ProgGen: Automatic Dataset Generation for the Halide Domain Specific Language

by

Zachary Holbrook

B.S. Computer Science and Engineering
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

ProgGen: Automatic Dataset Generation for the Halide Domain Specific Language

by

Zachary Holbrook

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Compilers use cost models to choose between different optimization opportunities, and increasingly these cost models are developed using data-driven techniques. Compilers for general-purpose languages rely on large real-world program datasets to train their cost models. However, cost models for domain-specific languages often have to use program generators due to a lack of large datasets of real-world programs. Program dataset generators are typically manually constructed or handwritten to generate programs in a randomly guided way. However, writing a program generator is time-consuming and requires considerable tuning to produce programs with realistic computation patterns in the desired domain.

This thesis presents ProgGen, a program generator inspired by genetic programming for automatically generating program datasets used in training compiler cost models. ProgGen automatically produces program datasets in different domains by starting with a small initial set of programs in the desired domain. I compare ProgGen with the random program generator used in the Halide Autoscheduler [1]. While the Halide random program generator performs better in the image processing and neural network domains it was designed for, ProgGen is competitive in video processing and linear algebra domains. Due to the automatic nature of ProgGen, ProgGen can also generate programs in new domains with far less engineering time.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I thank Charith Mendis for his advice and mentorship throughout my MEng research and preceding UROP work. I have worked closely with Charith throughout my time in the Commit Group, and I thank him for taking time to review my writing for this thesis and providing technical feedback on the MEng work. I would also like to thank my advisor Saman Amarasinghe for allowing me to join his group and working on this exciting project. I also thank him for his advice and support throughout the project.

Finally, I thank my parents, brother, and friends for their love and support during this journey.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Compilers use cost models in order to choose amongst mutually exclusive optimization opportunities. Increasingly, these cost models are based on machine learning rather than hand crafted analytical models because recent works have shown learned models to be more accurate [13, 1, 2]. Cost models based on machine learning require large datasets of programs in order to train. For example, Ithemal trains on 1.4 million unique x86 basic blocks [13]. Popular general purpose languages may have large repositories of programs that can be collected to form training datasets. For example, JSNice collects Javascript programs from over 10000 public Javascript Github repositories [18]. However, domain specific languages, like Halide and Tiramisu, don't usually have access to large program repositories and instead have to generate program datasets to use for training.

For domain specific languages, program datasets are usually generated with hand crafted random program generators that require considerable engineering and tuning to produce diverse program datasets that are representative of real programs. In the case of the random program generator for the Halide Autoscheduler [1], the generator is over a thousand lines of C++ code and defines 26 types of computation stages that it constructs programs with. For each kind of computation stage, it defines tuned probabilities that certain kinds of stages will come after it. The random program generator for the Tiramisu Autoscheduler [2] constructs programs by creating sequences of assignments, stencils, and reductions. To generate programs in new computation domains, these random program generators require defining new computation patterns and re-tuning. Because these generators are heavily integrated with their generated language, they require almost complete rewrites to generate programs in a different language.

This thesis presents ProgGen: a program generator for automatically generating program datasets inspired by genetic programming. Rather than creating programs out of sequences of pre-defined computation stages, ProgGen uses a context-free grammar to generate programs by mutating an initial small set of real world programs. ProgGen starts with a program population of 10 initial seed programs and grows this population by sampling and mutating programs from the population. To generate programs in a new language, ProgGen only requires 10 programs and a context-free grammar for the target language.

I evaluate ProgGen by generating Halide programs and comparing the Halide Autoscheduler's [1] cost model trained on programs generated by ProgGen versus programs generated by their hand crafted random program generator on benchmarks in four computation domains. When the Halide Autoscheduler uses a cost model trained with a program dataset generated by ProgGen, it gets better performance in linear algebra and video processing domains.

In this thesis, I make the following specific contributions.

- I present a genetic programming inspired program generator that can automatically generate programs in new domains and languages provided it is given 10 programs and a grammar for that language.

- I show that a cost model trained with Halide programs generated by ProgGen results in better Halide Autoscheduler performance on programs in linear algebra and video processing domains.

- I propose new measures of accuracy and error for compiler cost models and compare how well these measures correlate to downstream performance in the Halide Autoscheduler with well known measures.

1.1 Outline

The thesis is structured as follows. In Chapter 2, I provide background on the Halide Autoscheduler system [1] and its Halide random program generator. Then in Chapter 3, I present system and methodologies used by ProgGen. I provide a high level overview as well as in depth explanations on each component of ProgGen. I also describe attempts made to improve ProgGen that did not pan out and discuss why that might be the case. Next, I present empirical results and evaluations of the system compared to the random program generator found in the Halide Autoscheduler [1] in Chapter 4. I discuss related work in Chapter 5. I discuss limitations, future work, and conclude in Chapter 6.

