, the overall runtime decreases significantly. (iii) Optimization drastically improves resource utilization. As a result, we can target larger problem sizes than unoptimized codes and fit more logic into the same device.

Table 5.8: Place and Route Results (Cyclone 10 GX)

Application	**Runtime** (μs)	**Freq** (MHz)	**LUTs**	**Dedicated Registers**	**Block Mem Bits**	**DSP Blocks**
atax_Unoptimized	0.9	266	5755	10765	34624	5
atax_Optimized	0.1	388	1423	3257	8192	4
bicg_Unoptimized	0.8	273	5407	9779	45248	5
bicg_Optimized	0.08	414	1812	3467	33792	4
gemm_Unoptimized	0.7	351	4502	8865	34176	9
gemm_Optimized	0.2	234	1162	2505	24576	6
gesummv_Unoptimized	0.95	293	6563	11998	70272	8
gesummv_Optimized	0.09	397	1789	3363	16384	7
syr2k_Unoptimized	0.7	323	6321	12150	36864	11
syr2k_Optimized	0.2	231	1942	4908	24576	8
syrk_Unoptimized	0.7	333	5182	9858	32768	8
syrk_Optimized	0.2	235	1255	2861	16384	6

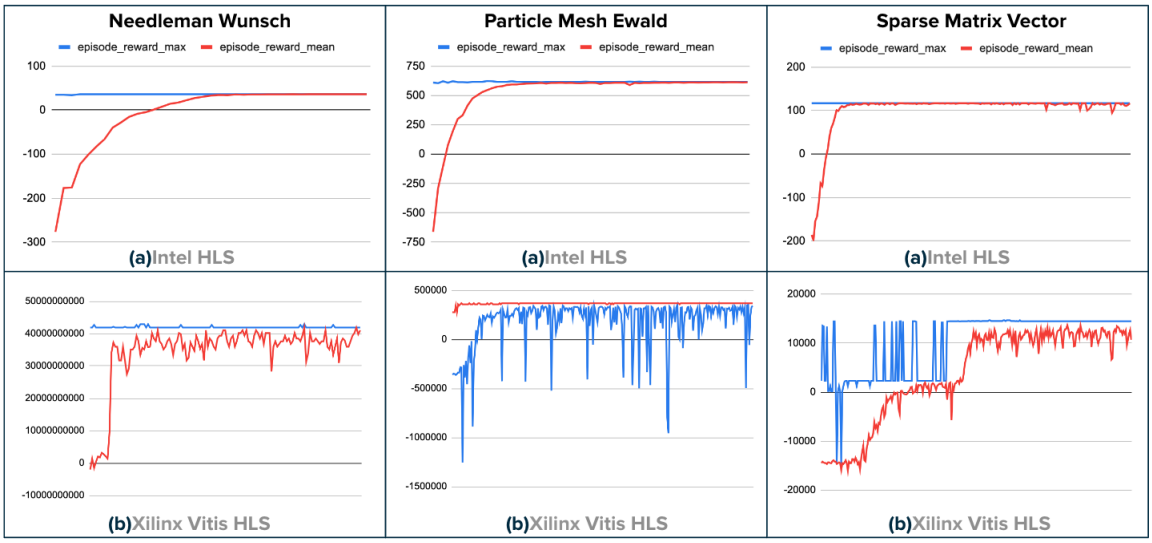

Figure 5·9: Reinforcement Learning Episode Reward Max and Episode Reward Mean. The rising mean shows how the RL agent learns to annotate code that increase its overall reward (decrease in latency).

Table 5.9: Speedup achieved on baseline codes using AnnotationGym to insert HLS compiler specific pragmas

Application	Intel HLS Speedup	Xilinx Vitis HLS Speedup
Needleman Wunsch(NW)	1.37	9882.9
Particle Mesh Ewald(PME)	4.63	31.6
Sparse Metric Vector(SpMV)	3.88	72.6

Large and Complicated Algorithms

We have also tested annotating more complicated FPGA workloads such as NW, PME, and SpMV and present them here in a separate subsection for both Intel and Xilinx HLS. Results are given in Table 5.9 for the target hardware in Section 5.4.2. Fig. 5·9 shows how the RL optimizer learns over time to annotate code. Note that its episode reward mean converges to the maximum episode reward that is possible for the application. These results strongly suggest how RL and BO can not only solve the problem of automating developer best practices, but could also find non-obvious optimizations.

Place and Route We verify all results output by AnnotationGym by running

Table 5.10: Place and Route Results (Cyclone 10 GX)

Application	Runtime (μs)	Freq (MHz)	LUTs	Dedicated Logic Registers	Block Memory Bits	DSPs
NW_Unoptimized	0.6	283.3	3239	5653	543296	0
NW_Optimized	0.3	301.1	2410	4901	529408	0
PME_Unoptimized	1.2	257.9	31188	71702	192832	9
PME_Optimized	0.8	199.6	46592	97526	1057600	90
SpMV_Unoptimized	0.6	299.5	2569	5196	6592	1
SpMV_Optimized	0.08	367.7	1563	3618	131072	1

Table 5.11: Place and Route Results (xc7a100tfgg676-2L)

Application	Runtime (μs)	Freq (MHz)	Latency (cycles)	LUTs	BRAM	FFs	DSPs
NW_Unoptimized	194819791	220	42949689428	6708	71	17078	0
NW_Optimized	14480	300	4345854	2442	58	1742	0
PME_Unoptimized	1706	216	369138	17029	31	31903	164
PME_Optimized	45	260	11681	5503	4	11778	102
SpMV_Unoptimized	64	227	14635	6944	31	16878	3
SpMV_Optimized	0.47	322	151	801	0	1139	3

the designs on the actual hardware. Tables 5.10 and 5.11 give the results. Note that these tables illustrate how, for the same target hardware (FPGA) and application, AnnotationGym allows the user to compare optimization strategies preferred by different compilers while optimizing for the same goal. For example, if the developer goal is latency reduction, the results after place and route show that HLS compilers can achieve this either (i) by creating efficient pipelines thereby increasing frequency and decreasing resource usage; (ii) by decreasing both the frequency and resource usage yet extracting latency advantage (reduced cycles) through runtime reductions; (iii) or by decreasing both runtime and latency(cycles) by increasing computational hardware and hence the resource usage.

5.5.3 Discussion

Overall, the results are promising: benefits are found for unoptimized codes; these persist for hand-optimized best codes. As expected, annotations guide various HLS compiler decisions and add substantially to the code optimization process. More specifically, we see that the performance benefit comes in large part from many sources: (i) Loop Unrolling and Pipelining; (ii) Memory Management (i.e. defin-

ing data locality, memory access patterns, data reuse); (iii) Specifying Interfaces to increase throughput (i.e. communication width, communication protocols); (iv) Resource Binding (specifying FPGA memory such as DSP, BRAM, and External memory to be used for specific parts of the code); (v) Parallelism (parallel regions within the code) and dataflow optimization. But the overall benefit is apparently derived from the creation of a very large search space and its effective traversal.

5.6 Conclusion

Perhaps the most important broader contribution of AnnotationGym is its generality and extensibility, allowing continued research across compilers. In particular, it provides a platform to test, train, and evaluate different compilers and target devices under one framework. AnnotationGym will be open-source and publicly available and we will work with alpha/beta users on the integration process. In future work, we are planning to integrate other compilers, including Clang, OpenMP, Merlin, Legup; and other hardware such as GPUs. It is also possible to integrate other domain specific programming infrastructures such as [165] and [289].

Chapter 6

Reinforcement Learning Strategies for Compiler Optimization in High level Synthesis

6.1 Motivation

High Level Synthesis (HLS) is a critical part of the Field Programmable Gate Array (FPGA) tool chain since it can substantially reduce the complexity and turnaround time for building custom hardware. This is especially crucial as FPGAs become critical components in the data center and HPC, e.g., in SmartNICs and disaggregated clusters, and run a variety of complex applications that must be coded by programmers without expertise in traditional logic design. Unlike traditional Hardware Description Languages (HDLs), HLS can generate hardware directly from CPU codes by automatically translating sequential functional descriptions into spatial circuits. This overlap between CPU and FPGA means new compilers need not be written from scratch. Rather, existing backend compilers such as LLVM can be modified to target FPGAs.

Such a modification typically involves: i) changing the optimization strategy for the Intermediate Representation (IR) code to better map the CPU-like sequential to FPGA-like spatial programming model, and ii) adding support for hardware generation through mapping of IR code fragments to hardware blocks and the interconnects between them. The former is especially important here since a different back end

means simply reusing CPU optimization strategies delivers far lower hardware quality than circuits written using Hardware Description Languages (HDLs) [216, 288, 304].

Since there are dozens of possible optimizations passes, the number of possible sequences of passes undergoes combinatorial explosion and discovering the appropriate optimization strategy for FPGAs is not practical. Moreover, any discovered optimization strategy has limited reuse given the diversity of FPGA workloads. Automating this discovery process is thus essential. One potential approach is to use supervised machine learning algorithms to model the relationship between input IR codes and effective optimization strategies. However, this approach is not practical due to the complexity of building a required labeled training data set - we hit the manual discovery roadblock again here. Another, more promising approach, is to use Reinforcement Learning (RL). Unlike manual or supervised approaches, RL can traverse the optimization space for input codes and, based on system state evolution and reward feedback, learn the LLVM optimization pass orderings that give good hardware quality.

While existing efforts in RL based HLS tuning have demonstrated improvements over -O3, they have been limited in scope. Specifically, these efforts have: i) only explored a small number of learning strategies, and ii) evaluated the impact of these strategies using a single metric for learning quality i.e. speedup over *-O3*. This is a significant drawback since learning goals can vary substantially across FPGA workloads and developer requirements. For example, developers can prioritize lower turnaround times over largest speed up values. In such cases, a different learning strategy would be needed which is able to trade off achieved speed up for a faster learning rate. Thus, similar to a uniform optimization strategy, a generic learning strategy is also inefficient.

In this work, we address the above limitation by expanding both the number of

valid learning strategies for HLS compiler tuning and the metrics used to evaluate their impact. To achieve this, we start by implementing an existing effort as the baseline strategy - this strategy trains a reinforcement learning model to learn the number of times each optimization pass should be applied. Next, we identify and implement a number of additional strategies that govern the agent-environment interaction and can potentially impact learning - we use these strategies to vary *what* is being learnt and not just *how* it is learnt. Next, we identify speed, fluctuation band and performance potential as metrics for evaluating learning quality, in addition to the existing metric of speedup over *-O3*. Finally, we evaluate the effectiveness of our approach by training the RL model using the Proximal Policy Optimization (PPO) method on all 9 benchmarks evaluated in former work and shortlisted from CHStone suite and Legup examples.

The specific contributions of this chapter are:

- Improving RL for HLS by enabling greater flexibility for developers in making trade offs based on their target learning goals.
- Identifying four novel learning strategies for HLS compiler tuning.
- Identifying and utilizing three additional novel metrics for learning quality.
- Demonstrating the effectiveness of our approach by comparing against the state-of-the-art using the CHStone benchmark suite.

The rest of this chapter is organized as follows. Section 6.2 discusses some of the prior work in this area. Section 6.3 discusses the RL framework for HLS compiler tuning used in existing efforts. Section 6.4 presents our proposed learning strategies and learning quality metrics. Section 6.5 evaluates the effectiveness of our strategies using the 9 benchmarks suite. Finally, Section 6.6 gives the conclusion.

6.2 Related Work

Compilers execute optimization passes to transform programs into efficient forms by leveraging the hardware design patterns. Optimization can serve several goals: reduce area/resource utilization, reduce latency, increase parallelism, reduce power/energy etc. Modern compilers offer various compilation options to the user [173]. Compilers provide options to select optimization levels for different goals (size, speed, debuggability) and within these goals sometimes allow to adjust for the time allotted to the optimization passes to control turn-around times.

However, the work performed for each of the optimization levels is defined statically and on a rather ad-hoc basis which doesn't take the problem nor really the variety of the target hardware into account. Even if the passes type, number and order applied to different applications in the standard -Ox level vary, the strategy is primarily fixed and insufficient [140]. Moreover the strategy is designed primarily for CPU workloads and hence not optimal for other specialised back ends like FPGAs. For simple applications like matrix multiply, we have seen that with appropriate number of passes and their ordering, more than 60% improvement in performance over the standard -O3 is possible.

Compiler optimization problem space for HLS is vast. Hardware quality is affected not only by the high level program but also by several other factors such as pass ordering, the number of passes and their parameter values. Hand tuned heuristic-based methods that been formerly proposed for compilers [12] no longer suffice. Compiler optimization has evolved over the years, from iterative compilation exploring the enumerations of the observation space one by one [84, 140] to machine learning based modeling in which models are trained to make relevant predictions [167, 266]. Recent years have seen a surge in compiler optimization work employing reinforcement learning (RL), a subset of machine learning in which the agent learns continually by trial

and error using its interactions with the environment. Prior research work employing RL for compiler optimization typically address the following categories:

1) Those that have looked at phase ordering to tackle the compiler optimization problem such as [10, 24, 101, 141, 176, 277]. Amongst these, [141] is the only state-of-art framework, to the best of our knowledge, that addresses the problem particularly using High Level Synthesis for specialised backends such as FPGAs. Our works builds upon it to improve the existing approach and explore further possibilities.

2) Others that analyze specific parameters with respect to compiler passes for example inferring appropriate vectorization and interleaving factors for the loops in the code [132], Improving the register allocation problem in compilers such as in [155]. Polyhedral models that optimize loops in the program and try to find an optimal schedule for the instructions in the defined polyhedral [7, 68, 159]. MLGO [254] where machine learning is applied with respect to the inlining pass.

3)Proposed frameworks and platforms such as Supersonic [265] to choose and tune a RL architecture for code optimization tasks, CompilerGym [94] that consolidates some compiler optimization frameworks into a platform to motivate research under the hood and generic tuning frameworks such as OpenTuner [20].

6.3 Background

6.3.1 Reinforcement Learning in High Level Synthesis

In this section we delve deeper into reinforcement learning and high-level synthesis and how the two can be combined for compiler tuning. Figure 6·1 part (i) shows the generic RL flow,6·1 part (ii) shows the generic high level synthesis flow and 6·1 part (iii) shows the combined RL based HLS.

High Level Synthesis (HLS) allows developers to implement workloads using High Level Language (HLL) codes, such as C, which can be compiled into circuits. Typical

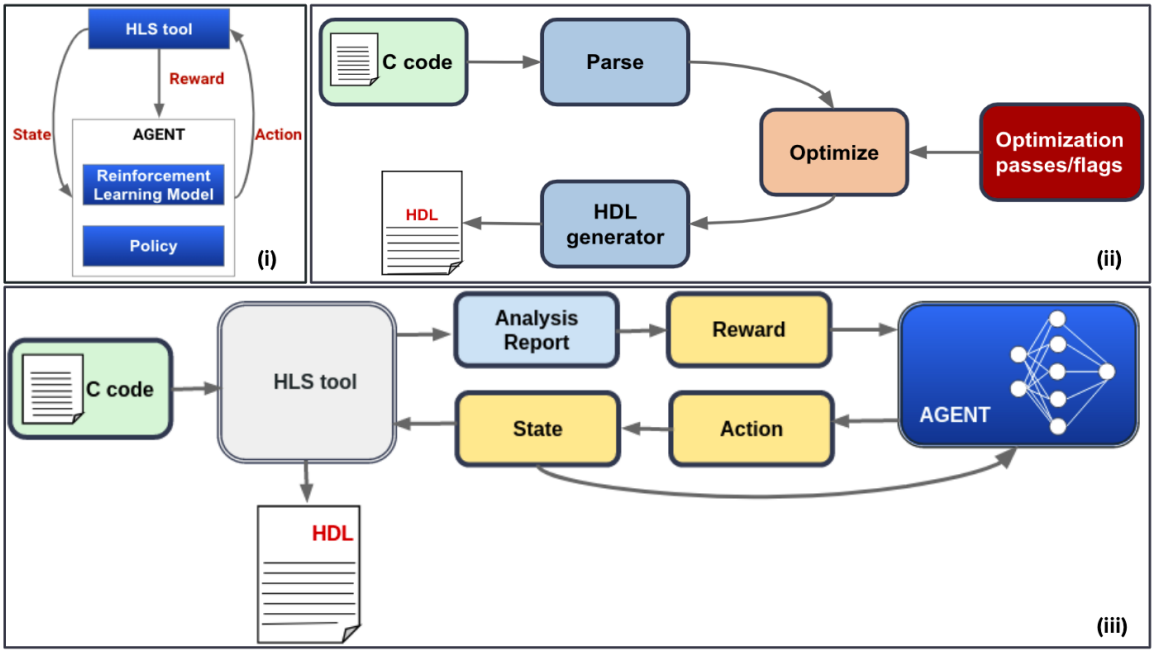

Figure 6·1: Overall Block Diagram: (i) - generic Reinforcement Learning (ii) - generic High Level Synthesis flow (iii) - combined, RL based HLS

HLS compilation flows involve: 1) using a *Parser* to convert HLL codes into a generic Intermediate Representation (IR), 2) running an *Optimizer* on the IR that use a series of codes transformation passes to optimize the code, and 3) mapping optimized IR code fragments to hardware blocks and adding an appropriate interconnect using an *HDL Generator* by replacing the normal CPU-specific back end of the compiler. The optimization step (2) is critical since it can improve hardware quality by increasing parallelism and reducing dependencies/hazards in the code structure [158].

6.3.2 RL based HLS compiler tuning: Workflow

The overall reinforcement learning framework for a high level synthesis compiler is illustrated in Figure 6·1. Reinforcement learning in this case requires specifying and tuning several components:

i) an *action space* that specifies appropriate actions that are suggested by the agent. Code optimization in compilers such as LLVM, is implemented as Passes.

These operate on the intermediate representation(IR) of the code and either analyse or transform portions of the program [173]. An action is usually the pass to apply next. Each pass transforms the IR into a valid yet, modified IR.

ii) an *observation space* that characterizes the environment and provides its state representation after an action is applied. In this case, it can be (a) a statistical analysis of the IR into a vector of features such as information extracted from the code about the loops, arithmetic operations, memory blocks etc. [109], (b) a runtime profiling of the program source code to dynamically characterize the system [81], (c) a graph based modeling of the feature set using Abstract Syntax Trees (ASTs), Control and Data flow graphs (CDFGs), graph based embeddings of Single Static Assignment (SSA)[93] (d) DNNs, LSTM or other neural nets to characterize the target code into a vector of features [92, 189], (e) A vector of applied passes (action history) or a histogram of applied passes (action histogram) to define the state of the environment (f) a concatenation/tuple of multiple aforementioned states.

iii) a *reward computation mechanism* that provides feedback to the agent in terms of the efficacy of the applied action and configures the frequency of reward computation. For high level synthesis this is usually dependent on the HLS tool used [18, 142, 168]. Most prior work have used cycle count as a reasonable metric for reward. However, other metrics such as post-routing Frequency, wall clock times, area and resource utilization details etc can also be used for comparison [140]. Cycle count can be estimated a) using software profiling as in Legup [79] b) simulating VHDL/Verilog code using tools such as Modelsim [144] or c) estimating from the Control Flow Graphs (CFGs). For the overall reinforcement learning framework, the reward has to be structured carefully for example, each good intermediate *step* of training returns a less negative reward, while a bad intermediate step returns a more negative reward, and achieving the goal by the agent returns a large positive reward.

This ensures the agent learns in an effective manner and does not get trapped in a vicious cycle while trying to maximize its rewards.

iv) a *policy* that defines the strategy learned by the reinforcement learning agent to decide the next action based on the current action's reward and state values i.e. a decision rule defined by its interactions with the environment.

v) the *environment* which in this case, embodies the application program and its various configurations. It also encapsulates the High Level Synthesis tool that supports user specified custom optimization passes. HLS is like a compiler however, the difference between CPU-targeting compilers and HLS environments is that in case of the latter the "instruction set" is not fixed and can be created alongside the logic which then uses these instructions. It has a front end where high level language (HLL) code is converted to language independent format such as intermediate representation (IR). At the middle-end, optimizations are applied to modify the IR and generate a new one. At the back-end, the HLS tool also contains some mechanism of returning performance estimates (clock cycle count, resource utilization etc) to execute the application code.

While numerous efforts have used reinforcement learning for compiler space, there is still a need to understand the options available in hyper-parameter exploration of the compiler reinforcement learning framework i.e. optimizing *step* of training, *episode* length, and a quantitative analysis of the state, action and reward space for high level synthesis compilers in particular. Our goal, is to highlight this research area and make valuable contributions. This work is presented in detail in Chapter 6

6.4 Learning Strategies

In this section, we start off by discussing the base learning strategy. Next we discuss the 4 additional strategies implemented in our work. Finally we talk about learning

quality metrics that can be used to compare the impact of these strategies.

6.4.1 Base Learning Strategy

Our base learning strategy, based on Autophase [141], is given below.

- Environment: The environment is composed of Legup HLS tool [168] and the specific application. It outputs a count of the number of clock cycles needed to execute the target application.

- Agent: Proximal Policy Optimization(PPO) is used as the reinforcement learning agent [219].

- Action: Each action represents a single optimization pass. This appended to an ordered list of actions for the episode, which in turn is used during compilation.

- State: A histogram of applied passes is used to represent the state - this is referred to as an *action histogram*. To implement this histogram, a list equal to length of passes is initialised and each element of the list is incremented and updated when a pass corresponding to the pass number is applied. The agent learns to map the distribution of number of passes applied, to the next pass that should be applied, in order to maximize the averaged/expected (across episodes) sum of rewards across time-steps in an episode.

- Reward: The reward is defined as the difference in cycle count between the previous step of training and the current step - lower cycle count thus results in a higher reward value. Reward is set to a default value of 0 for each step, except for the last step in the episode in which the actual clock cycle count is obtained from the HLS tool. The "previous step" reward value for the first step is set to the cycle count for the -O0 flag. *Maximum Reward* is defined as the highest value of reward obtained by any episode during training.

- Episode size: The episode size is fixed to 45 passes. This also places an upper bound on the size of action histogram.

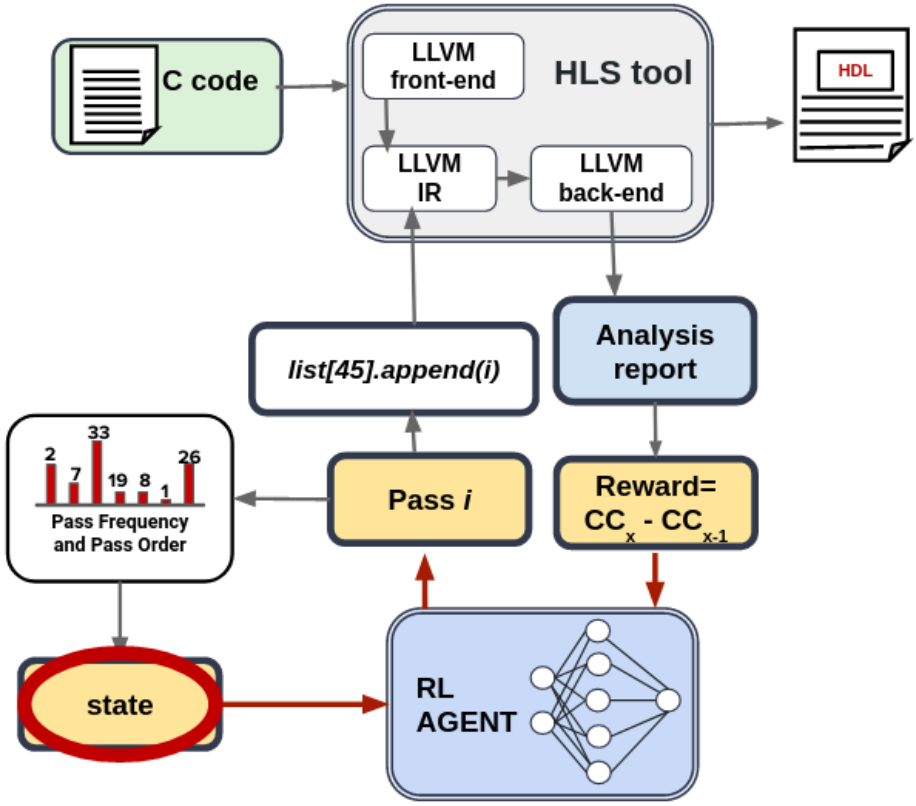

Figure 6·2: Strategy 1: Pass Ordering. Here we impact the state by passing information about the pass order to the agent and evaluating the impact on its learning policy.

6.4.2 Strategy 1: Pass Ordering

In a typical set of compiler optimization passes, both the repetition and ordering of individual passes impact the outcome of the optimization process. In the base learning strategy above, pass repetition is available as the action histogram while pass ordering is available as a list that is passed to the HLS tool during the compilation process. This means that, while both pass repetition and order are factored in during reward

calculation, only the pass repetition is used to represent state. This means that the same state can potentially correspond to substantially different reward values - these collisions can potentially negatively impact learning. To address this, we propose a new learning strategy that explicitly takes pass ordering into account. This can be done using two methods.

In Method 1, instead of using *action histogram* as the observation space, we use *action history* to learn the optimal pass ordering. Here, each value in the observation space is the actual pass that was applied at the corresponding step of training. Just like the action histogram, a fixed number of passes are applied in each episode and thus the observation space has known bounds.