Chapter 2

Background

Halide [17] is a domain specific language for writing high performance image and array processing code that targets multiple CPU and GPU architectures across multiple domains. A Halide program consists of two parts, an algorithm specification, and a schedule specification. The algorithm specification determines what computation is being performed, such as an image blur or a matrix multiply. The schedule specification determines how and in what order that computation is performed. For example, a schedule for an image blur might specify that the image is to be chunked into a grid of 16 blocks and the computation for each block is to be performed in parallel. The number of possible schedules for any given computation are enormous and the best schedule can be orders of magnitude faster than the worst schedule.

To address the problem of finding the best schedule, multiple Halide Autoschedulers [12, 14, 1] have been written to try and find the best performing schedules automatically. The state of the art Halide Autoscheduler uses a learned cost model to predict the runtime of a Halide algorithm and schedule [1]. The autoscheduler uses beam search to search through the space of possible schedules for a given Halide algorithm and uses the learned cost model to estimate the runtime of schedules without actually having to run them, which would drastically reduce the search speed.

In order to train their learned cost model, Adams et al. created a Halide random program generator that generates a training dataset of Halide algorithms [1]. This thesis compares ProgGen, a program generator inspired by genetic programming, to the Halide random program generator by training the learned cost model on datasets generated by either program generator.

The Halide random program generator [1] works as follows. The generator starts with color image like inputs, buffers with a width, height, and 3 channels, and ends with color image like outputs. The generator creates a directed graph of computation 'stages' that transforms the input into the output. Each computation stage may have multiple input stages or multiple output stages, but the graph of stages is constrained such that there is only one input that has no other inputs and one output that has no outputs from it.

Each computation stage has several properties: a function, width, height, and channels. The width, height, and channels properties define the size of the computation for that stage (stage sizes may change if there are up-sampling or down-sampling operations). The function property defines what computation that stage is performing, and it can fall within several defined function categories. These categories include patterns like convolutions, Relu operations, pooling operations, upsamples, downsamples, histogram operations, and more. Additionally, the Halide random program generator includes a transition matrix that specifies the probability that a specific function category occurs after another category. For example, it is more likely that a Relu or Pool stage occurs after a convolution stage. Altogether, the Halide random program generator creates programs that look similar to image processing programs or convolutional neural networks operating on images.

Chapter 3

System Overview

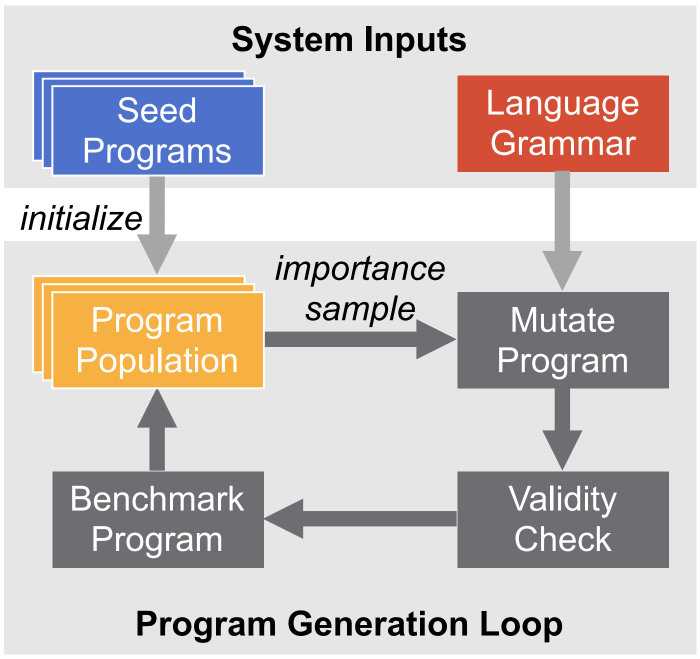


Figure 3-1: ProgGen starts with a set of seed programs and a grammar for the target programming language. ProgGen first uses the grammar to create abstract syntax trees out of the seed programs and then initializes the program population with those abstract syntax trees. From there, it continuously repeats a loop where it importance samples from its program population, mutates a program sample using the grammar rules, checks the validity of the mutated program, and benchmarks and adds the mutated program into its program population. The cycle stops when the program population grows to a specified size.