In Method 2, the reward computation frequency is varied while the observation space continues to be the *action histogram*. In the base learning strategy, the actual reward from the HLS tool is calculated once per episode while all other steps use a default reward value. This corresponds to a reward frequency of 1, and only assigns a non-zero reward to the observation space represented by a complete histogram. In order to vary the reward frequency, we evaluate additional frequency values. For a reward frequency of less than the episode size, the HLS tool is invoked to give the cycle count after every fixed number of actions. In this case, instead of ordering between individual passes, we factor in the ordering of multiple sets of passes. For example, for a reward frequency of 2 and maximum episode size of 45, we get the reward from the HLS tool after 23 and 45 passes. For a reward frequency equal to the maximum episode size, reward is computed every step once an action is applied. In this case the complete ordering information for individual passes is factored in (represented by incremental changes to the action histogram).

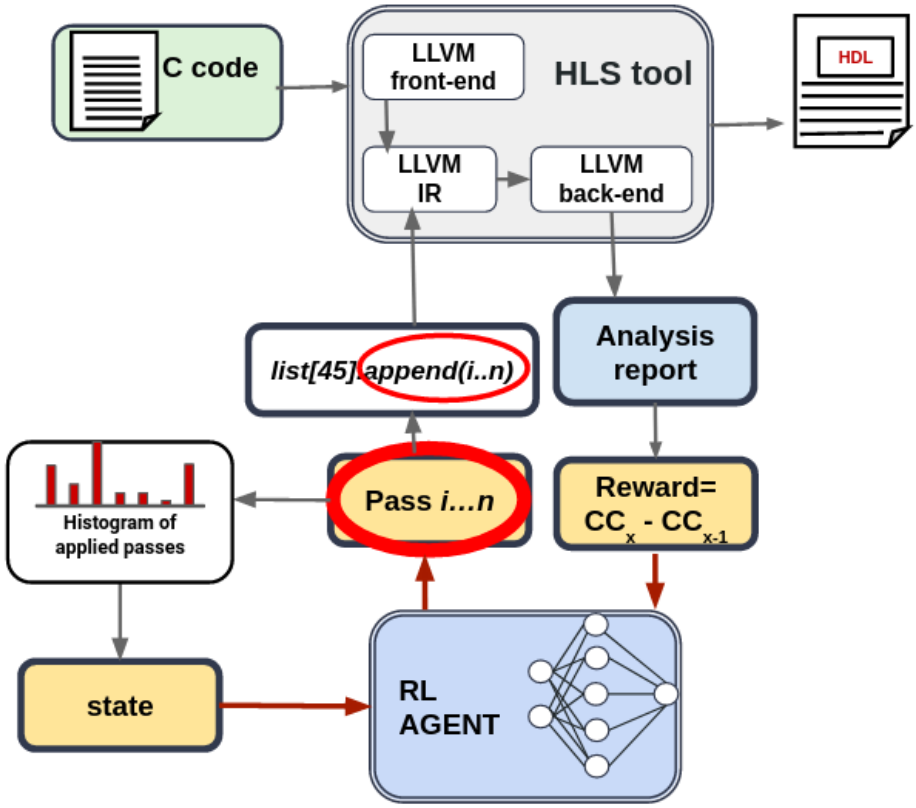

Figure 6·3: Strategy 2: Action Tuples. In this case multiple actions are applied as opposed to a single action.

6.4.3 Strategy 2: Action Tuples

As discussed in Strategy 1, pass ordering has an impact on learning and can be factored in by modifying the observation space and reward frequency. Another approach to factoring in pass ordering is through action tuples. Unlike Strategy 1, which focuses on learning a global pass ordering, this strategy is aimed at determining groups of passes that work well together i.e. local pass ordering. To achieve this, we modify the action space to represent a tuple of passes as opposed to individual ones. That is, at each step, the agent predicts multiple passes to apply. The maximum number of applied passes is kept (approximately) the same, which means that the episode size is reduced by an appropriate factor based on the size of the action tuple. Having a

large tuple size for the action space allows the agent to learn which passes work well together and how to apply these tuples, which in turn can improve learning. However, if the tuple sizes become too large, the agent can take substantially longer to learn them. In the extreme case, this would be equivalent to trying to predict the entire set of pass orderings in a single step - the neural net for this can become fairly large.

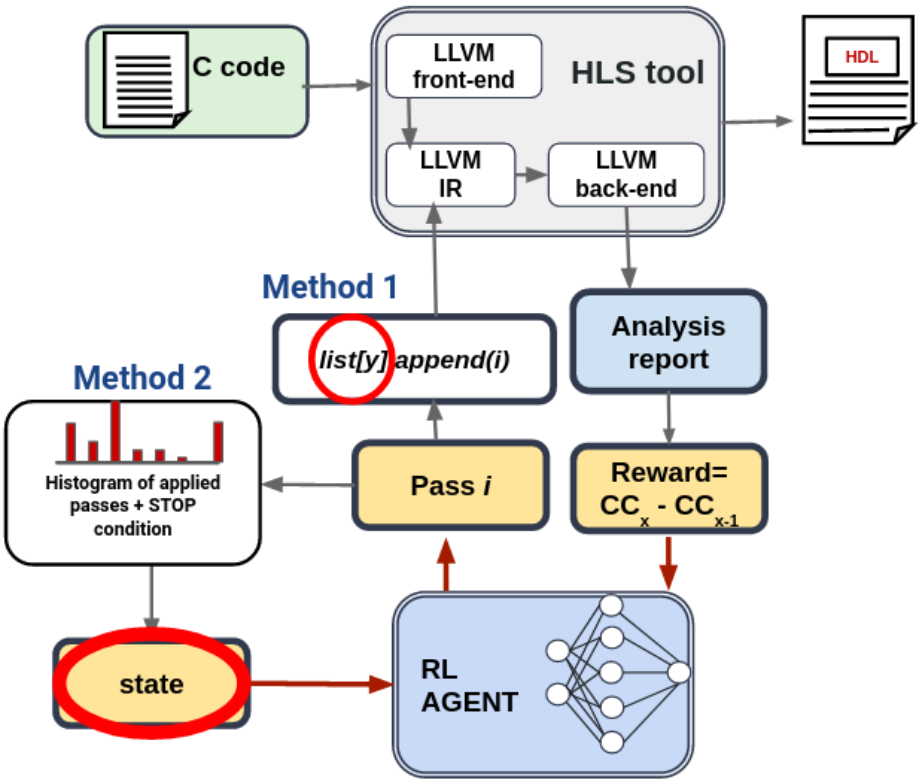

Figure 6·4: Strategy 3: Episode Sizing. In this case the number of steps that constitute a single episode of training are varied to evaluate the imapct on learning quality.

6.4.4 Strategy 3: Episode Sizing

The size of an episode can also impact the learning quality since it determines the dimensions of the action histogram and the maximum number of passes that are applied. Having a smaller episode size means the upper bound on the size of the

action histogram is lower, which can improve learning. However, the entire sequence of optimal passes must be specified within the limited number of steps. This not only limits the room for redundancy or sub-optimal passes, but it might not be possible to do so if the episode size is smaller than the size of the optimal pass ordering sequence. On the other hand, larger episode sizes have a greater margin for sub-optimal passes, but can have drawbacks such as greater combinations of action histograms. There are two methods for varying episode size.

In Method 1, we use static episode sizing. Similar to the base learning strategy, we fix the episode size for the duration of learning. In Method 2, we use dynamic episode sizing. Here, we train the agent to learn not only the pass frequency that maximises reward, but also the number of passes to apply within an *episode*. We insert a *stop* condition within the list of possible passes. Whenever, a stop pass is suggested by the agent, the training episode is terminated and the environment is reset after computing the reward. This reward is then mapped to the *action histogram* of the passes applied in that episode before the *stop* pass. The expectation is that the agent will learn the shortest sequence of passes that gives the highest reward.

6.4.5 Strategy 4: -O3 Backend

In the above strategies, our goal has been to learn the optimal pass ordering for the HDL generator back end. That is, what is the optimal code structure that allows the HDL generator to map code fragments and CFGs to high quality hardware. It assumes that the CPU and FPGA back ends are completely orthogonal and an entirely new optimization strategies must be learnt. However, this is not always the case and there is some potential overlap in how CPUs and FPGAs leverage different forms of parallelism.

This strategy aim to reuse the default CPU optimization strategy of a compiler instead of deriving a completely new one from scratch. Specifically, this is done by

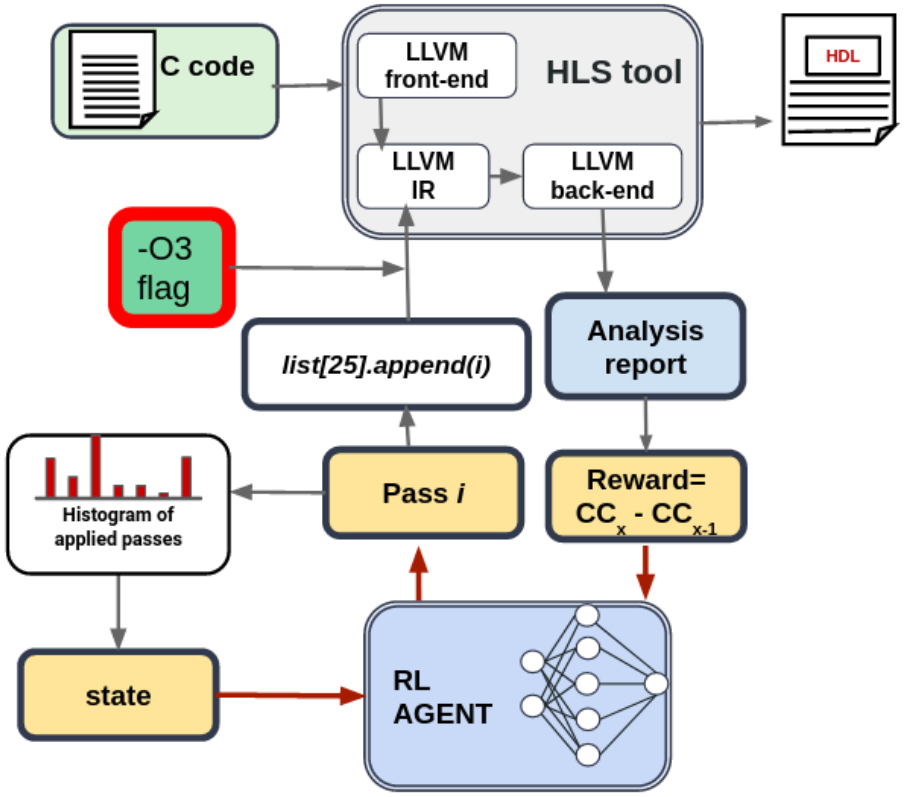

Figure 6·5: Strategy 4: -O3 Backend. In this strategy -O3 flag is applied at the end of each episode to train the agent to perform better than the compiler standard of -O3 level.

redefining the back end and, instead of targeting the HDL generator, learning the code transformations that enable the -O3 flag to be effective. Implementing the -O3 back end involves adding the -O3 flag to the action histogram before the HLS tool is invoked for clock cycle calculation. We also reduce the episode size to 25 (instead of the default 25) since we are not learning a complete optimization strategy.

6.4.6 Learning Quality Metrics

Figure 6·6 illustrates the different metrics used to evaluate learning quality for given training run.

i) Speedup over -03: This is defined as the ratio of the maximum reward obtained

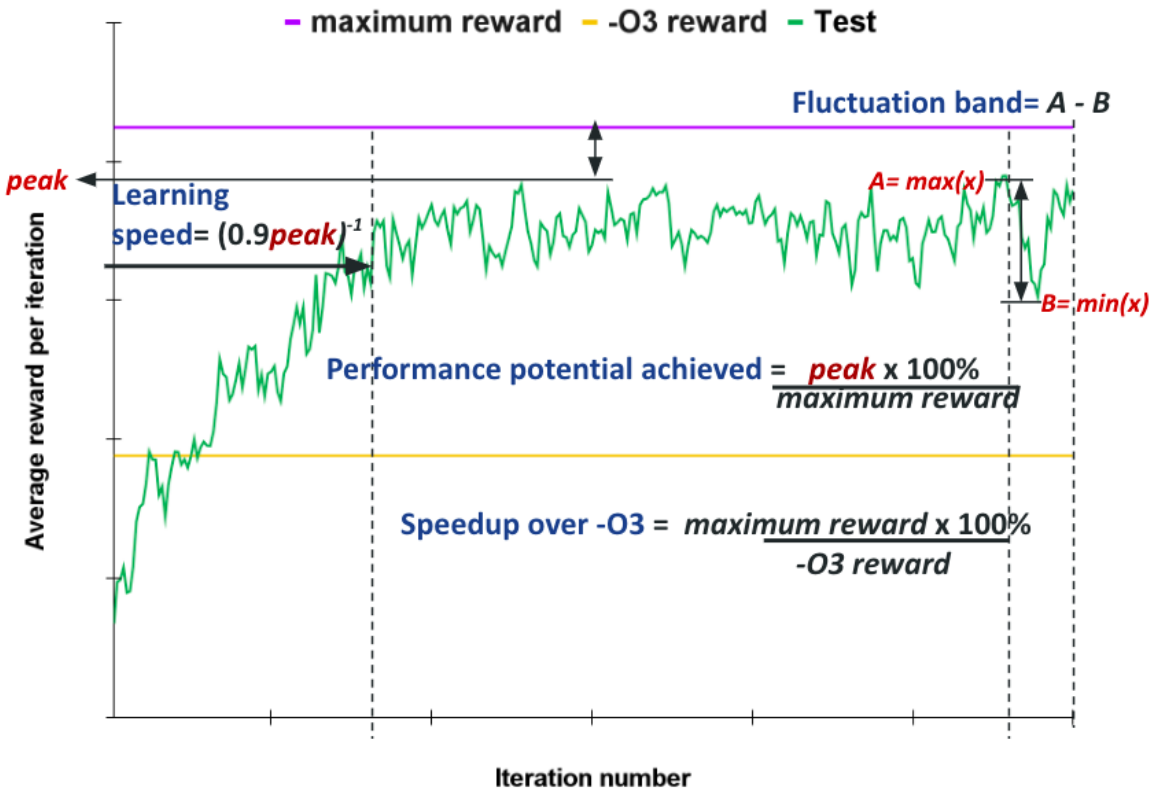

Figure 6·6: Learning quality metrics and how they are calculated

by any episode during training, and the reward from the -O3 flag.

i) Learning speed: This is defined as the number of iterations taken to reach 90% of the peak reward value. Peak reward value is defined to be the highest average reward value achieved across all training iterations.

ii) Performance potential achieved: This is defined as the ratio of the peak reward to the maximum reward obtained by any episode during training.

iii) Fluctuation band: This is the difference between the maximum and minimum average reward observed in the last 7% iterations of a training run.

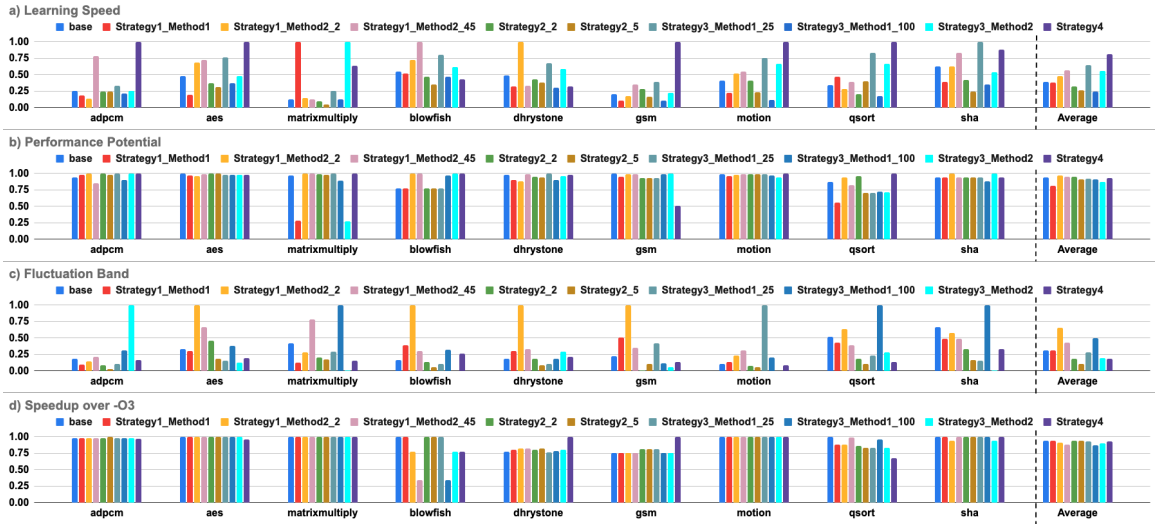

Figure 6·7: Normalized Results for each metric with respect to the standard application and strategy used. A higher learning speed, higher performance potential, lower fluctuation band and a higher speedup over -O3 are desirable.

6.5 Experimental Results

6.5.1 Experimental Setup

The framework is setup using Open AI Gym (0.21.0) to provide the unified environment interface for the reinforcement learning framework, Ray (1.7.0) to provide the unified API for reinforcement learning and its RLlib library to provide the agent interface. Tensorflow (2.6.0) is used for machine learning tasks together with Keras (2.6.0) to provide the neural net API for Python (3.9). To ensure standardization, each test is run using *2 workers* on a standard multi-core CPU, a total training time of *300 iterations* and the same initial *seed* value for random number generation. The same 9 benchmarks used in Autophase [141] are evaluated in this work. These include 6 CHStone benchmarks[136]: adpcm, aes, blowfish, gsm, motion and sha; and 3 benchmarks from Legup examples: matrixmultiply, dhrystone and qsort.

Similar to Autophase, we use Legup [79] as the HLS tool, which is built by modifying the LLVM 3.5 compiler. While we are constrained in using an older version

of the LLVM compiler due to Legup, the methods for learning effective optimization strategies and generating high quality HDL code are applicable to the latest versions as well.

6.5.2 Baseline Strategy Validation

To validate our baseline implementation, which is based on Autophase, we run each of the 9 benchmarks evaluated in the paper. The *maximum reward* for each benchmark is computed by letting it run at the baseline configuration for around 10,000 iterations, and then setting the maximum reward as the highest performance achieved. The means of maximum rewards we obtained as a results of the above matches the 26% efficiency over the -O3 flag of the LLVM compiler stated by the authors in [141].

6.5.3 Aggregate Results

Figure 6·7 shows the results for each standard application used and every strategy investigated. Note that results for each quality metric are normalized with respect to each application. The average gives the arithmetic mean for each strategy over all applications. Below we discuss our findings with respect to each learning quality metric.

Table 6.1: Best Strategy in terms of each Benchmark and Metric

Benchmark	Learning speed	Performance potential	Fluctuation band	Speedup over -O3
adpcm	Strategy_4	*multiple*	Strategy2_5	Strategy2_5
aes	Strategy_4	*multiple*	Strategy3_Method2	All except Strategy_4
matrixmul	*multiple*	*multiple*	Strategy3_Method2	*all*
blowfish	Strategy1_Method2_45	*multiple*	Strategy3_Method2	*multiple*
dhrystone	Strategy1_Method2_2	*multiple*	Strategy2_5	Strategy_4
gsm	Strategy_4	*multiple*	Strategy2_2	Strategy_4
motion	Strategy_4	Strategy_4	Strategy3_Method2	*multiple*
qsort	Strategy_4	Strategy_4	Strategy2_5	*multiple*
sha	Strategy3_Method1_25	*multiple*	Strategy3_Method2	*multiple*

Learning Speed

Figure 6·7a shows how learning speed is impacted by strategies. Higher learning speed is more preferable since it means the agent can train faster. We note that on average Strategy 4 takes the least number of iterations to reach 90% of the peak reward value achieved. Overall up to 23× improvement in learning speed is possible if the correct learning strategy is selected versus an inefficient strategy.

Performance Potential Achieved

From figure 6·7b we can see that that most strategies exhibit reasonable performance potential however, some strategies notably give poor performance for certain applications. For *matrixmultiply*, setting the state as action history and using dynamic sizing gives drastically poor performance. However, for other applications such as *adpcm, blowfish, sha* these result in high performance and even better than the baseline. Overall up to 4× improvement in performance potential is achievable by selecting the best strategy.

Fluctuation Band

A high fluctuation band means that the ripple in final steady state reward is high hence the confidence value is low. From figure 6·7c we can see that on average Strategy2_5 gives the best results in terms of the fluctuation band. On average Strategy1_Method2_2 results in the highest ripple. For *motion*, it is possible to largely eliminate the ripple by selecting Strategy3_Method2. From actual values, we note that more than 7700× improvement in fluctuation band is possible through the selection of reasonable strategy.

Speedup over -O3

As seen in figure 6·7d, up to 3% improvement in speedup over -O3 is possible by selecting the right strategy. Most strategies give a reasonable speedup over -O3. However, depending on the application, some like Strategy1_Method2_45 and Strategy3_Method1_100 give pretty poor performance for *blowfish* benchmark. Overall up to 3× improvement in speedup over -O3 is possible by choosing the correct strategy.

Strategy Selection

Table 6.1 gives the best strategy for each benchmark and metric. Where more than one strategies favor the particular test case, *multiple* is seen. For *matrixmultiply*, all strategies give a speedup over -O3 hence showing that for this benchmark, the standard compiler optimization strategy is highly inadequate. For *aes*, Strategy_4 leads to a degradation of the metric, speedup over -O3. An important deduction is that there is no one strategy that fits all. Different strategies favor different design criterion and it is up to the developer to choose according to their requirements.

6.6 Conclusion

In this work, we have explored a number of learning strategies that enable developers to customize their design according to their respective learning goals. To do this, we have proposed 4 additional learning strategies. We have also analyzed the results using 3 additional metrics that encapsulate developer goals more effectively when compared with the single speedup over -O3 standard metric. Results show there is no one size fit all solution: different learning strategies give best results in terms of different metrics. Choosing just the right strategy can give an improvement of 23× in learning speed, 4× in performance potential, 3× in speedup over -O3 and has the

potential to largely eliminate the fluctuation band from the final results.

Chapter 7

A Neural Network Based GCC Cost Model for Faster Compiler Tuning

7.1 Introduction

Even after decades of efforts on improving the code optimization process, there are continually new improvements [23, 109, 119, 150, 163, 176, 223, 225, 232, 249]. For example, in the last few years, machine learning (ML) has been employed to build models used within the compiler to help make optimization decisions for any given program.

Building an efficient ML model requires two things. First, efficient feature engineering is required to derive quantifiable properties to characterise a given code and then iteratively refine them to improve the accuracy of the model. Second, a training data set must be built and fed into a learning algorithm. Here, a training data set comprises a tuple of code features, performance values, and transformations that allow the ML algorithm to build its correlations and enable it to make predictions for an unseen code.

The problem addressed here is that computing the impact of transformations on a code, i.e., the performance values, is an expensive process. It involves invoking the downstream compiler and may take a significant amount of time. And often performance characteristics of the target hardware is altogether unavailable. Due to this cost bottleneck, models that can predict optimization decisions across a wide

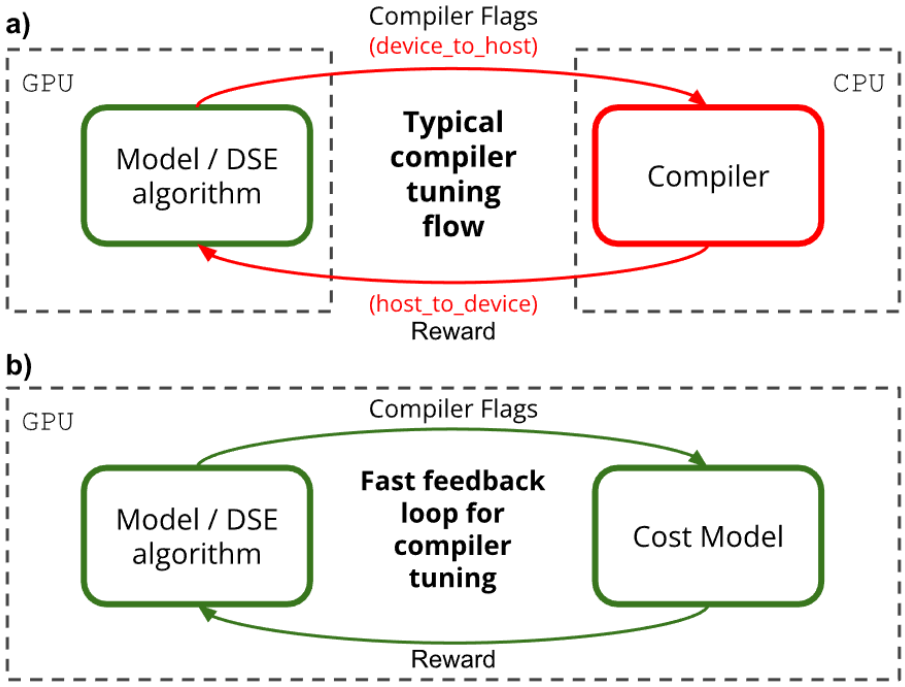

Figure 7·1: Circumventing the training time conundrum of compiler tuning models and Design Space Exploration (DSE) algorithms by replacing downstream compiler with neural net based cost model

range of codes require days to train [133, 169, 254].

In particular, while cost functions can be designed to quickly predict performance values, thereby substantially reducing the training times, simultaneously achieving accuracy from them is challenging. In the past several groups have tackled this limitation of compiler optimization by designing manual cost functions [164, 262]. However, this requires expert knowledge about the specific target architecture and can take months to tune [266]. With the proliferation in deep learning, some groups have proposed neural network based cost functions. The cost model implementations in Halide [9] predict runtimes for image processing and deep learning programs, but is limited in scope to Halide programs. The cost model implementation for the Tiramisu compiler[61] predicts speedup in runtime, but focuses on loop transformations.

In this chapter we propose a neural network based cost function that addresses

the limitations of previous work. We demonstrate accuracy in terms of mean absolute percentage error that is roughly 2× better than previous efforts. Moreover, to the best of our knowledge, our work is also the first employing a neural network based cost model for a production compiler such as GCC. While LLVM is a popular choice in academia, GCC is default choice for a large majority of significant software projects. We are also one of the first efforts in predicting code size as a performance metric, rather than standard schedules and runtime. Code size has been used as the optimization criteria in a number of recent works for compiler optimization [98, 169, 178, 203, 212, 221, 232, 265]. This is partly because code size reduction, firstly, leads to cost reductions in terms of storage, bandwidth and memory, all critical for today's resource constrained applications [211]; and secondly is platform-independent, deterministic, and relatively noise-free, and hence an ideal target for compiler optimizations [94, 248, 254].

A major challenge has been creating the training set for a cost function since this requires code features for labeling each transformation set and its code size reduction. Feature extraction for GCC is not straight-forward: the last major work to extract static features from GCC codes was Milepost [109] and their framework does not work beyond GCC 6+ [91]. We tested other alternatives such as a Python plugin by GCC developers [100]. However the limitation of most prior works is that they are not updated to recent GCC versions and hence are not upgraded for vectorization, interprocedural optimizations, or new architectures. For LLVM, there are many efforts that allow feature extraction [94, 133], including a custom pass within LLVM [172]. There is a need to develop similar feature extractors at different abstraction levels for GCC's latest versions (11-13).

We have therefore developed a *GIMPLE* (Intermediate Representation for GCC) parser that allows users to extract more than a 100 function-level static features.

In addition, users can choose among types of feature extractions: (i) After which GCC pass the features should be extracted? and (ii) Which subset of features are important? We also compare the subset of features extracted with state-of-art work (Autophase [133]) and demonstrate that our framework allows feature selection according to (i) and (ii) that train neural net with better accuracy in predicting code size. To the best of our knowledge, this is the first work to explore static feature extraction in such depth and with such success in predicting accuracy.

The specific contributions of this paper are:

- An automated framework for extracting over 100 function-level static features from GCC's intermediate representation, *GIMPLE*, that allows users to configure their optimization level and optimization flags, select the subset of features and the level of feature extraction (i.e., to analyze tree dumps after a selected compiler pass during the optimization pipeline).

- A neural network based cost-model that takes these code features and GCC flag transformations, and outputs the binary size prediction for GCC.

- An implementation and evaluation of the proposed cost model and mean average error comparison with the state of art.

- A demonstration that models trained using features extracted after individual passes in the optimization pipeline have different accuracy. Also, that by using the comprehensive list of features from our framework, better prediction accuracy is achieved than with features used in prior efforts.

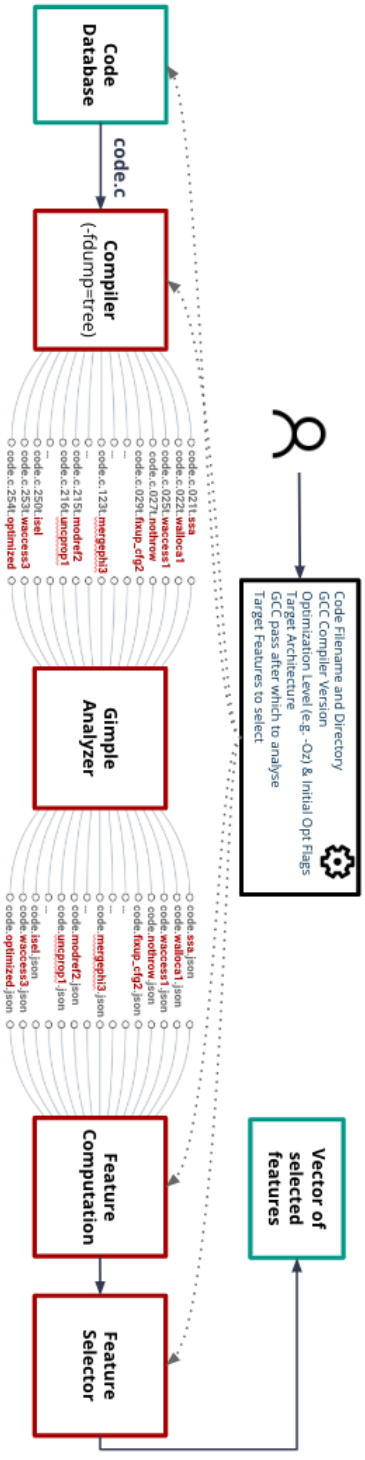

Figure 7·2: Feature Engineering Workflow

Figure 7·3: Function level features that can be extracted. The color codes show features that have also been used in two prior state-of-the-art works [109, 133]

7.2 Related Work

One of the earliest works in ML-based compiler cost models for GCC was MILEPOST [109]. It trained a k-nearest neighbor model using engineered static features and predicted the best combination of GCC flags for unseen applications. Our work trains a deep neural net to predict code size given a much larger pool of code features and set of GCC flags compatible with latest versions of GCC. Ithemal [181], Granite [242], Facile [5] and DiffTune [210] are basic block throughput estimators. Their limitation is that they do not support loop level transformations and memory accesses.

Static code analyzers like LLVM-MCA [174], OSAKA [166], and IACA[146] are not always accurate and sometimes give large errors [4, 85, 171]. Similarly early efforts by [149] developed a multilayer perceptron trained on SMG2000. Our approach allows various algorithms.

An artificial neural net to predict tile size performance was developed by [208]. Our work encompasses broader performance characteristics and not a specific trait of the program. Similarly, the static features-based performance model such as [103], and performance counter based feature vectors by Park et al. [198, 199] predict runtime speedup over a limited set of applications. Our work predicts code size with reasonable accuracy. To the best of our knowledge, only [188] have looked at neural nets for code size. However, the scope of their work encompasses only the code duplication and only for GraalVM compiler. We look at a much broader action space of GCC flags.

Static code features for compiler optimization were developed mainly by Milepost [109] and Autophase [133]. Most other groups define their subset depending upon the use case, e.g., loop tiling [61] or inlining [254]. We have curated a large list of static features that uniquely characterise different aspects of a code and can also be extracted at intermediate compilation steps of the compiler's optimization pipeline.

7.3 Feature Engineering

Figure 7·2 shows the framework for static feature extraction. The output is a vector of those feature values that the user specified for the target code. Figure 7·3 lists all of the function-level features that can be extracted from the framework. Features extracted by prior state of art work are shown in color. Green color code specifies features that have been considered by Autophase[133] and later used in many recent compiler optimization works [94, 169]. Purple shows features that Milepost [109] used and Cyan color code shows features that both Milepost and Autophase included in their subset to describe the code. Note some features have been omitted either because they are not relevant for GCC, or for latest versions, or are not appropriate for describing function-level attributes. Below we describe the important blocks.

7.3.1 User Configurations

Four sets of parameters are specified by the user: (i) code name, code directory, and build directory; (ii) details about the compiler such as the GCC version (tested for versions 11-13) such as, optimization level (e.g., -O3, -Oz), other flags or command options as specified in [118], the name of the GCC pass applied in the middle-end optimization pipeline for GCC, and the pass after which to analyse the tree dump for features; (iii) target architecture as specified in [112]; and (iv) target features, which should be a subset of features of 7·3.

7.3.2 *GIMPLE* Analyzer

In the first step, the GCC tree representation of functions is extracted. The advantage of using *GIMPLE* is that it is language-independent and its context is simple for optimization passes to operate on [183]. It is also relatively stable with the changes across compiler versions being minimal. GCC can dump out *GIMPLE* representation after

every optimization pass it applies. Some of the passes vary with the compiler flags and command options. However, for a fixed optimization level -O(0,g,1,s,2,3,z,fast), the changes are minimal and the framework emits all options available to the user for feature analysis. *GIMPLE* tree dumps for the relevant compile options of the specific C code (i.e., optimization level and GCC flags) are dumped into the build directory. These dump files are named according to the pass after which these dumps are generated. For example, code.c.021t.ssa gives the *GIMPLE* tree dump after ssa pass is applied by the compiler. 21 is the static pass number for the ssa pass and can vary across compiler versions. code.c.130t.dom2 is the tree dump after applying the dominator tree optimizations to the code. 130 is the static pass number for the dom2 pass and can vary across compiler versions. code.c.115t.vrp1 is after value range propagation is applied to the code. All tree dumps are then parsed by the *GIMPLE* Analyzer. The *GIMPLE* Analyzer stores all extracted information from the tree dump into a dictionary. We have tested it across a large range of codes (including elfutils and linux kernels) to ensure the *GIMPLE* Analyzer has the ability to handle corner cases. This information contains details about the functions within the C code, their arguments and types, local declarations with the functions and their values and types, basic blocks in the control flow and all the instructions, phi nodes, and edge source and destination within the basic block. It also decodes all the operands, expressions and variables used within every instruction in a basic block. This information is decoded for tree dumps after every pass and stored in a code.*pass_ name*.json file.

7.3.3 Feature Computation

The Feature Computation block contains one function for every feature from 7·3. These use the information from the output of the *GIMPLE* Analyzer and compute the relevant feature. For example, to compute the number of back-edges in the control flow graph after a vectorization pass is applied, the file code.c.veclower21.json is read

and, from the information on edge source and destination for each basic block, an adjacency matrix is computed. This is then used to calculate the back-edges.

7.3.4 Feature Selector

The Feature Selector uses the target features selected by the user in the configuration file and outputs a vector of those feature values from a list of all available features. The Feature Selector has three parts. First, it allows the user to select features using Principal Component Analysis (PCA). In this case, the user specifies a list of applications, the GCC pass after which to analyze code and the variance threshold/ number of features to extract. The *GIMPLE* Analyzer and Feature Computation blocks are invoked to extract features after the selected GCC pass for each C code. The results are then analyzed using PCA. PCA is a linear dimensionality reduction technique that transforms the p features into a smaller k (k«p) by taking advantage of the correlations between the input variables in the dataset. Second, it uses features selected in prior works [109, 133]. And third, it shortlists important features using lasso regression analysis. Lasso or L1 is a powerful regularization technique in statistics that shrinks the coefficients of less important features to 0 thus reducing variance in ML models and enabling better learning. A detailed analysis of results from these techniques is presented in Section 7.6.

-fassociative-math -fno-trapping-math -fno-signed-zeros, -fconserve-stack, -ffinite-loops, -fgcse-after-reload, -fgcse-las, -fgcse-sm, -fgraphite, -fgraphite-identity, -fipa-cp-clone, -fipa-pta, -fira-loop-pressure, -fisolate-erroneous-paths-attribute, -fkeep-gc-roots-live, -flimit-function-alignment, -flive-range-shrinkage, -floop-interchange, -floop-nest-optimize, -floop-parallelize-all, -floop-unroll-and-jam, -fmodulo-sched, -fmodulo-sched-allow-regmoves, -fnon-call-exceptions, -foptimize-strlen, -fpeel-loops, -fpredictive-commoning, -frename-registers, -freschedule-modulo-scheduled-loops, -fsched-pressure, -fsched-spec-load, -fsched-spec-load-dangerous, -fsched-stalled-insns, -fsched2-use-superblocks, -fschedule-insns, -fno-section-anchors, -fsel-sched-pipelining, -fsel-sched-pipelining-outer-loops, -fsel-sched-reschedule-pipelined, -fselective-scheduling, -fselective-scheduling2, -fsignaling-nans, -fsplit-loops, -fsplit-paths, -fsplit-wide-types-early, -ftracer, -ftree-cselim, -ftree-loop-distribution, -ftree-loop-vectorize, -ftree-lrs, -ftree-partial-pre, -ftree-slp-vectorize, -ftree-vectorize, -funconstrained-commons, -funroll-all-loops, -funroll-loops, -funswitch-loops, -fvariable-expansion-in-unroller, -fversion-loops-for-strides, -fweb, -fno-aggressive-loop-optimizations, -fno-allocation-dce, -fno-asynchronous-unwind-tables, -fno-auto-inc-dec, -fno-bit-tests, -fno-branch-count-reg, -fno-caller-saves, -fno-code-hoisting, -fno-combine-stack-adjustments, -fno-compare-elim, -fno-cprop-registers, -fno-crossjumping, -fno-cse-follow-jumps, -fno-dce, -fno-defer-pop, -fno-devirtualize, -fno-devirtualize-speculatively, -fno-dse, -fno-early-inlining, -fno-expensive-optimizations, -fno-forward-propagate, -fno-fp-int-builtin-inexact, -fno-function-cse, -fno-gcse, -fno-gcse-lm, -fno-guess-branch-probability, -fno-hoist-adjacent-loads, -fno-if-conversion, -fno-if-conversion2, -fno-indirect-inlining, -fno-inline, -fno-inline-atomics, -fno-inline-functions, -fno-inline-functions-called-once, -fno-inline-small-functions, -fno-ipa-bit-cp, -fno-ipa-cp, -fno-ipa-icf, -fno-ipa-icf-functions, -fno-ipa-icf-variables, -fno-ipa-modref, -fno-ipa-profile, -fno-ipa-pure-const, -fno-ipa-ra, -fno-ipa-reference, -fno-ipa-reference-addressable, -fno-ipa-sra, -fno-ipa-stack-alignment, -fno-ipa-strict-aliasing, -fno-ipa-vrp, -fno-ira-hoist-pressure, -fno-ira-share-save-slots, -fno-ira-share-spill-slots, -fno-isolate-erroneous-paths-dereference, -fno-ivopts, -fno-jump-tables, -fno-lra-remat, -fno-math-errno, -fno-move-loop-invariants, -fno-move-loop-stores, -fno-omit-frame-pointer, -fno-optimize-sibling-calls, -fno-partial-inlining, -fno-peephole, -fno-peephole2, -fno-plt, -fno-printf-return-value, -fno-ree, -fno-reg-struct-return, -fno-reorder-blocks, -fno-reorder-blocks-and-partition, -fno-reorder-functions, -fno-rerun-cse-after-loop, -fno-sched-critical-path-heuristic, -fno-sched-dep-count-heuristic, -fno-sched-group-heuristic, -fno-sched-interblock, -fno-sched-last-insn-heuristic, -fno-sched-rank-heuristic, -fno-sched-spec, -fno-sched-spec-insn-heuristic, -fno-sched-stalled-insns-dep, -fno-schedule-fusion, -fno-schedule-insns2, -fno-semantic-interposition, -fno-short-enums, -fno-shrink-wrap, -fno-shrink-wrap-separate, -fno-signed-zeros, -fno-split-ivs-in-unroller, -fno-split-wide-types, -fno-ssa-backprop, -fno-ssa-phiopt, -fno-stdarg-opt, -fno-store-merging, -fno-strict-aliasing, -fno-thread-jumps, -fno-toplevel-reorder, -fno-tree-bit-ccp, -fno-tree-builtin-call-dce, -fno-tree-ccp, -fno-tree-ch, -fno-tree-coalesce-vars, -fno-tree-copy-prop, -fno-tree-dce, -fno-tree-dominator-opts, -fno-tree-dse, -fno-tree-forwprop, -fno-tree-fre, -fno-tree-loop-distribute-patterns, -fno-tree-loop-im, -fno-tree-loop-ivcanon, -fno-tree-loop-optimize, -fno-tree-phiprop, -fno-tree-pre, -fno-tree-pta, -fno-tree-reassoc, -fno-tree-scev-cprop, -fno-tree-sink, -fno-tree-slsr, -fno-tree-sra, -fno-tree-switch-conversion, -fno-tree-tail-merge, -fno-tree-ter, -fno-tree-vrp, -fno-unwind-tables, -fno-exceptions, -fno-vect-cost-model

Figure 7·4: Flags used for binary size cost model

7.4 Dataset Generation

Since deep neural nets(DNNs) require a large amount of training data, we create a large dataset that is then used to train the DNN. A similar dataset is created for the test set. As a first step, selected features are extracted for the target configurations across 81 C functions. These functions are from a curated list of applications including linear algebra, image and signal processing, and sorting. They also include applications from the Polybench benchmarks suite[204], which has been widely used in prior compiler optimization work[11, 61]. The dataset used in this paper will be open sourced for the community. We split the functions into a randomly selected training set (74 functions) and test set (7 functions).

For each C function, we create 10,000 different random sequences of code transformations through GCC flags given in Fig7·4. Each random sequence of code transformation corresponds to on or off (0 or 1) for the corresponding flag. Then each of the C functions is compiled using each GCC flag sequence and the resulting binary size is measured from the compiled object file. Each vector in the dataset is then a tuple of the form [code features, sequence of code transformations/flags, binary size] with features and flags being model inputs and binary size being the expected output. Overall, we generated 740,000 training and 70,000 test vectors.

7.5 Model Training

We model the binary size estimation as a regression problem implemented using a neural net in PyTorch. The network architecture is fairly simple, with two dense hidden layers of size 512 each. The model uses the rectified linear unit (ReLu) activation function. We used the Adam optimizer with a learning rate of 10e-3 and Mean Absolute Percentage Error (MAPE) as the loss function. This is suitable for binary code size prediction since the target value is positive by design [61]. The other optimizer parameters are set with default values.

On average across multiple tests, we find that the best accuracy is achieved at around 800 epochs of training. The input features are normalized to the instruction count of the function. We tried min-max scaling and also standardizing features by removing the mean and scaling to unit variance. We noticed, however, that the learning of the model was better when normalized with respect to the instruction count. In addition to normalizing the input training data to the network, we also normalize the inputs to the layers within the network using the mean and variance of the values in the current batch, a technique called batch normalization [148]. This can improve the efficiency of the neural net and also improve the training speed. In

our model batch normalization is applied to the output of both hidden layers. We calculate the MAPE on both the test and training sets and validate that the values are close. This ensures that the model is not over-fit or under-fit and can generalize well on unseen data.

7.6 Evaluation

7.6.1 Methods

The model evaluation and data collection are performed on a single multi-core CPU + GPU node. The CPU is a 16-core AMD Ryzen Threadripper PRO 5955WX, with 196GB of RAM. The GPU is a NVIDIA GeForce RTX 4070, with 5,888 CUDA cores and 12GB RAM. GCC compiler version 13.2.1 is used to generate the training and test sets. For every compilation run, the -Oz flag is always applied; only the sequences of flags given in Fig. 7·4 are varied. The test applications used for our experiments were: *2mm* (polybench), *doitgen*(polybench), *heat3d* (polybench), *shellSort*, *boxBlur*, *dotProduct* and *jacobiEigenvals*.

Evaluating Model Accuracy: Model accuracy was evaluated using Mean Average Percentage Error (MAPE). This is given by:

$$\frac{1}{n}\sum_{i=1}^{n} \left|\frac{(actual\_byte\_size - predicted\_byte\_size)}{actual\_byte\_size}\right| \times 100\%$$

We have also used the Spearman rank-order correlation coefficient to measure the monotonicity between the measured and predicted test datasets. A high value shows that the two datasets are highly correlated.

Levels of Feature Extraction: A major advantage of our framework is that we can compute a large list of function-level features and also at intermediate points during the optimization pipeline of the compiler. As discussed in Section 7.3.2, this

means that we compile once using the -Oz flag and omit all tree dumps that GCC generates as it optimizes the code for -Oz. These tree dumps are labeled with the optimizing passes that the compiler applies during that compile. For each tree dump of the optimizing pass, we then compute the feature vectors as discussed in Section 7.3. As a next step, for features corresponding to every pass, we generate the training and test dataset as discussed in Section 7.4. The training dataset at pass *xyz* is used to train the cost model, and its prediction accuracy is recorded on the test dataset. In this way, we shortlist which features extracted after which pass within the optimization pipeline would result in the best cost function.

Selecting a Subset of Features: Our framework has the capability to extract over 100 features (describing the function) as given in Figure 7·3. To demonstrate the value of having such a comprehensive set of code features, referred to as *All Features*, we evaluate prediction accuracy with two methods of feature space pruning. The first method is to statically prune the feature space. This is done by using the feature set given in [133], which is the state-of-the-art work in compiler optimization. We refer to this feature space as *Autophase*. The second method is to prune the feature space at runtime. This is done using L1 or Lasso Regularization. Lasso regularization is applied to the weights of both the layers in the neural net and added to the overall loss computed for the training loop. We refer to this feature space as *Lasso*. We also tried other feature selection techniques such as PCA and Random Forests. However, these last techiques did not give any significant improvement in prediction accuracy of the cost model and hence have been omitted.

7.6.2 Finding the Best Level of Feature Extraction

Figure 7·5 shows the results. A lower MAPE means higher accuracy. The results are shown for different types of feature selections used to train individual cost functions for different passes: All Features from Figure 7·3, prior state-of-art, Autophase [133],

and using Lasso Regression. We see that the best accuracy for All Features and Autophase Features occurs at the ccp2 and ccp3 passes, respectively. This is the conditional constant propagation optimization that aims to simplify instructions and local scalar variables [111]. The numbers “2" and “3" mean that the respective pass is being applied for the 2nd or 3rd time during the optimization pipeline. For Lasso, features extracted for tree dumps after the waccess3 pass and then used to train the cost model give the best cost model accuracy. We find that waccess3 is one of the last *GIMPLE* passes to be applied in the optimization pipeline and that, for majority applications, it indeed gives improved feature values (such as lower Halstead difficulty, fewer local variables, and a smaller average number of instructions per basic block).

Table 7.1: Summary of results

	All Features	**Autophase**	**Lasso**
Best test MAPE	8.00%	10.14%	9.79%
Compiler pass	ccp2	ccp3	waccess3
Accuracy	92.0%	89.9%	90.2%
Spearman coefficient	0.98	0.97	0.97

7.6.3 Computing the Best Subset of Features

We ask: Which subset of the selected features' set is better at training the binary size cost model for GCC? From training individual cost models using features extracted after passes, we find that the least MAPE error occurs using the proposed comprehensive list of All Features. The MAPE of this cost model is 8%. If we train the cost model using features proposed by Autophase then the MAPE is 10%, and for features selected by Lasso it is 9.79%. These results are summarized in Table 7.1. The Spearman rank coefficient of the cost model is 0.98 when using All Features. This shows that the predicted and measured binary sizes on the test set are highly correlated. Figure 7·6 shows the accuracy of the cost model visualized as bar plot when used to evaluate the test dataset. The x-axis gives the relative error between

predicted and measured sizes, aggregated into buckets of size 10%. The y-axis lists the number of predictions made on the test set that lie in that range of relative error. We note that for "All Features/ccp2" the majority of the test cases have error less than 5%. In contrast, the relative error of "Autophase/ccp3" has higher variance.

7.6.4 Accuracy of individual test applications

Table 7.2 shows the prediction accuracy for individual applications in the test set, evaluated using the All Features/ccp2 model. The majority achieved >95% accuracy, while the lowest accuracy was 80%.

7.6.5 Model Accuracy Comparison with Prior Work

We compare the performance of other cost models trained for compilers such as Tiramisu and GraalVM. In [61], the authors have 16% MAPE in predicting speedups on full programs for the Tiramisu compiler. We achieve MAPE of 8%. In [188] the authors designed a cost model for the GraalVM compiler and quote accuracy in terms of 70% applications lying within 10% of the correct target value. In our case over 86% applications lie within 10% of the corrected predicted binary size.

7.6.6 Performance Improvements With Our Cost Model

We ask: How much speedup is possible by substituting GCC's binary size computation with the proposed neural net cost model? How much advantage can be extracted by using a GPU for the inference from the trained cost model? And, How many test cases (batch size) would be needed for the GPU inference to outperform the CPU?

To answer these questions, we measured the time to evaluate the binary size for a

Table 7.2: Average accuracy for test applications

shellSort	boxBlur	doitgen	heat3d	dotproduct	jacobi	2mm
80%	95%	95%	88%	94%	96%	96%

given tuple of C function and target compiler flags. Measurements were done for the following combinations: running the actual GCC compiler (CPU + GCC), running inference for the cost model on the CPU (CPU + Cost Model), and running inference on the GPU (GPU + Cost Model). For GPU inference, we varied the batch size from 2 to 1024. From Table 7.3, see that speedup of multiple orders of magnitude is achieved by using the cost model. Even if a compiler tuning workload is fully deployed on the CPU, it is still over 700× faster to use the cost model than the actual compiler. Moreover, in scenarios where multiple test vectors need to be evaluated in a batch, the GPU based inference will significantly outperform CPU.

Table 7.3: Performance improvement from cost model

Method	**Speedup over CPU + GCC**
CPU + GCC	1
CPU + Cost Model	705
GPU + Cost Model (Batch size=2)	134
GPU + Cost Model (Batch size=16)	1675
GPU + Cost Model (Batch size=1K)	134000

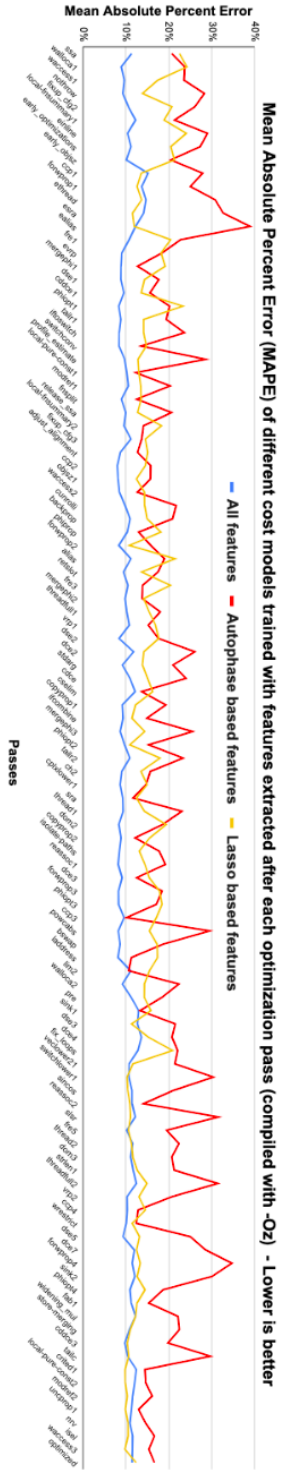

Figure 7·5: MAPE values after each optimization pass for three different types of feature selection: All Feature (Comprehensive), Autophase (Prior State-of-art Subset), and Lasso (Runtime Statistical Pruning in Neural Net).