3.1 Overview

Figure 3-1 presents the high level architecture of ProgGen. ProgGen abstracts domain and programming language specific features into system inputs (Section 3.2) so that ProgGen can generate programs in new domains or languages automatically. More specifically, ProgGen requires a set of seed programs to initialize its population of programs (Section 3.2.1). These seed programs largely determine the kinds of computations ProgGen generates. ProgGen also requires a language grammar that it uses to parse seed programs, inform mutation operations, and to transform its internal program representations into text (Section 3.2.2).

Once ProgGen is provided with its required inputs, it can start generating programs in a program generation loop (Section 3.3). First, ProgGen importance samples a program from its program population (Section 3.3.1). Importance sampling ensures programs with a variety of runtimes and sizes are generated. Second, ProgGen mutates the sampled program using one of its four mutation operations (Section 3.3.2), which it bases off the grammar. Third, ProgGen performs validity checks on the mutated program (Section 3.3.3). If the mutated program passes the validity checks, then ProgGen benchmarks the mutated program and adds it to the program population. Otherwise, it throws out the invalid mutated program. ProgGen then starts its generation loop again by importance sampling a new program, possibly sampling the program it just mutated. ProgGen continues this loop until the program population grows to a desired size.

Additionally, I tried to incorporate program crossover into ProgGen to work alongside mutation operations. I found that program crossovers rarely produced working programs, so I attempted to include a learnt program crossover repair mechanism based on Tree LSTMs used for program translation [4]. This attempt was initially successful at repairing relatively small Halide programs resulting from crossover operations. However, once the Halide programs became much larger, in part due to importance sampling selecting for larger programs, the accuracy of the crossover repair mechanism became very small. Program crossover was therefore not used for the main project and evaluations, but in Section 3.4 I detail the attempt at program crossover that was made.

3.1.1 Genetic Algorithm Inspiration

ProgGen is inspired by genetic algorithms and has some similarities to traditional genetic algorithms, but is also different in key ways. Traditionally, genetic algorithms require two main things: a genetic representation of an individual or the solution domain, and a fitness function that measures how fit an individual in the population is. Genetic algorithms start with an initial population of individuals and then perform random mutations or crossovers on these individuals to generate a successor generation. Mutations change a single individual, and crossovers combine two individuals in some way to produce a new individual. Once the successor generation is created, the fitness function is used to select the most fit individuals of that generation to become the parents of the next generation. This process repeats for many generations and then the individual with the highest fitness according to the fitness function is selected as the final output of the genetic algorithm. Traditionally, the goal of a genetic algorithm is to produce an individual or solution that maximizes (or minimizes) the fitness function. Genetic programming is similar to genetic algorithms except that the individual is a program, often represented as an abstract syntax tree.

ProgGen maintains a population of programs represented as abstract syntax trees. It also performs mutations on individual programs, and at one point I tried to perform crossover operations on programs, but had limited success. However, ProgGen doesn't maintain successive generations of programs like in traditional genetic algorithms. Instead, it maintains a single large population of programs that continuously grows by mutating programs in this large population. This is because unlike in traditional genetic algorithms, ProgGen's goal isn't to produce a single best program, but rather an entire dataset of programs. Additionally, ProgGen doesn't have a fitness function in the traditional sense as ProgGen isn't trying to optimize particular programs, rather it is trying to maximize the diversity of the entire generated program dataset.

The diversity of a program dataset refers to the variety of computations performed by programs in that dataset. If programs in a dataset perform a wider variety of different kinds of computations than another dataset, then it is more diverse. So instead of using a fitness function to sample or retain the best programs in each generation, it uses importance sampling to sample and mutate a broad range of programs in its program population. I found that importance sampling based on a target distribution for different properties, like runtime or schedule stage count, allowed diverse program datasets to be generated.

3.2 System Inputs

Typically, program generators are engineered to generate programs in a specific language with a specific computation domain in mind. For example, the Halide random program generator [1] generates Halide programs that primarily perform image processing and neural network computations. I made ProgGen more general by abstracting the domain and language specific parts of the program generator into two generator inputs: seed programs and a language grammar. Mutation operations are general and are based off of the grammar. The grammar determines the programming language the programs are generated in, and the seed programs determine the kinds of computations generated. This abstraction allows ProgGen to generate programs in a different language or domain with less engineering effort. For example, I was able to switch from generating C programs to Halide programs by just writing a new EBNF grammar and collecting new seed programs. This process only took a handful of hours. In comparison, writing an entirely new program generator might have taken around a hundred hours.