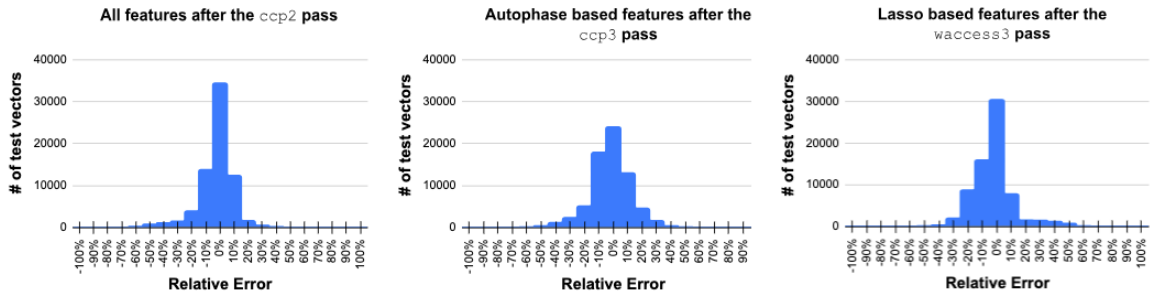

Figure 7·6: Percent error distribution for passes that give the best cost model accuracy.

7.7 Conclusion and Future Directions

In this paper, we present a GCC feature extraction framework that is able to generate a comprehensive set of code features. The intermediate representation used for it can be selected after multiple points in the compilation process. We also present cost model architecture for faster compiler tuning that achieves a high accuracy of 92%, and can deliver orders of magnitude speedup versus running the compiler.

Though we are able to achieve good numbers on MAPE and Spearman's correlation, we still believe we can improve our cost model by incorporating further representation learning to better generate embeddings from the feature vectors. We can also improve the structure of the neural net to enble better learning. Moreover, we can increase the number of applications both for train and test. We can also create a better pool of applications, with varied binary sizes and codes. We can identify which type of applications the model is not able to predict so well and then employ advanced techniques such as code similarity analysis or fine tuning over specific application sets to increase their prediction accuracies.

Chapter 8

Optimizing the Optimizer: A Practical Approach to Learning Compiler Heuristics

8.1 Introduction

A long-standing goal, which is increasingly important in the post-Moore era, is to augment system performance by building more intelligent compilers. One of our motivating hypotheses is that much of the capability needed to advance compiler optimization is already present: state-of-art compilers not only provide a large set of code transformations, but also (by-and-large) correctly apply them to preserve the semantics, syntax, and functionality of the code. The challenge lies in getting the compiler to select an appropriate sequence and number of these transformations so as to generate the highest possible performance code based on the developer goals of size, speed, or energy.

Modern compilers offer default optimization levels (e.g. -O_z, -O_3, and -O_{fast}) based on these goals. However the compiler transformations applied in these default levels are largely "one-size-fit-all" sequences that are often sub-optimal [15, 23, 252]: what works best for one application may not work for another. Several groups have tackled this problem using auto-tuning or iterative compilations targeting individual applications [74, 83, 110, 134, 176, 301]. But while these methods outperform the compiler's default settings, they are time-consuming and lack generality. A preferred solution is to have a single machine learning (ML) model that can accurately predict

bespoke optimal compiler transformations for any given application.

Several groups have attempted to predict compiler transformations using single ML models [22, 109, 302] but limitations remain. Models trained using supervised ML are overfitted to their training data and are therefore only applicable to a limited set of applications. Unsupervised ML techniques do not maximize performance. Also, in most cases the search space is restricted to some conservative sequence of transformations thus compromising accuracy [83, 169, 302].

Recently some groups have trained large language models (LLMs) on LLVM IR to predict compiler optimizations [95, 121]. However, such methods are constrained by the limited sequence length of inputs and thus place stringent boundaries on the code size. Moreover training LLMs require formidable amounts of data and resources [96]. Recent work like Coreset-NVP [169] has employed Graph Neural Networks (GNNs) to predict the optimal compiler passes. A major limitation of all these prior works is that, while they excel at learning patterns from static data, they lack the feedback loop for adaptive optimization. Reinforcement Learning (RL), on the other hand, is particularly well-suited for compiler optimization since the model learns from direct interaction with its environment and so can dynamically fathom complex interactions between transformation. RL may be the most promising approach for ML-enabled compilers [167].

RL-based approaches to training a single model for compiler optimization [133, 254, 300] have indeed shown considerable potential. However as [169] demonstrated, these methods have poor performance on unseen programs. [95] showed that models trained using RL had an overall net negative impact on code size when compared to -O_z.

We ask, can we overcome some limitations of prior work in training single RL model for compiler optimizations? In particular, can we reduce/eliminate the perfor-

mance regressions that are a bottleneck for compiler users in using ML models within compilers? To the best of our knowledge, our work is the first to propose a practical solution to performance degradations that come with any ML based model. We also present an approach to using ML models within state-of-art compilers using function level optimization at the middle-end of compiler pipeline. A trained model can make predictions in single shot without requiring compilation-intensive searches. An RL-based strategy with a feedback loop, presents an opportunity to keep evolving and improving with time. We demonstrate our approach by developing a neural network based pre-trained -O_{zmL} model for a production compiler such as *GCC*. To the best of our knowledge, the last significant effort in creating a single ML model for *GCC* was Milepost [109] and a long time ago. While *LLVM* is a popular choice in academia, *GCC* is the default choice for a large majority of significant software projects. Another major challenge has been feature extraction; we develop a *GIMPLE* parser to generate features for the training model.

In this chapter we first carry out a series of experiments changing different variables of the RL training phase to quantify performance of the model. We note that most functions cannot be improved over -O_z. Some only have a net negative change over the thousands of compiler flag combinations tested. We use these data to then train supervised models to classify the optimization potential of the functions. We find, however, that this method can only handle a limited subset of performance regressions. We then propose another method based on the capability of the pre-trained RL model to optimize functions. Both methods for designing classifier models are used to predict whether a test function will perform better with compiler default -O_z or using the pre-trained model's inference output -O_{zml}.

Our contributions are as follows:

- We propose a novel and practical function-level approach to training RL models for compiler optimization (rather than the standard of program-level method);

- We vary different variables during training and demonstrate a 46× performance benefit in code size reduction over previous efforts; and
- We propose and train classifier models in tandem with the pre-trained RL model to counter the performance regression limitation of ML-based compiler model. Our results show that it is possible to reduce performance degradations by 99% and yet achieve a 0.2% average performance improvement over -O_z. We also show how various "knobs" within the classifier's design can allow for different classification targets: it is possible to increase average performance improvement over -O_z to 1.88% but at the expense of 39% performance degradations.

8.2 Related Work

Optimization targets within state-of-art compilers can take several forms: (1) as analysis and transformative passes, (2) compiler flags that enable or disable compiler middle end optimizations, and (3) compiler parameters that control factors/compilation hyper-parameters. Prior research in compiler optimization has typically used the model as an auto-tuner to generate best heuristics for various goals: (1) phase ordering for *LLVM* compiler [24, 141, 176, 277]; (2) selecting optimal compilation flags typically for the *GCC* compiler [74, 83, 109, 110, 139, 170, 197, 301]; and (3) optimizing for specific parameters with respect to compiler passes. The latter include inferring appropriate vectorization and interleaving factors for the loops in the code [132], improving the register allocation [155], creating polyhedral models that optimize loops and find an optimal instruction schedule [7, 68, 159]; and MLGO [254] where ML is applied to the inlining pass. Amongst these, we look specifically at (2) to increase performance on *GCC* compiler's latest versions. Prior work along these lines is extremely limited and typically from many years ago [109]. Recent work [83, 301] has mainly exploited Bayesian optimization for flag tuning.

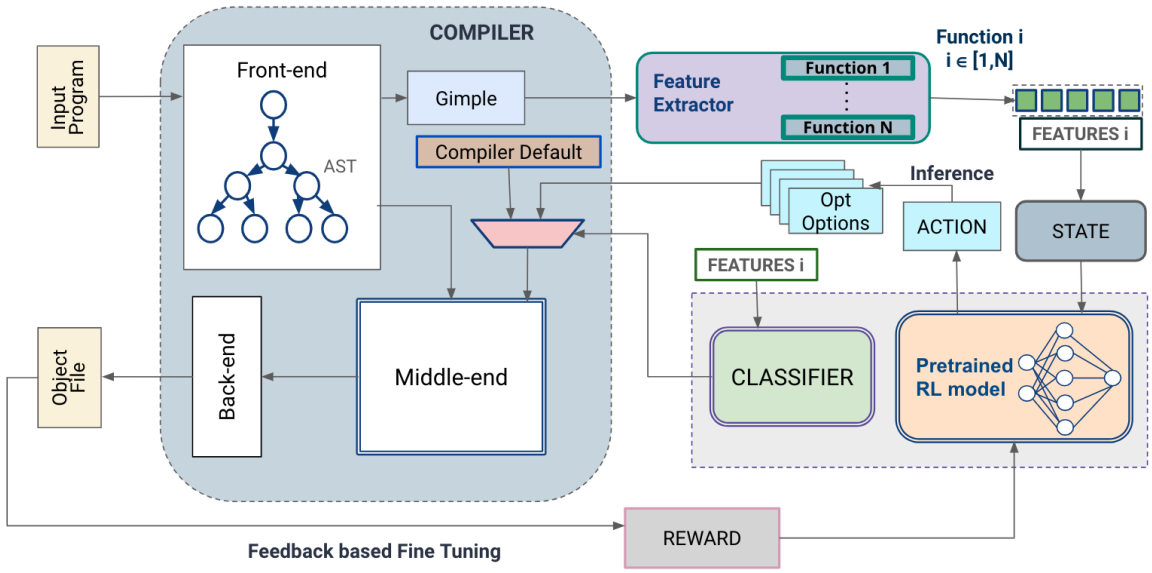

Figure 8·1: Proposed Workflow for the Deep Learning Compiler

We extend their work using RL combined with deep learning based approaches not for auto-tuning but in order to develop a best-effort, practical model that can be employed inside the *GCC* compiler to predict the optimization choices suited for the target architecture, application domain, and operating system. The only previous efforts along this line are by Autophase [133]. However they demonstrated over a limited training set of 100 applications and a small test set of 8 applications and for an HLS compiler. [169] reproduced the Autophase pipeline and concluded that RL based methods fail to generalize over unseen applications. We attempt to circumvent some limitations of Autophase and demonstrate how training variables such as better state features, reward computation, action space and possible actions allowed per episode of training, can lead to manifolds better performance in terms of average improvement over -O_z. We also make a broader contribution applicable to every work in ML enabled compiler model: reduction of performance degradations to enable practical inclusion of the model within production compilers.

RL-based frameworks are also common such as Supersonic [265] to choose and tune

a RL architecture for code optimization tasks and CompilerGym [94] that consolidates some compiler optimization frameworks into a platform to motivate research under the hood. Our work can be integrated within these frameworks and motivate future work along these lines.

8.3 Methodology

A major consideration of the deep learning compiler is that it should be able to do inference in a single compile. This is unlike prior work [95] where authors suggest an '-O_z Backup' extension: if model suggests pass list other than -O_z, the framework also runs -O_z and then selects the best of the two options. In practical compilers, we cannot compile twice; the overhead of this will diminish any gains that are to be made in the code size reduction. This also means that like any deep learning model our pre-trained model will also have performance degradations with respect to -O_z on some functions. We henceforth propose a classifier as a predictive model in tandem to the pre-trained model. It classifies function according to their optimization potential: if the classifier predicts the function will perform poorly on -O_{zmL}, it will propose its command line flag to be -O_z instead of -O_{zmL}.

8.3.1 The Workflow of the Deep Learning Compiler

Figure 8·1 shows the proposed framework for deep learning based compiler. The input program is fed into the front-end of the compiler. The front-end handles the syntactic and semantic analysis of the code, converting it into the abstract syntax tree (AST). This AST is then passed to the middle end of the compiler. The AST is also translated into *GIMPLE* , *GCC* 's intermediate representation of the original source code. The feature extractor is our *GIMPLE* parser that reads in the *GIMPLE* code and outputs function-level features for the input program. These include important properties of its functions such as information about basic blocks, instructions, loop characteristics,

control flow and data flow metrics etc. These features are cascaded in the form of a vector and fed as input to the pre-trained model. The pre-trained model uses this information to make predictions about the best compiler flags for this given function. The features are also fed as test labels for the pre-trained classifier. The classifier's predictive model decides whether the given function should be compiled with an -O_z, that is default optimization level for binary size reduction for compilers or -O_{zmL}, that is the inference output from our pre-trained RL model. The classifier output is passed as command line parameter to the middle end of the compiler. The back-end of the compiler generates the object code that is analyzed and its performance values are fed back into the pre-trained RL model to iteratively refine and evolve its policy.

8.3.2 Training the RL Model

Figure 8·2 shows the training loop. In the start we define benchmarks/C programs to use as our training data. These are fed into the compiler. The *GIMPLE* Representation of the input program is used to generate feature values for all functions within the input program. Moreover, other performance values such as per-function byte size at -O_z, per-function number of instructions at -O_z are recorded. Each function from the input program together with its feature values is used to train the RL Model.

Feature Engineering

Building an RL model requires efficient feature engineering: deriving quantifiable properties to characterize a given input program and then iteratively refining them to improve the accuracy of the model. The goal is to provide sufficient information to the model so it can reason and learn about program behavior. Prior works have used either used (i) feature based representation such as number of basic blocks, cyclomatic intensity etc [98, 109, 133, 191, 254] or (ii) representation learning using neural nets to represent input programs as embedding to the ML model [70, 93, 169, 260]. In

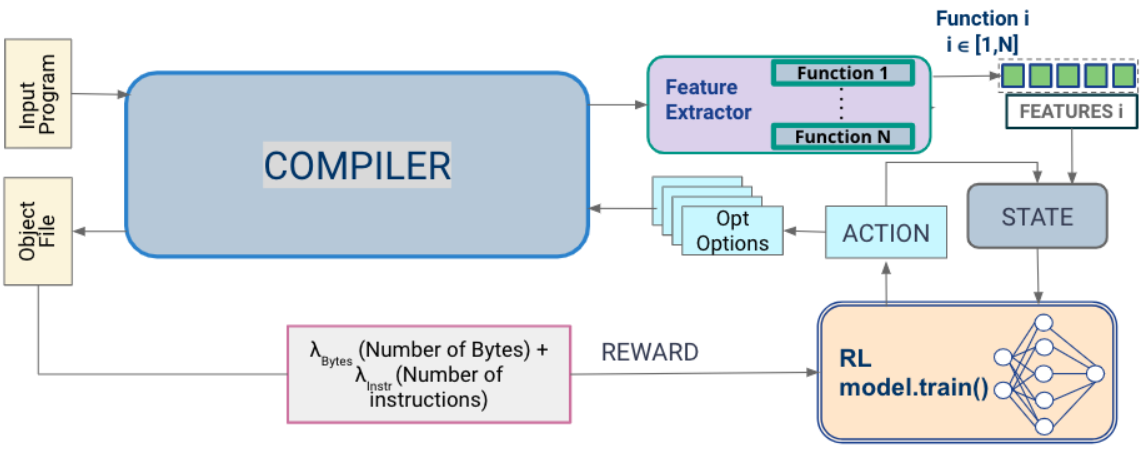

Figure 8·2: Training RL Model for Deep learning Compiler

this work we use static features along the lines of (i). These are derived from our Feature Extractor block and fed into the learning algorithm. We argue that no way of program representation is perfect. In addition, (ii) require training an initial model and the losses of this model simply add up to the overall losses. In using feature based representation, we try to stick to the roots by providing information to the model's policy network, that a CPU based compiler also utilizes in optimizing code. Hence the model roughly gets the same information as is available to heuristics within *GCC* .

Feature extraction for *GCC* is not straight-forward: the last major work to extract static features from *GCC* codes was Milepost [109] and their framework does not work beyond GCC 6+ [91]. We tested other alternatives such as a Python plugin by *GCC* developers [100]. However the limitation of most prior works is that they are not updated to recent *GCC* versions and hence are not upgraded for vectorization, inter-procedural optimizations, or new architectures. For LLVM, there are many efforts that allow feature extraction [94, 133], including a custom pass within *LLVM* [172].

We have therefore developed a *GIMPLE* (Intermediate Representation for *GCC*) parser that allows users to extract more than a 100 function-level static features.

Figure 8·3 shows the main blocks of our Feature Extractor workflow. In the first step, the *GCC* tree representation of functions is extracted. The advantage of using *GIMPLE* is that it is language-independent and its context is simple for optimization passes to operate on [183]. It is also relatively stable with the changes across compiler versions being minimal. *GCC* can dump out *GIMPLE* representation after every optimization pass it applies. We analyze after the *SSA*pass that is one of the first passes to be applied at the middle-end of the compiler and is a good representation of the program before it undergoes optimizations within the compiler middle end. The Feature Computation block contains a function to compute each feature from Figure8·3. These use the information from the output of the *GIMPLE* Analyzer and compute the relevant feature. The Feature Selector uses the target features selected by the user in the configuration file and outputs a vector of those feature values from a list of all available features.

Features extracted by prior state of art work are shown in color in Figure8·3. Green color code specifies features that have been considered by Autophase[133] and later used in many recent compiler optimization works [94, 169]. Purple shows features that Milepost [109] used and Cyan color code shows features that both Milepost and Autophase included in their subset to describe the code. Note some features have been omitted either because they are not relevant for *GCC* , or for latest versions, or are not appropriate for describing function-level attributes.

Features used are also highlighted in Figure8·3. These have been chosen carefully from the entire pool of options, based on training random forest regressor on training data and evaluating its performance on test data. The best accuracy of the regressor was shown on these subset of features. We have used the same set of features for both our RL model and also for the classifier models. Moreover, the features are normalized with respect to the number of instructions within that function. As [133]

noted, this normalization technique improves the performance of the model.

Action Space

This consists of optimization flags that engage with GCC's intermediate representation and modify it. GCC inherently applies a specific set of optimization flags when a code is compiled for size using -O_z. The premise is that this set of flags at -O_z does not guarantee best performance for every type of function. Hence a customized set of flags suited to the function needs to be applied. To generate the list of possible transformations that the model can apply we follow these steps:

1. We invoke GCC with -Q –help=optimizers to find out the exact set of optimizations that are enabled and disabled at -O_z.

2. We parse through this output and create list of optimization flags that contain flags currently not in -O_z by default but should be present. If a certain flag is [enabled] at -O_z we create its complementary disabled option by replacing -f/-g in its name with -fno/-gno. Similarly if a certain flag is [disabled] we create its complementary enabled state by replacing -fno with -f.

3. For each flags in the above mentioned list of optimization flags, we evaluate their impact on code size for a curated pool of 80 common functions. The results are given in Figure 8·4 and Figure 8·5. We further prune out flags that either do not impact the code size, or negatively impact it. This includes flags such as -falign-functions, -falign-jumps, -falign-labels, -falign-loops, -fallow-store-data-races, -fassociative-math, -fno-simd-cost-model, -fno-fp-contract, -fno-stack-reuse, -fno-ira-region, -fno-reorder-blocks-algorithm, -fno-ira-algorithm, -fno-stack-protector-all, -fno-stack-protector-explicit, -fno-stack-protector-strong, -fbranch-probabilities, -fcx-fortran-rules, -fcx-limited-range, -fdelete-dead-exceptions, -ffinite-math-only, -ffloat-store, -fharden-compares, -fharden-conditional-branches, -fopt-info, -fpack-struct, -fprofile-partial-training, -fprofile-reorder-functions, -freciprocal-math, -frounding-

math, -fshort-wchar, -fsingle-precision-constant, -fstack-clash-protection, -fstack-protector, -fstack-protector-all, -fstack-protector-explicit, -fstack-protector-strong, -ftrapv,-funreachable-traps, -funsafe-math-optimizations, -fvpt, -fwrapv, -fwrapv-pointer and -gstatement-frontiers.

We also remove flags from the list that are either not valid for C language or x86 architectures or are used for debugging such as -funroll-completely-grow-size, -fdelayed-branch, -fvar-tracking, -fvar-tracking-assignments, -fvar-tracking-assignments-toggle, -fvar-tracking-uninit and -fsave-optimization-record.

4. The remaining flags in the list of optimization flags after pruning process are used as the actions for our training workflow. These are given in Figure 8·6. In addition we pass an empty string as a valid option for action space. This enables the agent to learn **how many flags** it should apply for a given function.

Multi-task Reward

Code size has been used as the optimization criteria in a number of recent works for compiler optimization [95, 98, 121, 169, 178, 203, 212, 221, 232, 265]. This is partly because code size reduction, firstly, leads to cost reductions in terms of storage, bandwidth and memory, all critical for today's resource constrained applications [211]; and secondly is platform-independent, deterministic, and relatively noise-free, and hence an ideal target for compiler optimizations [94, 248, 254]. We however, optimize for both code size and number of instructions in this work. This is because (i) in many cases it is not possible to reduce binary sizes. For such there might be a decrease in number of instructions. Note however, a decrease in binary size can also lead to an increase in the number of instructions executed if those instructions require fewer bytes to encode. (ii) our experience has shown that RL training is more efficient with composite rewards likely because it provides intermediate objectives for the policy and hence a reward value for its incremental improvements. We use λ_{Bytes} and λ_{Instr}

to assign weights to the two goals. More weight is given to byte reduction(10:1 ratio). Since our workflow is automated we can easily tune for other goals such as runtime etc. or for other default levels such as -O_3. Byte size and number of instructions are evaluated from the object code. Our reward equation per episode of training is defined as the improvement with reference to -O_z:

$$\lambda_{\text{Bytes}} \frac{Bytes_{\text{Oz}} - Bytes}{Bytes_{\text{Oz}}} + \lambda_{\text{Num of Instr}} \frac{NumofInstr_{\text{Oz}} - NumofInstr}{NumofInstr_{\text{Oz}}}$$

Reinforcement Learning Policy

We use Proximal Policy Optimization(PPO) as our RL policy [219] and vary its hyper-parameters, activation function and size to enhance the training efficiency for compiler optimization. PPO is (by large), the state of art and considered best choice for optimization tasks [194]. It is also used within ChatGPT to provide feedback for policy enhancement [185]. PPO uses clipped surrogate objective to ensure the updates do not destabilize the training. Moreover, the batch updates can be synchronized easily across multiple workers(cores), each handling a separate function for optimization.

On the Use of Reinforcement Learning for Decision Making

Sequential Series of Decisions: Reinforcement Learning is typically used for sequential tasks: processes where future decisions depend on the action and state at a given step. Compiler optimization as defined in our work is sequential in nature. This is because: (i) We require a feedback loop for impacting the RL policy's future decisions. The actions(flags) applied at one step influence the future state(intermediate representation of the code) and reward(binary size) of the system.

(ii) We have a long term objective to maximize reward and are not optimizing an immediate decision or outcome. In each episode of training, the RL agent interacts

with a new function. Different functions require different flags to decrease their binary size. That is, there is no one-size-fit-all solution that works for all functions. Over the course of exploration and exploitation, the RL agent learns which flags work best for which function. That is it maps the states to the actions and rewards and hence learns a strategy. This is evident from the episode reward mean during training that steadily rises till it reaches its plateau.

(iii) Our environment is complex and unpredictable. The impact of actions(flags) on state and reward are not same across different functions. The cumulative effect of different flags may impact state and reward too. Hence a strategy is required that can learn and adapt to changes in environment.

RL is a promising choice for compiler optimization since it considers the consequences of earlier actions on future opportunities and can hence, adaptively learn the optimization strategy. Some might argue that since GCC flags are presumably independent hence it might be a non-sequential problem space and best optimized using other approaches. This is not true since GCC flags present a complicated action space that are not entirely independent of each other. Some flags turn on/off groups of GCC passes that alter the intermediate representation. Hence in such cases, the order of application of flags is important. Some flags inherently conflict and in such scenarios the order of flags matters. RL seems the most promising approach not only to adequately capture complex flag interactions but also allow the strategy to evolve over time as the policy encounters different functions, each requiring different set of flags to optimize for binary size.

Satisfying the Markovian Property: We have set up our framework as a Markovian Decision Process(MDP) defined by the State Space, Action Space, Reward (accumulated by the actions of the agent) and a Policy, (that is learned during the process to map state to a distribution of actions). A Markov property means that

the Policy's action depends on the current state and not on past states. PPO(the RL Policy used in our work) is not strictly Markovian: it does not necessarily base its decision on current state. However, we have addressed the Markov property by state augmentation. The histogram of flags together with static features adds historical context into the state. Intuitively meaning any current state of the environment captures the information of the past state: histogram of flags captures information about all flags that have been applied in prior states. To conclude, it is safe to say, our formulation of optimization problem is indeed Markovian in nature and fit for RL.

Training and Test Data

Table 8.1: Training and Test Data

Type	Benchmark	No of functions
Training Data	chstone	73
	mibench	454
	polybench	28
	stanford	63
Test Data	cbench	778
	convolution	26
	linear algebra	56

Table 8.1 gives the training and test data used in this work. While the rest are standard benchmarks widely used, convolution and linear algebra are a curated set of hand-picked programs that represent common patterns in code. For example convolution includes different types of filters such as laplacian, emboss, gaussian etc. Linear algebra includes different types of sorting algorithms, matrix operations etc. We realized such program patterns were lacking from the datasets. These programs are used for evaluating compiler optimization improvements.

Autotuner: To find the upper bound on performance for each of our functions in both training and test set, we use *autotuning*. Our autotuner is another PPO model

that trains on a single function. That is for every function within the training and test set we train its own PPO model and use it to output the best performance value that it encounters during its exploration and exploitation phase. Prior work have used random, greedy or heuristic based methods such as simulated annealing. Our choice of PPO is based on better results that it can achieve within similar compilation runs: PPO adapts itself to previous tuning results that it saw for a specific function within its certain episode. This adaptation allows it to improve its tuning strategy continuously instead of making isolated decisions like other methods. We ran PPO models for every function for approximately 10,000 episodes. The maximum possible improvement in binary size and instruction count possible for each benchmark is shown in Figure 8·7. However, such approach does have a drawback that it takes time. In total, it took us approximately 5 days on 20 cores to generate *autotuning* results(best binary size possible with different iterations of the flags) for all our training and test data.

In total, for the test data, the autotuner predicts a 7.15% improvement in binary size and a 4.45% improvement in number of instructions over -O_z. Our aim is to achieve some portion of the autotuner's performance using a single predictive model that surpasses compiler's default -O_z and does not require thousands of compiler runs.

Does a flag always cause byte reduction and should be included in GCC's -O_z? In order to answer this question, we run our autotuner on a curated pool of 80 functions (also analyzed in section 8.3.2) and store the list of best flags that give the maximum performance (defined as a sum of byte reduction and number of instruction reduction - as given in Section 8.3.2) We analyze how often a certain flag is present in the list of best flags. Figure 8·8 gives the result. We note that the flag *-fno-move-loop-invariants* is present almost 80% of the time in the set of best flags over the 80 functions analyzed. *-fno-tree-loop-im* is also present around

76% of the times. Other flags that are present more than 50% of the time in the set of best flags are *-fallow-store-data-races,-ffinite-math-only, -fwrapv, -fno-dce, -fno-devirtualize, -fno-inline-functions, -fno-ipa-pure-const, -fno-optimize-sibling-calls, -fno-reorder-blocks, -fno-rerun-cse-after-loop, -fno-sched-group-heuristic, -fno-shrink-wrap, -fno-signed-zeros, -fno-tree-ccp, -fno-tree-dce, -fno-tree-dominator-opts, -fno-tree-dse, -fno-tree-fre, -fno-tree-pta, -fno-tree-switch-conversion* Although there is a need for further investigation into what can be added/removed from the primitive -O_z list of flags, our results present an interesting case for further exploration.

8.3.3 Classifier

The primary purpose of classifier is to reduce performance regressors: functions whose performance degrades using -O_{zmL} when compared to -O_z. These performance degradations with respect to -O_z pose the biggest hindrance in the acceptance of any ML based model within modern compilers. With a classifier of reasonable accuracy, we present a stopgap solution for the embracing of any such model by the compiler users. As shown in figure 8·1, for every test function, the classifier decides whether it should be compiled with an -O_z or an -O_{zmL}. We propose multiple different approaches to training such classifiers and the performance benefit that can be achieved.

Method_1: Function Optimization Potential-Model Agnostic

While training our -O_{zmL} model (results in Section8.4.1), we extensively instrument the process analyzing how our model learns over the various training functions. In this process, we noted that for some functions, the model just could never learn anything better than *GCC*'s default -O_z. This observation is also evident from the autotuning results in Figure 8·7. For 206 functions in the training data and 227 functions in test data, the best reward can only be achieved at -O_z. For majority of other combinations of flags, the reward was always negative. Our goal is to classify all such functions

whose performance can only be less than or at best, equal to -O_z so they are not evaluated using -O_{zmL}. We postulate there is possibility for these functions to have performance degradation with respect to O_z during inference. Figure 8·9 gives the workflow. The training data is generated from the autotuner's results for the training set. In this case, the y_train is 1 for all functions except the 206 mentioned above. The x_train is the set of features highlighted in Figure 8·3 and also used for training the -O_{zmL} model. The classifier is evaluated by using it to predict optimization capability beyond -O_z for test functions. The classifier's accuracy is evaluated using true labels from the autotuning data for test functions. We tested different classifier types such as Decision Trees, Random Forest, XGBoost and Voting Classifier since each perform better for different goals. The results are discussed in Section 8.4.

Method_2: Trained -O_{zmL} Model's Capability to Optimize the Given Function

In this case, we ask: which functions can an RL model learn well to optimize? That is, once we have a trained -O_{zmL} model, we run inference on it using the same data it was trained on: its training functions. We notice that for the majority of the functions, the policy was able to learn pretty well. However, for 25 functions it predicted flags that resulted in performance worse than -O_z. We hence label these as 0 and remaining functions in training data as 1. The accuracy of the classifier is evaluated by running inference of the -O_{zmL} model on the test functions. The true labels are assigned by computing which functions the model could optimize over -O_z and which not. We evaluate this method of classification using two approaches:

Approach_1: Using Random Forest Classifier The workflow is shown in Figure 8·10. A random forest classifier is trained to classify which functions our pre-trained -O_{zmL} model will predict well. We define different goals and then tune hyper-parameters for the random forest classifier to maximize the assigned goal. Hyper-

parameters for random forest include number of trees, maximum depth of each tree, minimum samples in a leaf node, minimum samples required to split a node, maximum features for splitting nodes etc. For example if compiler user wants to reduce negative regressions, we use bayesian optimization for tuning hyper-parameters of the random forest model such that the negative regressions can be minimized.

Approach_2: Harnessing the Potential of Deep Learning We ask: can we outperform the capability of classical classifiers and achieve even better performance from our classification block? In this case we replace the classifier with deep neural net based RL-PPO model. We train the PPO model's cross entropy loss function for classifying which functions will perform better using -O_{zmL} and which should be optimized using -O_z. The workflow is given in figure 8·11. The PPO model is trained for multiple goals by altering its reward function accordingly.

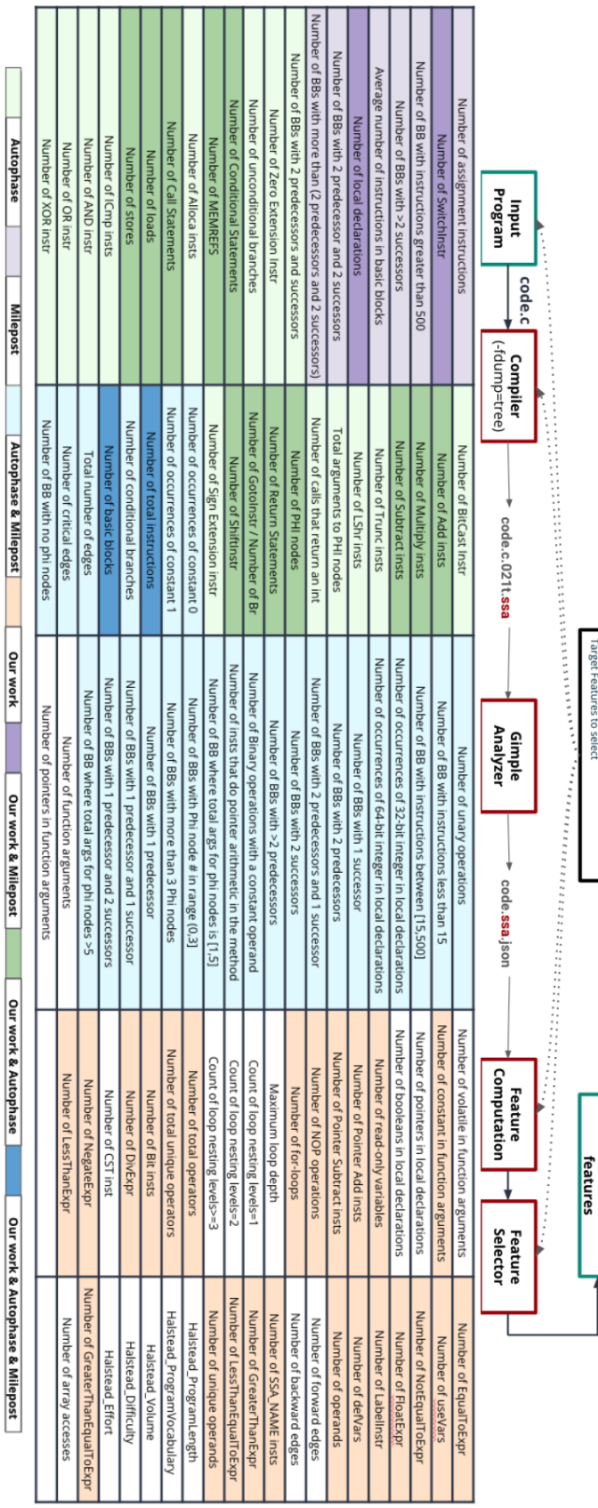

Figure 8·3: Feature Extractor Workflow and Function-level Features that can be extracted. The color codes show the features used in two state of art work in comparison to this work [109, 133]

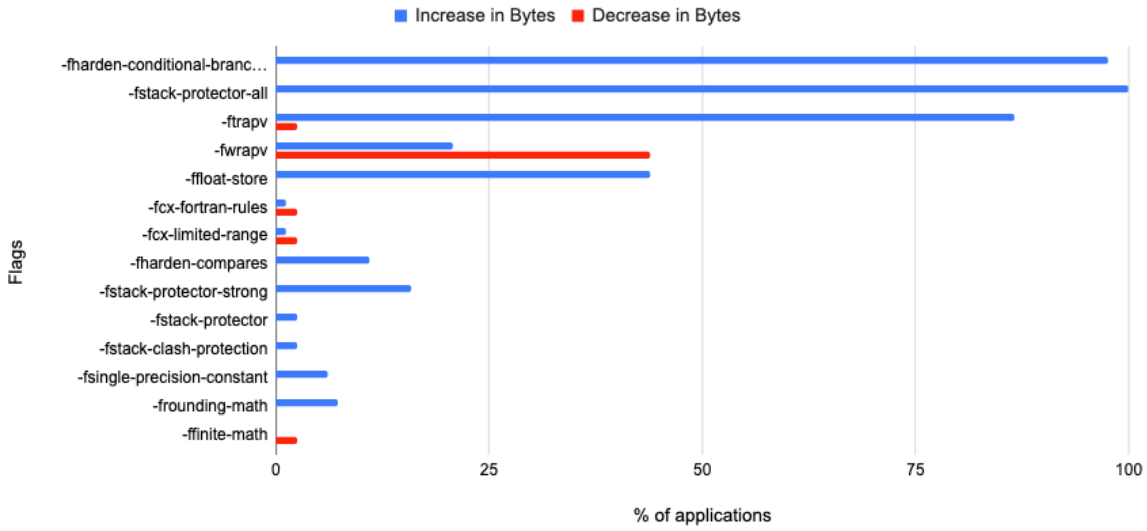

Figure 8·4: We test 80 functions and evaluate the impact of each flag on byte size of that function. The flags that caused a change to num of bytes are plotted together with percentage of total functions out of 80 that got impacted

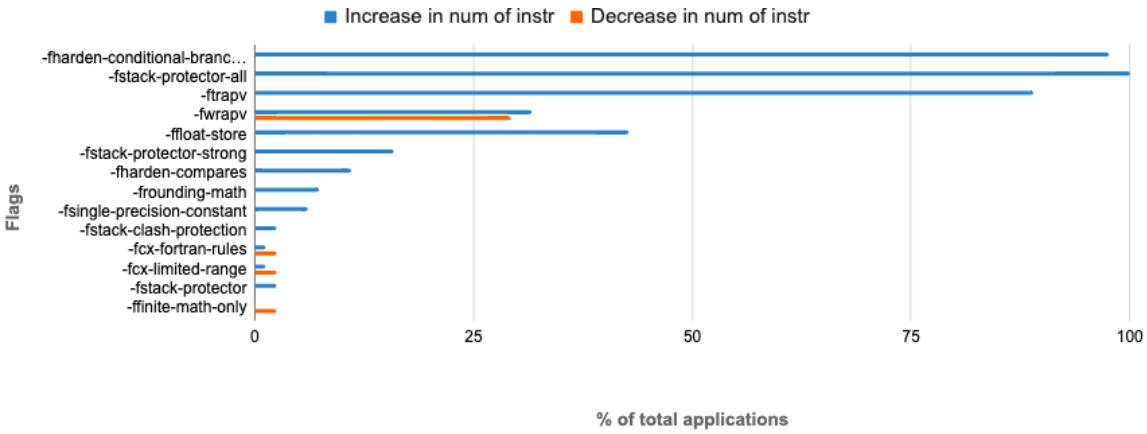

Figure 8·5: We test 80 functions and evaluate the impact of each flag on number of instructions in that function. The flags that caused a change to num of instructions are plotted together with percentage of total functions out of 80 that got impacted

-fassociative-math -fno-trapping-math -fno-signed-zeros, -fconserve-stack, -ffinite-loops, -fgcse-after-reload, -fgcse-las, -fgcse-sm, -fgraphite, -fgraphite-identity, -fipa-cp-clone, -fipa-pta, -fira-loop-pressure, -fisolate-erroneous-paths-attribute, -fkeep-gc-roots-live, -flimit-function-alignment, -flive-range-shrinkage, -floop-interchange, -floop-nest-optimize, -floop-parallelize-all, -floop-unroll-and-jam, -fmodulo-sched, -fmodulo-sched-allow-regmoves, -fnon-call-exceptions, -foptimize-strlen, -fpeel-loops, -fpredictive-commoning, -frename-registers, -freschedule-modulo-scheduled-loops, -fsched-pressure, -fsched-spec-load, -fsched-spec-load-dangerous, -fsched-stalled-insns, -fsched2-use-superblocks, -fschedule-insns, -fno-section-anchors, -fsel-sched-pipelining, -fsel-sched-pipelining-outer-loops, -fsel-sched-reschedule-pipelined, -fselective-scheduling, -fselective-scheduling2, -fsignaling-nans, -fsplit-loops, -fsplit-paths, -fsplit-wide-types-early, -ftracer, -ftree-cselim, -ftree-loop-distribution, -ftree-loop-vectorize, -ftree-lrs, -ftree-partial-pre, -ftree-slp-vectorize, -ftree-vectorize, -funconstrained-commons, -funroll-all-loops, -funroll-loops, -funswitch-loops, -fvariable-expansion-in-unroller, -fversion-loops-for-strides, -fweb, -fno-aggressive-loop-optimizations, -fno-allocation-dce, -fno-asynchronous-unwind-tables, -fno-auto-inc-dec, -fno-bit-tests, -fno-branch-count-reg, -fno-caller-saves, -fno-code-hoisting, -fno-combine-stack-adjustments, -fno-compare-elim, -fno-cprop-registers, -fno-crossjumping, -fno-cse-follow-jumps, -fno-dce, -fno-defer-pop, -fno-devirtualize, -fno-devirtualize-speculatively, -fno-dse, -fno-early-inlining, -fno-expensive-optimizations, -fno-forward-propagate, -fno-fp-int-builtin-inexact, -fno-function-cse, -fno-gcse, -fno-gcse-lm, -fno-guess-branch-probability, -fno-hoist-adjacent-loads, -fno-if-conversion, -fno-if-conversion2, -fno-indirect-inlining, -fno-inline, -fno-inline-atomics, -fno-inline-functions, -fno-inline-functions-called-once, -fno-inline-small-functions, -fno-ipa-bit-cp, -fno-ipa-cp, -fno-ipa-icf, -fno-ipa-icf-functions, -fno-ipa-icf-variables, -fno-ipa-modref, -fno-ipa-profile, -fno-ipa-pure-const, -fno-ipa-ra, -fno-ipa-reference, -fno-ipa-reference-addressable, -fno-ipa-sra, -fno-ipa-stack-alignment, -fno-ipa-strict-aliasing, -fno-ipa-vrp, -fno-ira-hoist-pressure, -fno-ira-share-save-slots, -fno-ira-share-spill-slots, -fno-isolate-erroneous-paths-dereference, -fno-ivopts, -fno-jump-tables, -fno-lra-remat, -fno-math-errno, -fno-move-loop-invariants, -fno-move-loop-stores, -fno-omit-frame-pointer, -fno-optimize-sibling-calls, -fno-partial-inlining, -fno-peephole, -fno-peephole2, -fno-plt, -fno-printf-return-value, -fno-ree, -fno-reg-struct-return, -fno-reorder-blocks, -fno-reorder-blocks-and-partition, -fno-reorder-functions, -fno-rerun-cse-after-loop, -fno-sched-critical-path-heuristic, -fno-sched-dep-count-heuristic, -fno-sched-group-heuristic, -fno-sched-interblock, -fno-sched-last-insn-heuristic, -fno-sched-rank-heuristic, -fno-sched-spec, -fno-sched-spec-insn-heuristic, -fno-sched-stalled-insns-dep, -fno-schedule-fusion, -fno-schedule-insns2, -fno-semantic-interposition, -fno-short-enums, -fno-shrink-wrap, -fno-shrink-wrap-separate, -fno-signed-zeros, -fno-split-ivs-in-unroller, -fno-split-wide-types, -fno-ssa-backprop, -fno-ssa-phiopt, -fno-stdarg-opt, -fno-store-merging, -fno-strict-aliasing, -fno-thread-jumps, -fno-toplevel-reorder, -fno-tree-bit-ccp, -fno-tree-builtin-call-dce, -fno-tree-ccp, -fno-tree-ch, -fno-tree-coalesce-vars, -fno-tree-copy-prop, -fno-tree-dce, -fno-tree-dominator-opts, -fno-tree-dse, -fno-tree-forwprop, -fno-tree-fre, -fno-tree-loop-distribute-patterns, -fno-tree-loop-im, -fno-tree-loop-ivcanon, -fno-tree-loop-optimize, -fno-tree-phiprop, -fno-tree-pre, -fno-tree-pta, -fno-tree-reassoc, -fno-tree-scev-cprop, -fno-tree-sink, -fno-tree-slsr, -fno-tree-sra, -fno-tree-switch-conversion, -fno-tree-tail-merge, -fno-tree-ter, -fno-tree-vrp, -fno-unwind-tables, -fno-exceptions, -fno-vect-cost-model

Figure 8·6: Optimization Flags Used

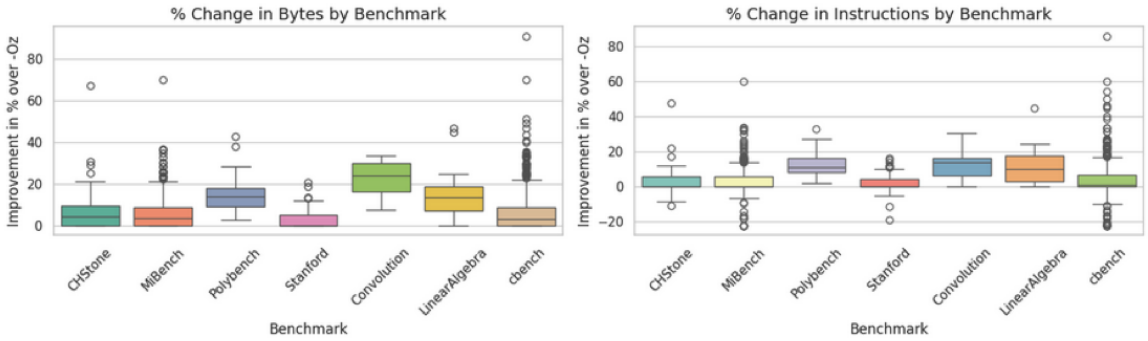

Figure 8·7: Autotuning Results: Max % Improvement over -$O_z$in Binary Size and Instruction Count for each Benchmark

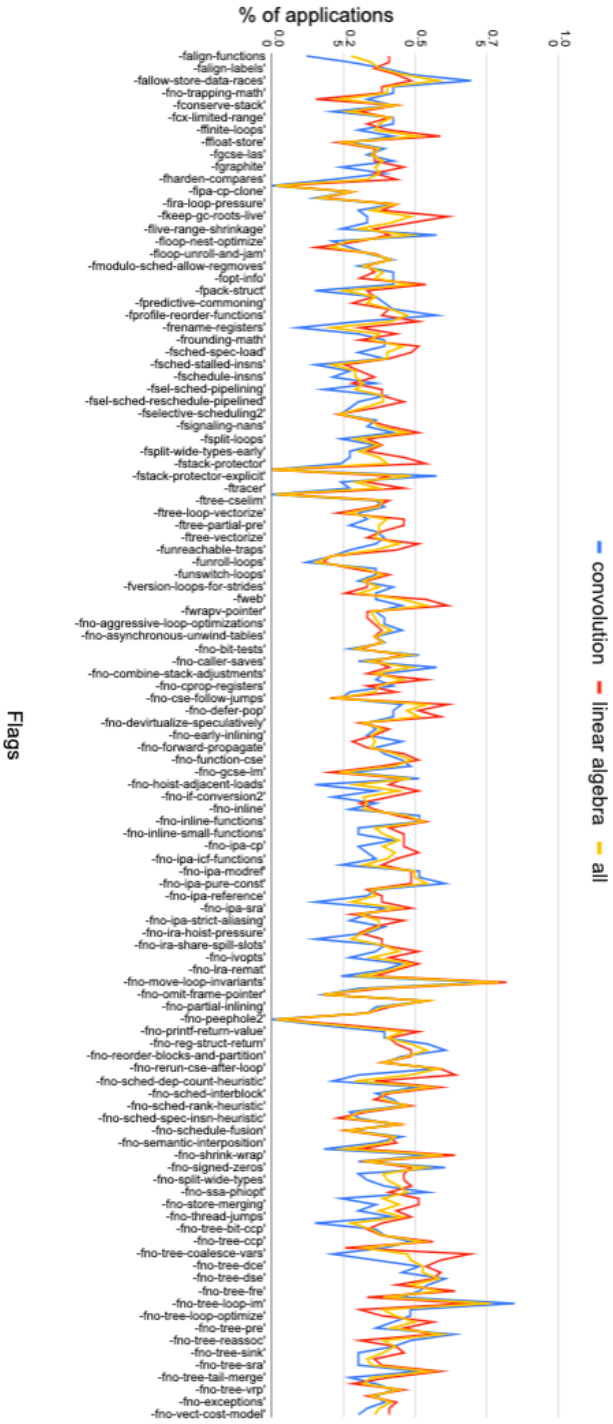

Figure 8·8: We run our PPO-based autotuner on 80 functions and evaluate which are the best list of flags for each function(that give best performance in terms of byte size and number of instructions). This graph shows the percentage of applications that have a flag 'X' in its best set of flags

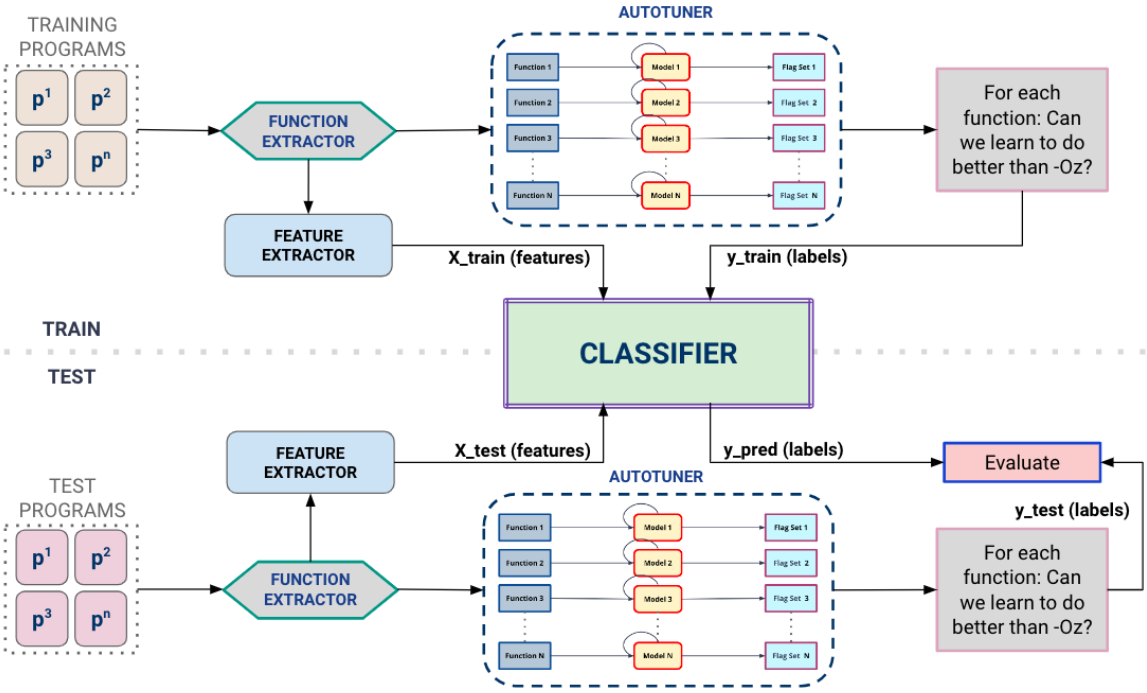

Figure 8·9: Classifier based on Programs' Optimization Potential

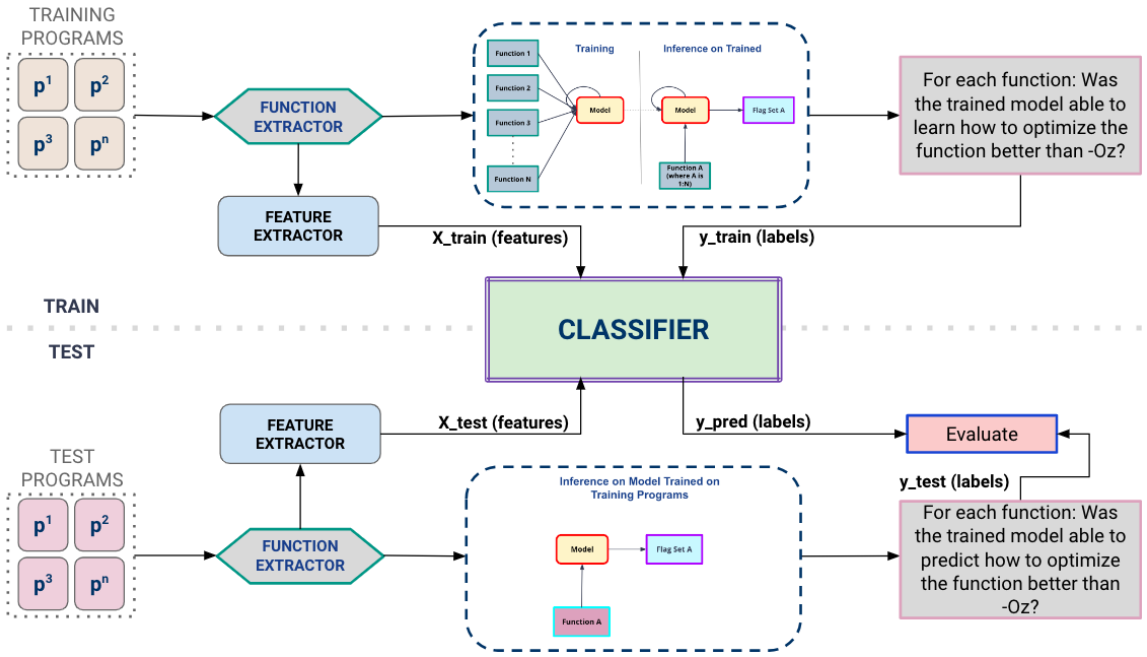

Figure 8·10: Method_2 - Approach 1

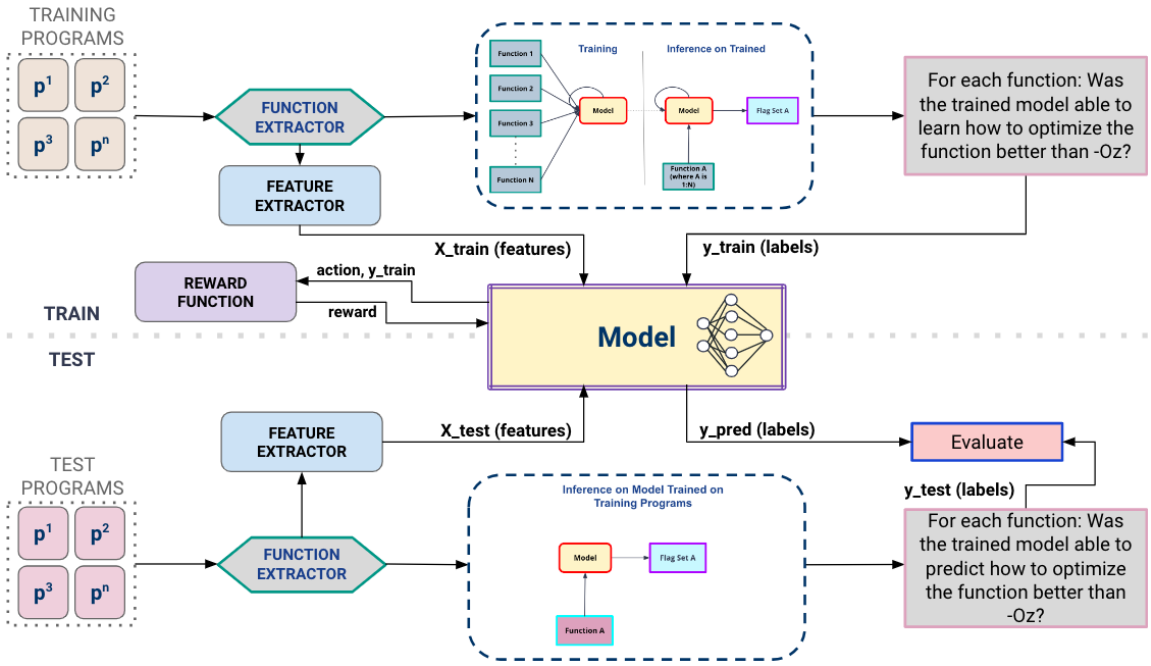

Figure 8·11: Method_2 - Approach 2

8.4 Evaluation

In this section we evaluate (i) the ability of our pre-trained -O_{zmL} model in generating flag list for improved performance over -O_z and variation in its performance by changing training variables (ii) comparison with the state of art (iii) the performance of the different classifiers and their impact on the overall efficiency of the workflow.

All training, evaluation and data collection are performed on a single multi-core CPU node. The CPU is a 16-core AMD Ryzen Threadripper PRO 5955WX, with 196GB of RAM. *GCC* compiler version 13.2.1 is used. For every compilation run, the -O_z flag is always applied; only the sequences of flags given in Figure 8·6 are varied.

Table 8.2: Different Training Versions Explored in this Work

Versions	Action Space	State	Reward
v0	$A = \{a_1, a_2, a_3, \ldots, a_{N+1}\}$	action history + normalized features	$\lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}} + \lambda_{\text{Flags}}$ (No. of $-fno$ flags − No. of $-f$ flags)
v1	$A = \{a_1, a_2, a_3, \ldots, a_{N+1}\}$	action history + normalized features	$\lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}}$
v2	$A = \{a_1, a_2, a_3, \ldots, a_{M+1}\}$ **where M includes only -f options**	action history + normalized features	$\lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}}$
v3	$A = \{(a_1, a_2, a_3, \ldots, a_N) \mid a_i \in \{0, 1\}\}$	action history + normalized features	$\lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}}$
v4	$A = \{a_1, a_2, a_3, \ldots, a_N\}$	action history + normalized features **these marked as Autophase in Fig 3**	$\lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}}$
v5	$A = \{(a_1, a_2, a_3, \ldots, a_N) \mid a_i \in \{0, 1\}\}$	action history + normalized features	$Reward = \begin{cases} 1, & \text{if } \lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}} > 0 \\ -1, & \text{if } \lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}} < 0 \\ 0, & \text{if } \lambda_{\text{Bytes}} \frac{\text{Bytes}_{Oz} - \text{Bytes}}{\text{Bytes}_{Oz}} + \lambda_{\text{Instr}} \frac{\text{Instr}_{Oz} - \text{Instr}}{\text{Instr}_{Oz}} = 0 \end{cases}$

8.4.1 Training Results

Table 8.2 gives the different training versions that were explored in this work. Each version is trained using the training datasets and evaluated after 3000 iterations of training on the test datasets given in Table 8.1. Figure 8·12 gives the inference results in terms of improvement over -O_z averaged over all test functions. Figure Section8·13 gives the performance of test functions as a percentage of the total. In the default case, the RL model applies one flag at a time from the action space as discussed in Section8.3.2. Since we also have an *N+1* action as empty string (do not apply any flag), the model learns to minimize the set of flags it applies for a given function. The default "state" is the list of features we explored as discussed in Section8.3.2 normalized over the number of instructions and concatenated with action history.

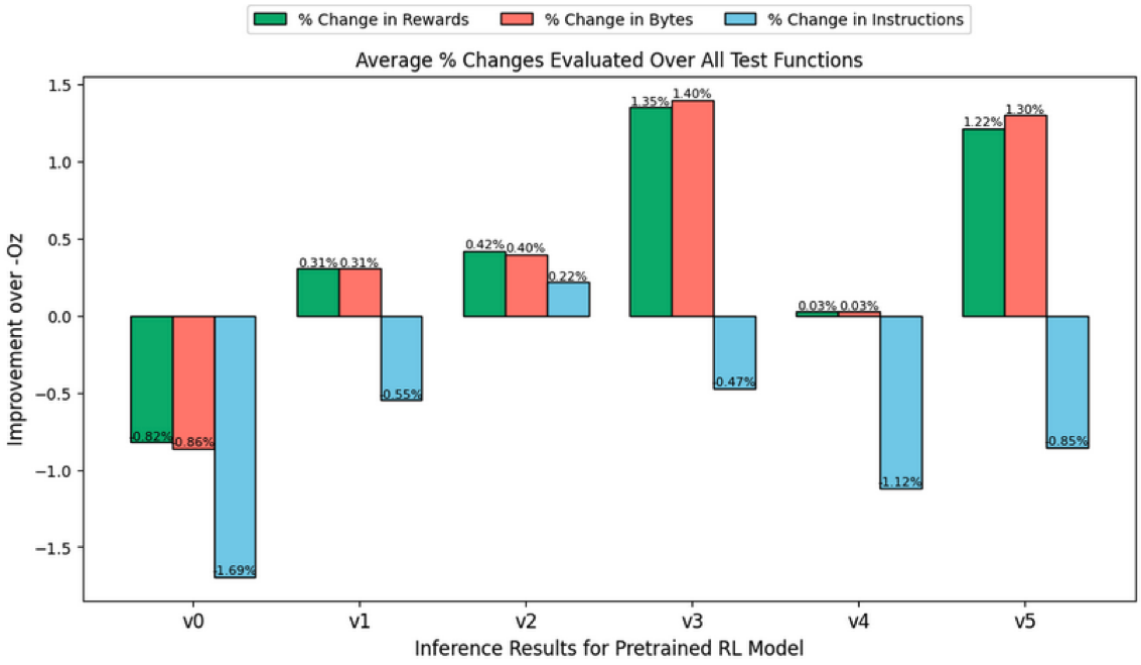

Figure 8·12: Average % Improvement over -O_z for Models Trained using Training Versions given in Table 8.2

The default reward is the sum of gains from reduction in bytes and reduction in number of instructions as given in Section8.3.2.

For each version v0-v5, we varied the state, reward and action space to evaluate the impact on training and inference results. In v0, we include a λ_{Flags} to encourage the policy to learn so as to increase the number of -fno flags (flags that turn off certain passes from the compiler default set of 256 passes in -O_z) and decrease number of -f flags(that generally turn on additional passes). This gives the largest percentage of functions improved but the net improvement summed over all test functions is negative.

In v1, we try to circumvent this by decreasing the complexity of the reward equation and only making the policy learn to optimize bytes size. This leads to a net improvement over -O_z.

In v2 we prune our action space to include only -f options. Our assumption is

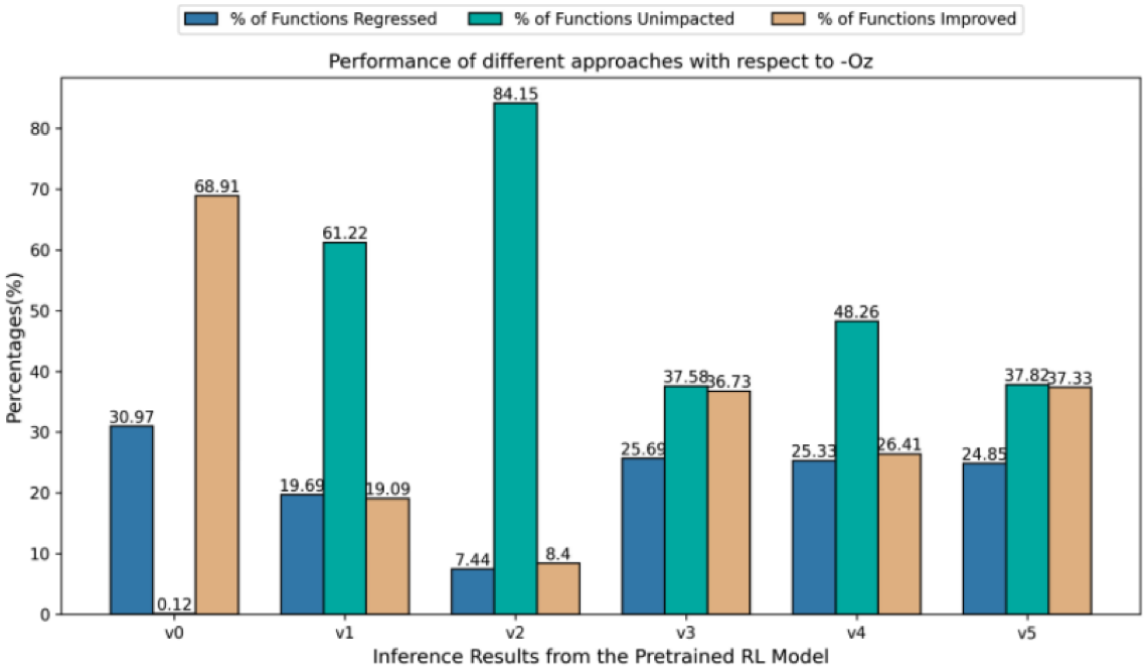

Figure 8·13: Functions Regressed, Improved and Unchanged for Models Trained using Training Versions given in Table 8.2

that -fno flags might turn off certain optimizations that *GCC* requires to decrease the byte size of functions. Hence by restricting action space to only enable additional optimizations without disabling anything from -O_z. This gives an even greater improvement over -O_z and also in reducing the number of instructions overall. This also decreases the regressions, maximizing the number of functions un-impacted.

In v3, we restructure our action space as a set of independent binary actions (i.e. a multi-binary action space) where each flag can either be applied [1] or not [0]. This way we enable PPO to learn whether to apply each flag independently. This gives the maximum benefit in terms of overall improvements. The functions that are improved have an average 8.45% improvement over -O_z.

In v4, we try to replicate the environment proposed in state of art work, Autophase [133]. We discuss it in detail in subsection 8.4.2.

In v5, we ask: Can we combine the classification within PPO model? We set up

our environment using the multi-binary action space and use the same default state. We calculate reward as given by the default reward equation. However, before passing the reward value to the policy we discretize it by converting it into -1,0 or 1. Our assumption is that categorical values of reward instead of continuous, might enhance PPO's learning. The -1 reward refers to functions for which performance regressed over -O_z. However, in this case we did not reap any additional benefit in terms of negative regressions.

Table 8.3: Comparison of Results For Method_2: RL based Classifier(Deep Neural net) and Random Forest(RF) Classifier

	Measurements									
	neg (Functions Regressed)		pos (Functions Improved)		avg_reward		avg_pos_reward		avg_neg_reward	
	RL based Classifier	RF	RL based Classifier	RF	RL based Classifier	RF	RL based Classifier	RF	RL based Classifier	RF
without classifier	214		306		1.35		8.45		-6.18	
ideal classifier	0		306		8.45		8.45		0	
neg (Functions)	3	58	14	151	0.2	1.09	10.87	8.91	-1.91	-7.53
pos Functions	156	143	246	234	1.35	1.32	7.45	8.48	-4.55	-6.18
avg_reward	146	84	245	207	1.66	1.88	8.65	9.36	-5.04	-4.42
avg_pos_reward	48	87	74	169	0.48	1.51	12.25	10.82	-10.6	-6.57
avg_neg_reward	83	85	126	184	0.56	1.4	5.86	8.08	-3.25	-3.81

8.4.2 Comparison to State-of-Art

To the best of our knowledge, Autophase [133] is the only other work that have attempted to create a generalized, RL-based predictive model for compiler optimizations. Autophase demonstrated on a limited training set of 100 randomly generated programs, the possibility of generalization across a wide range of applications. Our work is different in that (i) we target CPU based compilations and not high level synthesis as was originally proposed by Autophase; (ii) we target binary size reduction and not execution time speedup; (iii) we target function level optimization rather than application level; (iv) we use a much greater set of training and test applications; (v) we are the first to demonstrate applicability of this for GCC. All other work was demonstrated on LLVM.

Some recent work [95, 169] used LLM and GNN to create a generalized model for compiler passes and replicated Autophase's environment on their benchmarks to illustrate improvements. We also follow along their lines and replicate Autophase's

environment for comparison. In v4, we defined state using the features set also used in that work. The reward is set to binary size reduction and action space is fixed to a constant equal to the size of possible actions. The results show that it had a marginal improvement over -O_z. From all our results we conclude that v3 overall gives the best performance improvement and a more than 45 times improvement achievable by prior state of art work (v4-Autophase). In the subsequent sections we further improve v3 by reducing negative regressions using the proposed Classifiers.

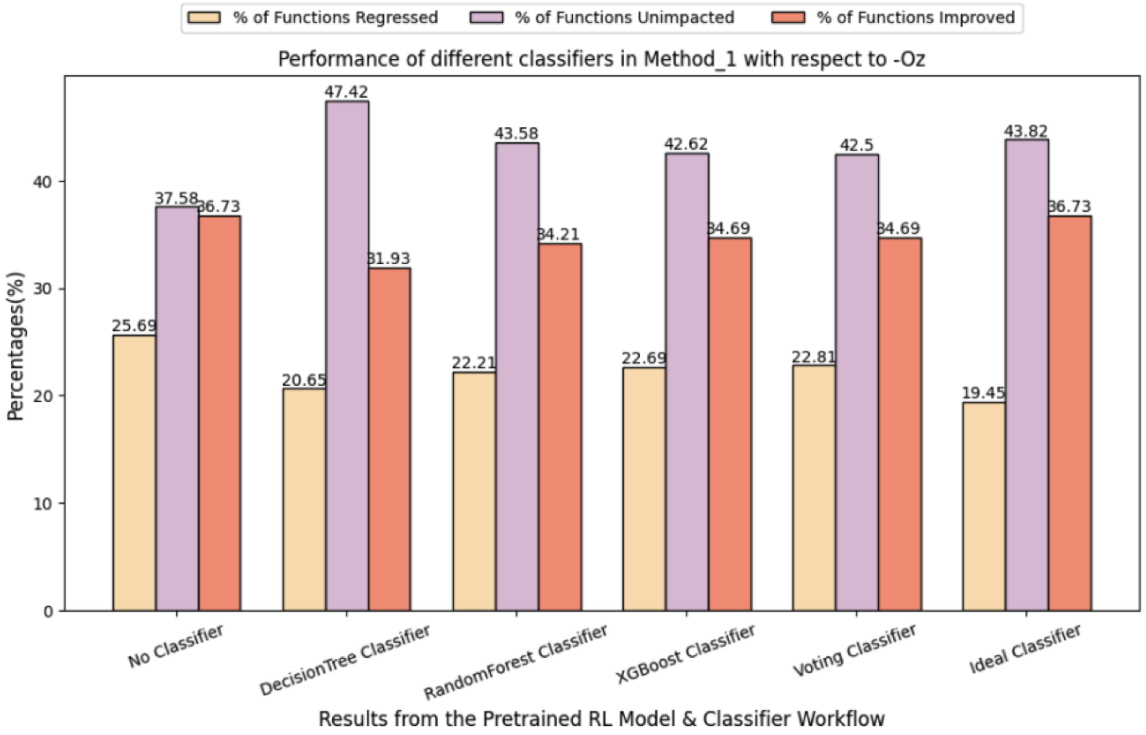

Figure 8·14: Comparison of different classifiers in Method_1

8.4.3 Evaluating Performance of Classifier Method_1

Figure 8·14 shows the classification results (post-inference) when Method_1 is evaluated using different types of classifiers. Green color highlights the best performance that can be achieved for each goal. For example if we want to remove negative regressions using method 1 (i.e. functions that we predict will perform bad), we should

use decision tree classifiers. Similarly if we want to increase overall average reward the Voting Classifier is a better choice.

8.4.4 Evaluating Performance of Classifier Method_2

Table 8.3 shows the classifier performance when evaluated using Random Forest (Section 8.3.3) and deep neural net as classifier (Section8.3.3). Green color highlights what best performance can be achieved for each goal. Rows represent compiler user goal for example decreasing negative regressions, increasing positive improvements, increasing average reward improvement over compiler defaults, increasing the average positive reward across functions with positive improvements and decreasing the average negative reward across functions with performance degradations. The results are very promising. If a compiler user wants to ensure that inference using -O_{zmL} will not give them performance below *GCC*'s -O_z, then it is possible to do that with more than 99.6% certainty using an RL based classifier in tandem with a pre-trained RL based -O_{zmL} model. However, as Table 8.3 shows, the average reward in terms of reduction in bytes and number of instruction comes down to 0.2%. We can choose to train classifier model with high penalty such that true positives are reduced. If we want to ensure high average improvement over -O_z, we can train a Random Forest Classifier to increase average reward to 1.88%.

Figure 8·15 summarizes the best results in the form of a confusion matrix. If our goal is to reduce negative regressions, the results are shown in (i). In this case we minimize false positives bringing them down to 3. This means that out of 214 functions that would have given performance degradations, we have reduced them by 99.6% and are down to just 3. However we get a high proportion of false negatives. This just means that these 296 functions are pooled under -O_z instead of -O_{zmL} where they would have performed better. If we want to increase true negatives (that is possibility of a function getting evaluated by -O_{zmL} for an improved performance),

we get case (ii). If we want to increase the average reward summed over all test functions, we get (iii). Here at the cost of 84 function degradations, we get an average 1.88% reward improvement over -O_z. If we want to increase the average reward over functions that are better optimized using -O_{zmL}, we use case (iV). Here at the cost of 48 performance degradations we get a net 12.25% average improvement over -O_z for the pool of 74 functions.

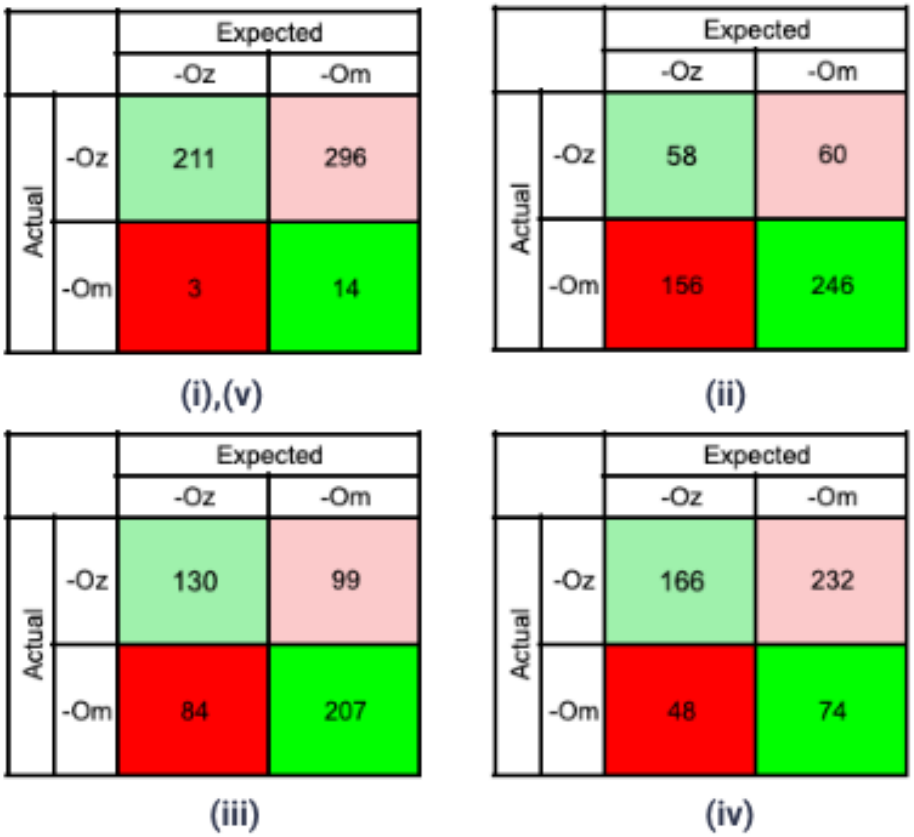

Figure 8·15: Best results for different goals: (i)Reducing Negative Regressions by 99%, (ii)Increasing Functions Improved (iii)Increasing Average Reward (iv) Increasing Average Positive Reward for Functions Improved (v) Decreasing Average Negative Reward for Functions Regressed

8.5 Conclusion

In this chapter we have presented a practical approach to widespread inclusion of an -O_{mL} option within modern compilers. A major hindrance in all deep learning compilers is the unavoidable performance degradations over compiler defaults for a subset of functions. We have proposed classifier models to circumvent and reduce this. However, our approach is just an initial solution. For future work, we propose inclusion of the classification within the pre-trained model through powerful models invented to suit the nature of programs: a limitation at present.

Chapter 9

Conclusion

This thesis makes significant contributions:

- **Research Thrust 1:** We learn to modify source code for performance. This is covered in Chapters 4 and 5
- **Research Thrust 2:** We learn a better compiler strategy to optimize a source code. This is covered in Chapter 6
- We adequately represent code as fixed size vector of features. This is covered in Chapter 7
- **Research Thrust 3:** We reduce the training time of ML models by replacing performance computation from Compiler with a neural net Cost Model. This is a practical approach to estimating binary size of kernels without compiling. This is covered in Chapter 7
- **Research Thrust 4:** We pre-learn compiler strategy using a deep learning model such that it performs better than compiler default in terms of predicting compiler heuristics. It is sufficiently practical to be integrated into compiler as an -OmL option. This is covered in Chapter 8.

Figure 9·1 sketches at a very high level how this thesis contributes to the big picture relating to the Performance, Portability and Programmability dilemma as discussed in Chapter 2. Each research thrust is automated - the input is HLL code in

C. This way portability remains unaltered. Programmability is assumed unchanged since developer does not alter the source code. The framework for each research thrust takes the C code and outputs the intended solution. Performance is improved without changing the other two Ps. The PPP plane is raised along the Z axis for each research thrust as labeled in the figure.

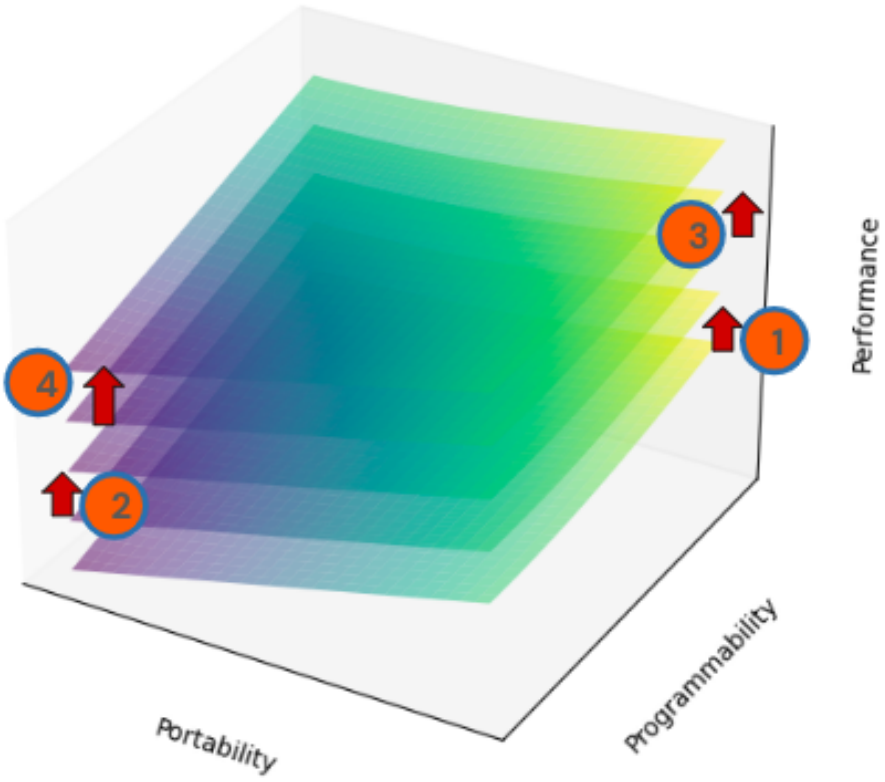

Figure 9·1: State of PPP: How each research thrust contributes to the PPP problem

Despite all the progress made in decades of research and development, compilers still face many unresolved challenges. The introduction of new and evolution of existing hardware architectures, innovation in semiconductors, and ever-changing algorithms and workloads demand complex yet more “intelligent" compilers. This also gives rise to other challenges such as security and complexity of software. While our work is a step in this direction, we strongly believe that the future will behold more artificial intelligence infused into compilers and AI based models predicting decisions

for compilers that are faster and accurate. It will not be surprising if AI replaces every heuristic within a compiler with a learned model, thereby minimizing all human interaction. This will nevertheless require a coordinated effort. For the compiler developers, it will be necessary to acquire a deeper understanding of the behavior of the optimizations, their interactions with each other, and their targeted impact on source codes. Developing better and improvised optimization options will also be required to meet the demands of the AI landscape in compilers. It will also be necessary to expose more options within a compiler's flow to enable optimizations at a fine-grained level.

For machine learning experts, it will be necessary to develop model architectures specifically suited for the compilers. It will also be necessary to represent source codes in forms that models can better decipher. At present, multiple program representations are available, yet none is complete in describing a given program. A more comprehensive feature representation will enable better training and inference from the model.

We discuss future directions for each research thrust in its relevant chapter. Here we present briefly our broader vision of what the future in this field holds. We believe that our work has helped opened doors to make both legacy and future workloads be efficient and performant, while at the same time being fully automated. All research thrusts presented here are automated, end-to-end frameworks, hence enhancing accessibility to naive and advanced users alike. A next step along the lines of this dissertation will involve incorporating fine-tuning within the reinforcement learning model; training the pre-trained model on a specific dataset, to allow the model to enhance its learning for a specific set of applications. While AI allows obvious improvements to optimizations than those a human developer would make, it can still make errors. We presented one approach in tackling these errors (or performance

degradations) by classification of functions/applications. We believe this can be further improved by incorporating a deeper understanding of applications and how they are perceived by the compiler and then aiding the AI model in making its decisions. We can also employ a multi-agent approach whereby a different agent is trained to predict heuristics for a class of applications. This coupled with an efficient classifier for application grouping can become a valuable next step.

Bibliography

[1] Enzian. http://enzian.systems/ [Last accessed: April 29, 2021], 2021.

[2] IBM SuperVessel, OpenPower Cloud. https://www.research.ibm.com/labs/china/supervessel.html [Last accessed: April 29, 2021], 2021.

[3] AWS. EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/ [Last accessed: April 29, 2021], 2021.

[4] A. Abel and J. Reineke. uica: Accurate throughput prediction of basic blocks on recent intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–14, 2022.

[5] A. Abel, S. Sharma, and J. Reineke. Facile: Fast, accurate, and interpretable basic-block throughput prediction. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 87–99. IEEE, 2023.

[6] F. Abel, J. Weerasinghe, C. Hagleitner, B. Weiss, and S. Paredes. An FPGA Platform for Hyperscalers. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 29–32. IEEE, 2017.

[7] A. Brauckmann, A. Goens, and J. Castrillon. PolyGym: Polyhedral Optimizations as an Environment for Reinforcement Learning. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 17–29. IEEE, 2021.

[8] Acronix. Speedster 7t FPGAs. https://www.achronix.com/product/speedster7t-fpgas [Last accessed: April 29, 2021], 2021.

[9] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38 (4):1–12, 2019.

[10] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using Machine Learning to focus Iterative Optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11–pp. IEEE, 2006.

[11] N. B. Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo. An MLIR-based compiler flow for system-level design and hardware acceleration. In *International Conference on Computer-Aided Design*, pages 1–9, 2022.

[12] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986. ISBN 0201100886.

[13] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[14] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 39(7):231–239, 2004.

[15] M. Almakki, A. Izzeldin, Q. Huang, A. H. Ali, and C. Cummins. Autophase v2: Towards function level phase ordering optimization. In *Machine Learning for Computer Architecture and Systems 2022*, 2022.

[16] G. Alonso, T. Roscoe, D. Cock, M. Owaida, K. Kara, D. Korolija, Z. Wang, et al. Tackling hardware/software co-design from a database perspective. In *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, January 2020.*, 2020.

[17] W. Altoyan and J. J. Alonso. Investigating performance losses in high-level synthesis for stencil computations. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203. IEEE, 2020.

[18] AMD Xilinx. Vitis High Level Synthesis User Guide. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS` [Last accessed: May 10, 2022], 2022.

[19] P. Amiri, A. Pérard-Gayot, R. Membarth, P. Slusallek, R. Leißa, and S. Hack. FLOWER: a comprehensive dataflow compiler for high-level synthesis. In *Int. Conf. on Field-Programmable Technology*, 2021. doi: 10.1109/ICFPT52863.2021.9609930.

[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[21] H. Arabnejad, J. Bispo, J. M. Cardoso, and J. G. Barbosa. Source-to-Source Compilation Targeting OpenMP-based Automatic Parallelization of C Applications. *J. of Supercomputing*, 76:6753–6785, 2020.

[22] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–28, 2017.

[23] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[24] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano. The Phase-Ordering Problem: A Complete Sequence Prediction Approach. In *Automatic Tuning of Compilers Using Machine Learning*, pages 85–113. Springer, 2018.

[25] C. Aubert, T. Rubiano, N. Rusch, and T. Seiller. A novel loop fission technique inspired by implicit computational complexity. *arXiv preprint arXiv:2206.08760*, 2022.

[26] Alibaba. Alibaba FPGA Cloud. `https://www.alibabacloud.com/help/doc-detail/108504.htm` [Last accessed: April 29, 2021], 2021.

[27] Baidu. FPGA Cloud. `https://cloud.baidu.com/product/fpga.html` [Last accessed: April 29, 2021], 2021.

[28] Bittware. Computational Storage. `https://www.bittware.com/fpga/storage/` [Last accessed: April 29, 2021], 2021.

[29] Blocks&Files. A brief look at AWS Redshift's AQUA acceleration hardware. `https://blocksandfiles.com/2019/12/05/amazon-aqua-data-warehouse-acceleration-hardware/` [Last accessed: April 29, 2021], 2021.

[30] Blocks&Files. Our Computational Storage Drives are Bigger, Faster and Cheaper than Ordinary SSDs. `https://blocksandfiles.com/2021/02/22/scaleflux-ceo-hao-zhong-interview/` [Last accessed: April 29, 2021], 2021.

[31] Cisco. Nexus 3000 Switch Architecture. `https://www.ciscolive.com/c/dam/r/ciscolive/us/docs/2018/pdf/BRKDCN-3734.pdf` [Last accessed: April 29, 2021], 2021.

[32] Huawei. CloudEngine Is the Foundation of the Intent-driven Network. `https://actfornet.com/ueditor/php/upload/file/20191206/1575567949680852.pdf` [Last accessed: April 29, 2021], 2021.

[33] Intel. SoC FPGAs. https://www.intel.com/content/www/us/en/products/programmable/soc.html [Last accessed: April 29, 2021], 2021.

[34] Nimbix. FPGA Cloud. https://www.nimbix.net/what-is-an-fpga [Last accessed: April 29, 2021], 2021.

[35] OVH. Acceleration-as-a-Service leveraging Intel PAC. https://www.ovh.com/world/news/press/cp2541.ovh_launches_acceleration-as-a-service_leveraging_the_new_intel_programmable_acceleration_card_and_app_store_from_fpga_acceleration_partner_accelize [Last accessed: April 29, 2021], 2021.

[36] Scaleflux. Computational Storage. https://www.scaleflux.com/ [Last accessed: April 29, 2021], 2021.

[37] Tencent. FPGA Cloud Server. https://cloud.tencent.com/product/fpga [Last accessed: April 29, 2021], 2021.

[38] TheNextPlatform. Computational Storage Winds its Way Towards the Mainstream. https://www.nextplatform.com/2020/02/25/computational-storage-winds-its-way-towards-the-mainstream/ [Last accessed: April 29, 2021], 2021.

[39] Xilinx. SmartSSD Computational Storage Drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html [Last accessed: April 29, 2021], 2021.

[40] Xilinx. SoCs with Hardware and Software Programmability. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html [Last accessed: April 29, 2021], 2021.

[41] Baidu. Baidu ABC Platform. https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/Baidu_ABC_Platform.pdf [Last accessed: April 29, 2021], 2021.

[42] Bittware. How OVHcloud Uses FPGAs to Mitigate DDoS Attacks. https://www.bittware.com/resources/case-study-ovh/ [Last accessed: April 29, 2021], 2021.

[43] Cisco. About the Nexus 3550-T Triton. https://exablaze.com/docs/3550t/about/ [Last accessed: April 29, 2021], 2021.

[44] FaBRIC. FPGA Accelerator Research Infrastructure Cloud (FAbRIC). https://wikis.utexas.edu/display/fabric/Home [Last accessed: April 29, 2021], 2021.

[45] Huawei. FPGA Cloud. https://www.huaweicloud.com/en-us/product/fcs.html [Last accessed: April 29, 2021], 2021.

[46] Scaleflux. Applications. https://www.scaleflux.com/news.html [Last accessed: April 29, 2021], 2021.

[47] TheNextPlatform. Baidu Takes FPGA Approach to Accelerating SQL at Scale. https://www.nextplatform.com/2016/08/24/baidu-takes-fpga-approach-accelerating-big-sql/ [Last accessed: April 29, 2021], 2021.

[48] Xilinx. FPGAs: The Key to Accelerating High-Speed Storage Systems. https://www.flashmemorysummit.com/Proceedings2019/08-07-Wednesday/20190807_Keynote11_Xilinx_Raje.pdf [Last accessed: April 29, 2021], 2021.

[49] AWS. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://pages.awscloud.com/AQUA_Preview.html [Last accessed: April 29, 2021], 2021.

[50] AWS. Accelerate applications using Amazon EC2 F1 FPGA instances. https://d1.awsstatic.com/events/reinvent/2019/Accelerate_applications_using_Amazon_EC2_F1_FPGA_instances_CMP314.pdf [Last accessed: April 29, 2021], 2021.

[51] Baidu. AI Cloud. https://intl.cloud.baidu.com/product/abc-stack.html [Last accessed: April 29, 2021], 2021.

[52] Huawei. FPGA as a Service in the Cloud. https://indico.cern.ch/event/669648/contributions/2838181/attachments/1581893/2500031/Huawei_Cloud_FPGA_as_a_Service_CERN_openlab.pdf [Last accessed: April 29, 2021], 2021.

[53] Intel. FPGA Programmable Acceleration Card D5005. https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/intel-fpga-pac-d5005/overview.html [Last accessed: April 29, 2021], 2021.

[54] Xilinx. Alveo SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo.html [Last accessed: April 29, 2021], 2021.

[55] J. Weerasinghe. *Standalone Disaggregated Reconfigurable Computing Platforms in Cloud Data Centers*. PhD thesis, Technical University Munich, 2017.

[56] Intel. Intel FPGAs Power Acceleration-as-a-Service for Alibaba Cloud. https://newsroom.intel.com/news/intel-fpgas-power-acceleration-as-a-service-alibaba-cloud/#gs.uijjhu [Last accessed: April 29, 2021], 2021.

[57] TheNextPlatform. Another Step towards FPGAs in Supercomputing. https://www.nextplatform.com/2018/04/04/another-step-toward-fpgas-in-supercomputing/ [Last accessed: April 29, 2021], 2021.

[58] Xilinx. Alveo SN1000 Accelerator Card. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html [Last accessed: April 29, 2021], 2021.

[59] Intel. Intelligent Infrastructure Transformation. https://newsroom.intel.com/news/intel-baidu-drive-intelligent-infrastructure-transformation [Last accessed: April 29, 2021], 2021.

[60] TheNextPlatform. A Deep Dive into Cisco's Use of Merchant Switch Chips. https://www.nextplatform.com/2018/06/20/a-deep-dive-into-ciscos-use-of-merchant-switch-chips/ [Last accessed: April 29, 2021], 2021.

[61] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, 3:181–193, 2021.

[62] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt. Enabling VirtIO Driver Support on FPGAs. In *8th International Workshop on Heterogeneous High Performance Reconfigurable Computing*, 2022. doi: 10.1109/H2RC56700.2022.00006.

[63] S. Bandara, N. Cherry, and M. Herbordt. Fully Transparent Client-Side Caching for Key-Value Store Applications Using FPGAs. In *IEEE High Performance Extreme Computing Conference*, 2024.

[64] S. Bandara, A. Ducimo, C. Wu, and M. Herbordt. Long-Range MD Electrostatics Force Computation on FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 35(10):1690–1707, 2024. doi: 10.1109/TPDS.2024.3434347.

[65] S. Bandara, A. Sanaullah, Z. Tahir, U. Drepper, and M. Herbordt. Performance Evaluation of VirtIO Device Drivers for Host-FPGA PCIe Communication. In *31st Reconfigurable Architectures Workshop (RAW)*, 2024. doi: 10.1109/IPDPSW63119.2024.00043.

[66] AWS. Marketplace. `https://aws.amazon.com/marketplace/search/results?x=0&y=0&searchTerms=fpga` [Last accessed: April 29, 2021], 2021.

[67] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest. Maxwell - a 64 FPGA Supercomputer. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 287–294, 2007. doi: 10.1109/AHS.2007.71.

[68] A. Brauckmann, A. Goens, and J. Castrillon. A Reinforcement Learning Environment for Polyhedral Optimizations. *arXiv preprint arXiv:2104.13732*, 2021.

[69] S. J. Beaty. Genetic algorithms and instruction scheduling. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 206–211, 1991.

[70] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[71] T. Ben-Nun, B. Ates, A. Calotoiu, and T. Hoefler. Bridging control-centric and data-centric optimization. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 173–185, 2023.

[72] Berkley. OpenDwarfs Benchmarks. `https://github.com/vtsynergy/OpenDwarfs` [Last accessed: Mar 27, 2023], 2023.

[73] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[74] C. Blackmore, O. Ray, and K. Eder. Automatically tuning the gcc compiler to optimize the performance of applications running on embedded systems. *arXiv preprint arXiv:1703.08228*, 2017.

[75] Blaiech, Ahmed Ghazi and Khalifa, Khaled Ben and Valderrama, Carlos and Fernandes, Marcelo AC and Bedoui, Mohamed Hedi. A Survey and Taxonomy of FPGA-based Deep Learning accelerators. *Journal of Systems Architecture*, 98:331–345, 2019.

[76] C. Bobda, J. Mandebi, P. Chow, M. Ewais, N. Tarafdar, J. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofstee, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier. The Future of FPGA

Acceleration in Datacenters and the Cloud. *ACM Transactions on Reconfigurable Technology and Systems*, 15(3):1–42, 2022. doi: 10.1145/3506713.

[77] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on profile and feedback-directed compilation*, 1998.

[78] T. Boku, R. Kobayashi, N. Fujita, H. Amano, K. Sano, T. Hanawa, and Y. Yamaguchi. Cygnus: GPU meets FPGA for HPC. In *International Conference on Supercomputing*, 2019. https://www.r-ccs.riken.jp/labs/lpnctrt/assets/img/lspanc2020jan_boku_light
.pdf.

[79] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, et al. From Software to Accelerators with LegUp High-level Synthesis. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–9. IEEE, 2013.

[80] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al. A Cloud-Scale Acceleration Architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[81] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197. IEEE, 2007.

[82] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[83] J. Chen, N. Xu, P. Chen, and H. Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021.

[84] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu. Deconstructing Iterative Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):1–30, 2012.

[85] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sỳkora, S. Amarasinghe, and M. Carbin. Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 167–177. IEEE, 2019.

[86] Chung, Eric and Fowers, Jeremy and Ovtcharov, Kalin and Papamichael, Michael and Caulfield, Adrian and Massengill, Todd and Liu, Ming and Lo, Daniel and Alkalay, Shlomi and Haselman, Michael and others. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2): 8–20, 2018.

[87] CodeForces. GCC Optimization Pragmas. `https://codeforces.com/blog/entry/96344` [Last accessed: May 11, 2023], 2023.

[88] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang. FPGA HLS today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[89] C. Conger, I. Troxel, D. Espinoza, V. Aggarwal, and A. George. NARC: Network Attached Reconfigurable Computing for High Performance, Network Based Applications. In *Proceedings of the Eighth Annual International Conference on Military and Aerospace Programmable Logic Devices (MAPLD'05)*, 2005.

[90] R. A. Cooke and S. A. Fahmy. Characterizing Latency Overheads in the Deployment of FPGA Accelerators. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 347–352. IEEE, 2020.

[91] CTuning. Reproducing MILEPOST Project using CK Framework (Machine Learning based Self-tuning Compiler). `https://github.com/ctuning/reproduce-milepost-project?tab=readme-ov-file` [Last accessed: Nov 7, 2024], 2017.

[92] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, 2017. doi: 10.1109/PACT.2017.24.

[93] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. O'Boyle, and H. Leather. ProgramL: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *International Conference on Machine Learning*, pages 2244–2253. PMLR, 2021.

[94] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, et al. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *International Symposium on Code Generation and Optimization (CGO)*, 2022.

[95] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[96] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[97] CXL. CPU-to-Device Interconnect. `https://www.computeexpresslink.org/about-cxl` [Last accessed: April 29, 2021], 2021.

[98] A. F. Da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimaraes, and F. M. Q. Pereira. 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)*, pages 378–390. IEEE, 2021.

[99] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12): 36–42, 2009.

[100] DavidMalcolm. GCC Plugin that Embeds CPython inside the Compiler. `https://github.com/davidmalcolm/gcc-python-plugin` [Last accessed: Nov 7, 2024], 2011.

[101] J. Davidson, G. Tyson, D. Whalley, P. Kulkarni, J. Davidson, G. Tyson, D. Whalley, and P. Kulkarni. Evaluating Heuristic Optimization Phase Order Search Algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 157–169, 2007. doi: 10.1109/CGO.2007.9.

[102] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2020.

[103] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, 2007.

[104] Eideticomm. NVMe Computational Storage. `https://www.eideticom.com/` [Last accessed: April 29, 2021], 2021.

[105] D. Ernst. Competing in Artificial Intelligence Chips: China's Challenge amid Technology War. *Centre for International Governance Innovation, Special Report*, 2020.

[106] L. Ferretti, G. Ansaloni, and L. Pozzi. Cluster-based heuristic for high level synthesis design space exploration. *IEEE Transactions on Emerging Topics in Computing*, 9(1):35–43, 2021. doi: 10.1109/TETC.2018.2794068.

[107] Forbes. Xilinx FPGAs: The Chip Behind Alibaba's Singles Day. https://www.forbes.com/sites/moorinsights/2018/11/29/xilinx-fpgas-the-chip-behind-alibabas-singles-day/?sh=5f2294e27e3b [Last accessed: April 29, 2021], 2021.

[108] N. Fujita, R. Kobayashi, Y. Yamaguchi, and T. Boku. Parallel Processing on FPGA Combining Computation and Communication in OpenCL Programming. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488. IEEE, 2019.

[109] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[110] U. Garciarena and R. Santana. Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1159–1166, 2016.

[111] GCC. Sparse Conditional Constat Propagation. http://gnu.ist.utl.pt/software/gcc/news/ssa-ccp.html [Last accessed: Nov 7, 2024], 2024.

[112] GCC. x86 Options. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html [Last accessed: Nov 7, 2024], 2024.

[113] GCC-GNU. Other Built-in Functions Provided by GCC. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html [Last accessed: May 11, 2023], 2023.

[114] GCC GNU. Options that control optimizations. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html [Last accessed: May 12, 2023], 2023.

[115] A. George, H. Lam, and G. Stitt. Novo-G: At the Forefront of Scalable Reconfigurable Supercomputing. *Computing in Science & Engineering*, 13(1):82–86, 2010.

[116] A. George, M. Herbordt, H. Lam, A. Lawande, J. Sheng, and C. Yang. Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2016. doi: 10.1109/ HPEC.2016. 7761639.

[117] Global Industry Analysts, Inc. Data Center FPGA Market. https://www.researchandmarkets.com/reports/4804594/data-center-accelerators-global-market\#pos-0 [Last accessed: April 29, 2021], 2021.

[118] GNU. GCC Command Options. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html [Last accessed: Nov 7, 2024], 2024.

[119] Z. Gong, Z. Chen, J. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau, et al. An empirical study of the effect of source-level loop transformations on compiler stability. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[120] J.-M. Gorius, S. Rokicki, and S. Derrien. SpecHLS: Speculative accelerator design using high-level synthesis. *IEEE Micro*, 42(5):99–107, 2022.

[121] D. Grubisic, C. Cummins, V. Seeker, and H. Leather. Compiler generated feedback for large language models. *arXiv preprint arXiv:2403.14714*, 2024.

[122] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[123] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt. FCsN: A FPGA-Centric SmartNIC Framework for Neural Networks. In *30th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2022. DOI: 10.1109/FCCM53951.2022.9786193.

[124] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt. A Framework for Neural Network Inference on FPGA-Centric SmartNICs. In *International Conference on Field-Programmable Logic and Applications*, 2022. DOI: 10.1109/FPL57034.2022.00071.

[125] A. Guo, Y. Hao, C. Wu, P. Haghi, Z. Pan, M. Si, D. Tao, A. Li, M. Herbordt, and T. Geng. Software-hardware co-design of heterogeneous smartnic system for recommendation models inference and training. In *ICS 2023: International Conference on Supercomputing*, 2023. DOI = 10.1145/3577193.3593724.

[126] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt. FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020. DOI: 10.1109/ FCCM48280.2020.00028.

[127] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt. Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study. In *IEEE Conference on Field Programmable Technology*, 2020.

[128] P. Haghi, A. Guo, Q. Xiong, R. Patel, C. Yang, T. Geng, J. Broaddus, R. Marshall, A. Skjellum, and M. Herbordt. FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives. In *IEEE High Performance Extreme Computing Conference*, 2020.

[129] P. Haghi, A. Guo, Q. Xiong, C. Yang, T. Geng, J. Broaddus, R. Marshall, D. Schafer, A. Skjellum, and M. Herbordt. Reconfigurable switches for high performance and flexible mpi collectives. *Concurrency and Computation: Practice and Experience*, 34(2), 2022. doi: 10.1002/cpe.6769.

[130] P. Haghi, W. Krska, C. Tan, T. Geng, P. Chen, C. Greenwood, A. Guo, T. Hines, C. Wu, A. Li, A. Skjellum, and M. Herbordt. Flash: Fpga-accelerated smart switches with gcn case study. In *37th ACM International Conference on Supercomputing (ICS)*, 2023. DOI = 10.1145/3577193.3593739.

[131] P. Haghi, C. Tan, A. Guo, C. Wu, D. Liu, A. Li, A. Skjellum, T. Geng, and M. Herbordt. Smartfuse: Reconfigurable smart switches to accelerate fused collectives in hpc applications. In *38th ACM International Conference on Supercomputing (ICS)*, 2024. DOI: 10.1145/3650200.3656616.

[132] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[133] A. Haj-Ali, Q. J. Huang, J. Xiang, W. Moses, K. Asanovic, J. Wawrzynek, and I. Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[134] R. Han and H. Kim. Exponentially expanding the phase-ordering search space via dormant information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, pages 250–261, 2024.

[135] S. Handagal, M. Herbordt, and M. Leeser. OCT: The Open Cloud FPGA Testbed. In *31st International Conference on Field Programmable Logic and Applications (FPL)*, 2021.

[136] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE*

International Symposium on Circuits and Systems (ISCAS), pages 1192–1195, 2008. doi: 10.1109/ISCAS.2008.4541637.

[137] M. C. Herbordt. *The Evaluation of Massively Parallel Array Architectures*. University of Massachusetts Amherst, 1994.

[138] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson. Formal verification of high-level synthesis. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

[139] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 2008.

[140] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):1–26, 2015.

[141] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. Autophase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. *arXiv preprint arXiv:2003.00671*, 2020.

[142] Intel. Intel® High Level Synthesis Compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html` [Last accessed: April 18, 2021], 2022.

[143] Intel. Intel® High Level Synthesis Compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html` [Last accessed: October, 2023], 2023.

[144] Intel. Modelsim FPGA edition. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html/` [Last accessed: August 19, 2022], 2023.

[145] Intel. Intel® Rendering Framework Using Software-Defined Visualization. `https://www.intel.com/content/dam/develop/public/us/en/documents/parallel-universe-issue-35.pdf` [Last accessed: February 14, 2024], 2024.

[146] Intel. Intel® Architecture Code Analyzer. `https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html` [Last accessed: July 2, 2024], 2024.

[147] Inventec. FPGA SmartNIC C5020X. `https://ebg.inventec.com/en/product/Accessories/Smart\%20NIC\%20Card/Inventec\%20FPGA\%20SmartNIC\%20C5020X` [Last accessed: April 29, 2021], 2021.

[148] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456. PMLR, 2015.

[149] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*, pages 196–205. Springer, 2005.

[150] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert. Towards compile-time-reducing compiler optimization selection via machine learning. In *50th International Conference on Parallel Processing Workshop*, pages 1–6, 2021.

[151] G. Jo, H. Kim, J. Lee, and J. Lee. SOFF: An opencl high-level synthesis framework for FPGAs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 295–308, 2020. doi: 10.1109/ISCA45697.2020.00034.

[152] H. Jun, H. Ye, H. Jeong, and D. Chen. AutoScaleDSE: a scalable design space exploration engine for high-level synthesis. *ACM Transactions on Reconfigurable Technology and Systems*, 16(3):1–30, 2023.

[153] C. Kachris and D. Soudris. A Survey on Reconfigurable Accelerators for Cloud Computing. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10. IEEE, 2016.

[154] K. Kikuchi, N. Fujita, R. Kobayashi, and T. Boku. Implementation and performance evaluation of collective communications using CIRCUS on multiple FPGAs. In *Proceedings of the HPC Asia 2023 Workshops*, 2023. doi: 10.1145/3581576.3581602.

[155] M. Kim, J.-K. Park, and S.-M. Moon. Solving PBQP-Based Register Allocation using Deep Reinforcement Learning. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, 2022. doi: 10.1109/CGO53902.2022.9741272.

[156] O. Knodel, P. R. Genssler, and R. G. Spallek. Virtualizing reconfigurable hardware to provide scalability in cloud architectures. In *International Conference on Advances in Circuits, Electronics and Micro-electronics (CENICS)*, pages 33–38. sn, 2017.

[157] R. Kobayashi, Y. Oobata, N. Fujita, Y. Yamaguchi, and T. Boku. OpenCL-Ready High Speed FPGA Network for Reconfigurable High Performance Computing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 192–201, 2018.

[158] D. Koch, F. Hannig, and D. Ziener. *FPGAs for Software Programmers*. Springer International Publishing, 2016. ISBN 9783319264080. URL `https://books.google.com/books?id=p-N6DAAAQBAJ`.

[159] J. Koo, P. Balaprakash, M. Kruse, X. Wu, P. Hovland, and M. Hall. Customized Monte Carlo tree search for LLVM/Polly's composable loop optimization transformations. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 82–93. IEEE, 2021.

[160] D. Korolija, T. Roscoe, and G. Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 991–1010, 2020.

[161] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas. Opendwarfs: Characterization of Dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, 85:373–392, 2016.

[162] M. Kruse and H. Finkel. User-directed loop-transformations in Clang. In *5th Workshop on LLVM Compiler Infrastructure in HPC*, 2018.

[163] S. Kulkarni and J. 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*, pages 147–162, 2012.

[164] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.

[165] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[166] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*, pages 121–131. IEEE, 2018.

[167] H. Leather and C. Cummins. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2020.

[168] LegUp. High-Level Synthesis Open Source Software. http://legup.eecg.utoronto.ca/ [Last accessed: Feb 28, 2023], 2023.

[169] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. J. Leather, et al. Learning compiler pass orders using coreset and normalized value prediction. In *International Conference on Machine Learning*, pages 20746–20762. PMLR, 2023.

[170] S.-C. Lin, C.-K. Chang, and N.-W. Lin. Automatic selection of gcc optimization options using a gene weighted genetic algorithm. In *2008 13th Asia-Pacific Computer Systems Architecture Conference*, pages 1–8. IEEE, 2008.

[171] Z. Lin. *Performance Modeling and Optimization for Machine Learning Workloads*. University of California, Davis, 2023.

[172] LLVM. LLVM opt -stats. https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/InstCount.cpp [Last accessed: Nov 7, 2024], 2018.

[173] LLVM. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html [Last accessed: Aug 10, 2022], 2023.

[174] LLVM. llvm-mca - LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html [Last accessed: July 2, 2024], 2024.

[175] LLVM-MCA. LLVM's Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html [Last accessed: May 20, 2023], 2023.

[176] R. Mammadli, A. Jannesari, and F. Wolf. 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)*, pages 1–11. IEEE, 2020.

[177] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

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

[179] Matas, Kaspar and La, Tuan and Grunchevski, Nikola and Pham, Khoa and Koch, Dirk. Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 11–20, 2020.

[180] Mellanox. Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/files/doc-2020/pb-innova-2-flex.pdf [Last accessed: April 29, 2021], 2021.

[181] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

[182] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. Pereira. DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.

[183] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Summit*, pages 171–180, 2003.

[184] Microchip. SmartHLS High-Level Synthesis Software. https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/smarthls-compiler [Last accessed: Feb 28, 2023], 2023.

[185] Microsoft. ChatGPT- What? Why? And How? https://techcommunity.microsoft.com/blog/educatordeveloperblog/chatgpt--what-why-and-how/3799381 [Last accessed: Nov 7, 2024], April 2023.

[186] Microsoft Azure. Deploy ML models to FPGAs with Azure Machine Learning. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/ [Last accessed: April 29, 2021], 2021.

[187] N. Mohammedali and M. O. Agyeman. A study of Reconfigurable Accelerators for Cloud Computing. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, pages 1–5, 2018.

[188] R. Mosaner, D. Leopoldseder, L. Stadler, and H. Mössenböck. Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 127–135, 2021.

[189] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[190] R. Munafo, H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt. Improved models for policy-agent learning of compiler directives in hls. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2023.

[191] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, pages 197–206, 2010.

[192] Napatech. FPGA acceleration cards. `https://www.napatech.com/products/` [Last accessed: April 29, 2021], 2021.

[193] NewWaveDV. 32-Port Programmable Switch. `http://newwavedv.com/wordpress/wp-content/uploads/2019/04/32-Port-Programmable-Switch-Datasheet.pdf` [Last accessed: April 29, 2021], 2021.

[194] OpenAI. Proximal Policy Optimization. `https://openai.com/research/openai-baselines-ppo` [Last accessed: Nov 7, 2024], 2017.

[195] OpenCAPI. A New Standard for High Performance Memory, Acceleration and Networks. `https://opencapi.org/2017/04/opencapi-new-standard-high-performance-memory-acceleration-networks/` [Last accessed: April 29, 2021], 2021.

[196] Paderborn Center for Parallel Computing, University of Padeborn. Noctua. `https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua` [Last accessed: April 29, 2021], 2021.

[197] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 12–pp. IEEE, 2006.

[198] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 41(5):704–750, 2013.

[199] E. J. Park. *Automatic Selection of Compiler Optimizations using Program Characterization and Machine Learning*. University of Delaware, 2015.

[200] R. Patel, P.-F. Wolfe, R. Munafo, M. Varia, and M. Herbordt. Arithmetic and Boolean Secret Sharing MPC on FPGAs in the Data Center. In *IEEE High Performance Extreme Computing Conference*, 2020. doi: TBD.

[201] R. Patel, P. Haghi, S. Jain, A. Kot, V. Krishnan, M. Varia, and M. Herbordt. COPA Use Case: Distributed Secure Joint Computation. In *30th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2022. doi: 10.1109/FCCM53951.2022.9786156.

[202] C. Plessl. Bringing FPGAs to HPC Production Systems and Codes. In *H2RC'18 workshop at Supercomputing (SC'18)*, 2018. doi: 10.13140/RG.2.2.34327.42407.

[203] M. Poorhosseini, W. Nebel, and K. Grüttner. A compiler comparison in the risc-v ecosystem. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE, 2020.

[204] L.-N. Pouchet et al. Polybench: The Polyhedral Benchmark Suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437:1–1, 2012.

[205] PRNewsWire. NVMe Production Ready System. `https://www.prnewswire.com/news-releases/eideticom-ibm-rackspace-and-xilinx-demonstrate-worlds-first-pcie-gen4-nvm-express-production-ready-system-676532203.html` [Last accessed: April 29, 2021], 2021.

[206] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[207] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Data Center Services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014. doi: 10.1109/ISCA.2014.6853195.

[208] M. Rahman, L.-N. Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*, 2010.

[209] P. Ramos, G. Souza, D. Soares, G. Araújo, and F. M. Q. Pereira. Automatic annotation of tasks in structured code. In *Int. Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[210] A. Renda, Y. Chen, C. Mendis, and M. Carbin. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–455. IEEE, 2020.

[211] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 854–868, 2020.

[212] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather. Hyfm: Function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 110–121, 2021.

[213] A. Sanaullah. *Towards Hardware as a Reconfigurable, Elastic, and Specialized Service*. PhD thesis, Department of Electrical and Computer Engineering, Boston University, 2019. OpenBU URL https://hdl.handle.net/2144/38517.

[214] A. Sanaullah and M. Herbordt. FPGA HPC using OpenCL: Case Study in 3D FFT. In *9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, page 1–6, 2018. doi: 10.1145/3241793.3241800.

[215] A. Sanaullah and M. Herbordt. Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018. doi: 10.1109/HPEC.2018.8547646.

[216] A. Sanaullah, R. Patel, and M. Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 46–53. IEEE, 2018.

[217] A. Sanaullah, V. Sachdeva, and M. Herbordt. SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels. In *IEEE 4th Annual Heterogeneous High Performance Reconfigurable Computing*, 2018. doi: 10.1186/s12859-018-2505-7.

[218] B. C. Schafer and Z. Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2020. doi: 10.1109/TCAD.2019.2943570.

[219] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[220] Sdxcentral. Broadcom Sharpens Tomahawk Switch Chips, Versatile SmartToR. `https://www.sdxcentral.com/articles/news/broadcom-sharpens-tomahawk-switch-chips-versatile-smarttor/2020/12/` [Last accessed: April 29, 2021], 2021.

[221] V. Seeker, C. Cummins, M. Cole, B. Franke, K. Hazelwood, and H. Leather. Revealing compiler heuristics through automated discovery and optimization. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 55–66. IEEE, 2024.

[222] H. Shahzad, A. Sanaullah, and M. Herbordt. Survey and future trends for fpga cloud architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10. IEEE, 2021.

[223] H. Shahzad, A. Sanaullah, S. Arora, R. Munafo, X. Yao, U. Drepper, and M. Herbordt. Reinforcement learning strategies for compiler optimization in high level synthesis. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 13–22. IEEE, 2022.

[224] H. Shahzad, A. Sanaullah, , S. Arora, U. Drepper, and M. Herbordt. Neural Network based GCC Cost Model for Accelerating Compiler Tuning Workloads. In *IEEE High Performance Extreme Computing Conference*, 2024.

[225] H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt. Autoannotate: Reinforcement learning based code annotation for high level synthesis. In *2024 25th International Symposium on Quality Electronic Design (ISQED)*, pages 1–9. IEEE, 2024.

[226] J. Sheng, C. Yang, and M. Herbordt. Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2015. https:// pdfs.semanticscholar.org /832d/ c69145f5ba0ed6a951583201b1b20dd 2096e.pdf.

[227] J. Sheng, Q. Xiong, C. Yang, and M. Herbordt. Collective Communication on FPGA Clusters with Static Scheduling. *ACM SIGARCH Computer Architecture News*, 44(4), 2017. doi: 10.1145/ 3039902.3039904.

[228] J. Sheng, C. Yang, A. Caulfield, M. Papamichael, and M. Herbordt. HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *27th International Conference on Field Programmable Logic and Applications*, 2017. doi: 10.23919/ FPL.2017.8056853.

[229] J. Sheng, C. Yang, and M. Herbordt. High Performance Dynamic Communication on Reconfigurable Clusters (Extended Abstract). In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–219, 2018. doi: 10.1109/ FCCM.2018.00053.

[230] J. Sheng, C. Yang, and M. Herbordt. High Performance Dynamic Communication on Reconfigurable Clusters. In *28th International Conference on Field Programmable Logic and Applications*, 2018. doi: 10.1109/ FPL.2018.00044.

[231] Silicom. FPGA SmartNIC N5010 Series. https://www.silicom-usa.com/pr/fpga-based-cards/fpga-intel-based/fpga-intel-strat ix-based/silicom-fpga-smartnic-n5010_series/ [Last accessed: April 29, 2021], 2021.

[232] A. F. d. Silva, B. N. De Lima, and F. M. Q. Pereira. Exploring the space of optimization sequences for code-size reduction: Insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 47–58, 2021.

[233] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[234] R. Skhiri, V. Fresse, J. P. Jamont, B. Suffran, and J. Malek. From FPGA to support cloud to cloud of FPGA: State of the art. *International Journal of Reconfigurable Computing*, 2019, 2019.

[235] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong. Automated accelerator optimization aided by graph neural networks. In *ACM/IEEE Design Automation Conference*, page 55–60, 2022.

[236] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong. AutoDSE: enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022.

[237] V. Sreenivasan, R. Javali, M. Hall, P. Balaprakash, T. R. Scogland, and B. R. de Supinski. A framework for enabling OpenMP autotuning. In *15th International Workshop on OpenMP*, pages 50–60. Springer, 2019.

[238] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *ACM sigplan notices*, 38(5):77–90, 2003.

[239] B. Sukhwani and M. Herbordt. Acceleration of a Production Rigid Molecule Docking Code. In *2008 International Conference on Field Programmable Logic and Applications*, pages 341–346, 2008. doi: 10.1109/ FPL.2008.4629955.

[240] B. Sukhwani and M. Herbordt. FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195, 2010. doi: 10.1049/ iet-cdt.2009.0013.

[241] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu. Correlated multi-objective multi-fidelity optimization for hls directives design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022.

[242] O. Sỳkora, P. M. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. Granite: A graph neural network model for basic block throughput estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 14–26. IEEE, 2022.

[243] Synopsys. An Introduction to CCIX. `https://www.synopsys.com/designware-ip/technical-bulletin/introduction-ccix-2017q3.html` [Last accessed: April 29, 2021], 2021.

[244] Z. Tahir, A. Sanaullah, S. Bandara, U. Drepper, and M. Herbordt. Multi-core Multi-rule VeBPF Firewall for Secure FPGA IoT Device Deployments. In *IEEE High Performance Extreme Computing Conference*, 2024.

[245] N. Tarafdar, N. Eskandari, T. Lin, and P. Chow. Designing for FPGAs in the Cloud. *IEEE Design and Test*, 35(1):23–29, 2017.

[246] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow. Enabling flexible network fpga clusters in a heterogeneous cloud data center. In *FPGA*, pages 237–246, 2017.

[247] N. Tarafdar, T. Lin, D. Ly-Ma, D. Rozhko, A. Leon-Garcia, and P. Chow. Building the Infrastructure for Deploying FPGAs in the Cloud. In *Hardware Accelerators in Data Centers*, pages 9–33. Springer, 2019.

[248] T. Theodoridis and Z. Su. Refined input, degraded output: The counterintuitive world of compiler behavior. *Proceedings of the ACM on Programming Languages*, 8(PLDI):671–691, 2024.

[249] T. Theodoridis, M. Rigger, and Z. Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709, 2022.

[250] S. Tian, W. Xiong, I. Giechaskiel, K. Rasmussen, and J. Szefer. Fingerprinting Cloud FPGA Infrastructures. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 58–64, 2020.

[251] TIBCO. Customers See Success with TIBCO on AWS. `https://www.tibco.com/blog/2018/11/26/tibco-customers-see-success-with-tibco-on-aws/` [Last accessed: April 29, 2021], 2021.

[252] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215. IEEE, 2003.

[253] S. Trimberger and S. McNeil. Security of FPGAs in Data Centers. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 117–122. IEEE, 2017.

[254] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *arXiv preprint arXiv:2101.04808*, 2021.

[255] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. LAMDA: Learning-assisted multi-stage autotuning for FPGA design closure. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 74–77. IEEE, 2019.

[256] A. Vaishnav, K. D. Pham, and D. Koch. A survey on fpga virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 131–1317. IEEE, 2018.

[257] T. VanCourt and M. Herbordt. Families of FPGA-based algorithms for approximate string matching. In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.*, pages 354–364, 2004. doi: 10.1109/ ASAP.2004.1342484.

[258] T. VanCourt and M. Herbordt. Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10, 2006. doi: 10.1155/ ASP/2006/97950.

[259] T. VanCourt and M. Herbordt. Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *IEEE Conference on Field Programmable Logic and Applications*, pages 395–401, 2006. doi: 10.1109/ FCCM.2006.25.

[260] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–27, 2020.

[261] K. Vipin and S. A. Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

[262] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, 1994.

[263] H. M. Waidyasooriya and M. Hariyama. Multi-FPGA Accelerator Architecture for Stencil Computation Exploiting Spacial and Temporal Scalability. *IEEE Access*, 7:53188–53201, 2019.

[264] F. Wang, W. Zhang, S. Lai, M. Hao, and Z. Wang. Dynamic gpu energy optimization for machine learning training workloads. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2943–2954, 2021.

[265] H. Wang, Z. Tang, C. Zhang, J. Zhao, C. Cummins, H. Leather, and Z. Wang. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 129–143, 2022.

[266] Z. Wang and M. O'Boyle. Machine Learning in Compiler Optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[267] Wang, Xiuxiu and Niu, Yipei and Liu, Fangming and Xu, Zichen. When FPGA Meets Cloud: A First Look at Performance. *IEEE Transactions on Cloud Computing*, 2020.

[268] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086, 2015. doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[269] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. Disaggregated FPGAs: Network Performance Comparison against Bare-Metal Servers, Virtual Machines and Linux Containers. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 9–17, 2016. doi: 10.1109/CloudCom.2016.0018.

[270] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. Network-Attached FPGAs for Data Center Applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016. doi: 10.1109/FPT.2016.7929186.

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

[272] P.-F. Wolfe, R. Patel, R. Munafo, M. Varia, and M. Herbordt. Secret Sharing MPC on FPGAs in the Datacenter. In *IEEE Conference on Field Programmable Logic and Applications*, 2020.

[273] C. Wu, T. Geng, S. Bandara, C. Yang, V. Sachdeva, W. Sherman, and M. Herbordt. Upgrade of FPGA Range-Limited Molecular Dynamics to Handle Hundreds of Processors. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021. doi: 10.1109/FCCM51124.2021.00024.

[274] C. Wu, S. Bandara, T. Geng, A. Guo, P. Haghi, W. Sherman, V. Sachdeva, and M. Herbordt. Optimized Mappings for Symmetric Range-Limited Molecular Force Calculations on FPGAs. In *International Conference on Field-Programmable Logic and Applications*, 2022. DOI: 10.1109/FPL57034.2022.00026.

[275] C. Wu, T. Geng, A. Guo, S. Bandara, P. Haghi, C. Liu, A. Li, and M. Herbordt. FASDA: An FPGA-Aided, Scalable and Distributed Accelerator for Range-Limited Molecular Dynamics. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023. DOI: 10.1145/3581784.3607100.

[276] C. Wu, C. Yang, S. Bandara, T. Geng, P. Haghi, A. Li, and M. Herbordt. FPGA-Accelerated Range-Limited Molecular Dynamics. *IEEE Transactions on Computers*, 73(6):1544–1558, 2024. doi: 10.1109/TC.2024.3375613.

[277] J. Wu, J. Xu, X. Meng, and Y. Lei. A Highly Reliable Compilation Optimization Passes Sequence Generation Framework. *IEICE Transactions on Information and Systems*, 103(9):1998–2002, 2020.

[278] N. Wu, H. Yang, Y. Xie, P. Li, and C. Hao. High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing. In *ACM/IEEE Design Automation Conference*, pages 49–54, 2022.

[279] N. Wu, Y. Xie, and C. Hao. IronMan-Pro: Multiobjective Design Space Exploration in HLS via Reinforcement Learning and Graph Neural Network-Based Modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(3):900–913, 2023. doi: 10.1109/TCAD.2022.3185540.

[280] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall. Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurrency and Computation: Practice and Experience*, 34(20), 2022.

[281] Xilinx. Xilinx Vitis Unified Software Platform. `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html` [Last accessed: Feb 22, 2023], 2023.

[282] Xilinx. Xilinx Runtime Library. `https://www.xilinx.com/products/design-tools/vitis/xrt.html` [Last accessed: Feb 22, 2023], 2023.

[283] Xilinx Case Study. Xilinx Powers Alibaba Cloud FaaS with AI Acceleration Solution for E-Commerce Business. https://www.xilinx.com/publications/powered-by-xilinx/xilinx-alibaba-case-study.pdf [Last accessed: April 29, 2021], 2021.

[284] Q. Xiong, A. Skjellum, and M. Herbordt. Accelerating MPI Message Matching Through FPGA Offload. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 191–1914, 2018. doi: 10.1109/FPL.2018.00039.

[285] Q. Xiong, C. Yang, R. Patel, T. Geng, A. Skjellum, and M. Herbordt. GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266, 2019. doi: 10.1109/FCCM.2019.00042.

[286] Q. Xiong, C. Yang, P. Haghi, A. Skjellum, and M. Herbordt. Accelerating MPI Collectives with FPGAs in the Network and Novel Communicator Support. In *IEEE Symposium on Field Programmable Custom Computing Machines*, 2020.

[287] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang. A parallel bandit-based approach for autotuning FPGA compilation. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 157–166, 2017.

[288] C. Yang, J. Sheng, R. Patel, A. Sanaullah, V. Sachdeva, and M. C. Herbordt. OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2017.

[289] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *International Symposium on High-Performance Computer Architecture*, pages 741–755, 2022.

[290] H. Ye, H. Jun, J. Yang, and D. Chen. High-level synthesis for domain specific computing. In *Proceedings of the 2023 International Symposium on Physical Design*, pages 211–219, 2023.

[291] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong. S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[292] M. Yu, S. Huang, and D. Chen. Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for FPGA high-level synthesis.

In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 524–536. Springer, 2021.

[293] Yu, Xiaoyu and Wang, Yuwei and Miao, Jie and Wu, Ephrem and Zhang, Heng and Meng, Yu and Zhang, Bo and Min, Biao and Chen, Dewei and Gao, Jianlin. A Data-Center FPGA Acceleration Platform for Convolutional Neural Networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 151–158. IEEE, 2019.

[294] Y. Zha and J. Li. Virtualizing FPGAs in the Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 845–858, 2020.

[295] C. Zhang, T. Geng, A. Guo, J. Tian, M. Herbordt, A. Li, and D. Tao. H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture. In *International Conference on Field-Programmable Logic and Applications*, 2022. DOI: 10.1109/FPL57034.2022.00040.

[296] J. Zhang, Y. Xiong, N. Xu, R. Shu, B. Li, P. Cheng, G. Chen, and T. Moscibroda. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, 2017.

[297] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 430–437. IEEE, 2017.

[298] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. Performance modeling and directives optimization for high-level synthesis on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39 (7):1428–1441, 2020. doi: 10.1109/TCAD.2019.2912916.

[299] R. Zhao, J. Cheng, W. Luk, and G. A. Constantinides. Polsca: Polyhedral high-level synthesis with compiler transformations. In *Int. Conf. on Field-Programmable Logic and Applications*, pages 235–242, 2022.

[300] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. Phothilimtha, S. Wang, A. Goldie, et al. Transferable graph optimizers for ml compilers. *Advances in Neural Information Processing Systems*, 33:13844–13855, 2020.

[301] M. Zhu and D. Hao. Compiler auto-tuning via critical flag selection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1000–1011. IEEE, 2023.

[302] M. Zhu, D. Hao, and J. Chen. Compiler autotuning through multiple-phase learning. *ACM Transactions on Software Engineering and Methodology*, 33(4): 1–38, 2024.

[303] Zhu, Zhuangdi and Liu, Alex X and Zhang, Fan and Chen, Fei. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing*, 2018.

[304] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, 2016.