3.2.1 Seed Programs

In order for ProgGen to generate programs by mutation in its generation loop, it needs an initial population of programs. I call the initial population it uses the seed programs because they provide a seed for the mutation and generation loop to start from. The seed programs largely determine what the generated programs will look like because the generated programs can only be so many mutations away from the seed programs. For example, in one dataset of 7000 generated programs, generated programs were at most 70 mutations, and a mean of 30 mutations away from a seed program. Domain specific language like Halide are designed to concisely express computations, so 30 mutations can create programs that perform very different computations.

For example, a brightness filter that performs the following computation:

$$\text{output}(x, y, c) = \text{input}(x, y, c) * 1.5$$

was transformed to the following after 6 mutations:

$$\text{output}(y, x, c) = \text{fast\_log}(\text{input}(x, y, c))/x + c$$

Each seed program specifies its input size and output size. For example, the greyscale filter takes in a color image represented as a tensor with 3 dimensions (width, height, and color channels) and outputs a greyscale image which is represented as a tensor with 2 dimensions (width and height). Whereas a video brighten filter takes in a video represented as a tensor with 4 dimensions (width, height, color channels, and time) and outputs a video with the same dimensions and size. Mutations are generally unable to change the input and output dimensions of a program because doing so would require many particular changes to make such a change valid. Each seed dataset I used for our evaluations contained only 10 programs (Section 4.1.1). I used seed datasets containing 50 programs, but those didn't have a downstream performance improvement compared to starting with 10 seed programs, so I used 10 seed programs for the evaluations.

3.2.2 Halide Grammar

```
body: statement | statement body
statement: for_statement
```

```
    | declaration
    ...
declaration: type name_list ";"
call_statement: function_call ";"
function_call: halide_function | var_call | method_call
for_statement: "for" "(" assignment_declaration expr ";"
   assignment ")" "{" body "}"
expr: sum
    | expr "==" sum
    | expr ">=" sum
    ...
halide_function: cast | min | max | ceil | floor | round
   | abs | sum_func | pow | fast_exp | exp | fast_log |
   sqrt | sin | tanh | cos | undef | repeat_edge
cast: "cast" "<" PLAIN_TYPE ">" "(" expr ")"
min: "min" "(" arg_list ")"
...
```

Listing 3.1: A snippet of the Halide Grammar written using the Lark package from Python. Non-terminals are lowercase and terminals are uppercase. String literals are surrounded in quotation marks. A grammar rule consists of a non-terminal on the left, followed by a colon, and then one or more expansions separated by a vertical bar ("|"). An ellipsis denotes parts of the full grammar left out of this snippet. See Listing B.1 in the Appendix for the full Halide grammar used and a description of the syntax used for the grammar.

ProgGen uses an input grammar for parsing seed programs into abstract syntax trees (ASTs), informing the mutation operations how to mutate program ASTs, and turning program ASTs into text in order to compile them and perform validity checks.

I observed that the Halide random program generator would have required a rewrite of all the AST structures and related functions in order to generate programs in a different language. This would require significant engineering time. ProgGen could have used similar internal Halide AST structures and Halide specific mutation operations, but then it would take significant effort to use ProgGen to generate programs in a different language other than Halide. Instead, ProgGen uses a grammar to define its AST and mutation operations. This allows ProgGen to perform mutation operations on a new language by just using a new grammar for that language.

The grammar ProgGen uses for Halide is in EBNF (Extended Backus-Naur Form), and was written using the Lark Python package. It contains general structures like if statements, loops, variable declarations, and other language constructs used in Halide. A snippet of the grammar is shown in Listing 3.1 and the full grammar is shown in Listing B.1. The grammar generally allows any function to be called with any number of arguments, thereby ensuring any Halide function call made by the seed programs can be parsed and modified with mutation operations. However, the grammar also specifically includes 18 Halide functions by specifying the function name and the number of arguments (but not the argument type since that is context dependent) in the grammar rules.

The Halide functions included in the grammar are: cast, min, max, ceil, floor, round, abs, sum, pow, fast_exp, exp, fast_log, sqrt, sin, tanh, cos, undef, repeat_edge. These functions are used by the Halide random program generator and including them in the grammar ensures that mutation operations can add them into a program even if the program did not originally have those functions in it. This is important because it allows programs with more types of functions in it to be generated. This increases the diversity of computations performed in a generated dataset and I hypothesize it to make training neural networks using that generated training set more generalized.

The Halide grammar I used is 110 lines long, contains 44 nonterminals, 65 terminals, and 100 grammar rules (some nonterminals have multiple rules describing how they can be expanded in different ways). Writing the grammar took me only a couple hours. ProgGen automatically produces a simplified version of the grammar that it uses to construct and mutate ASTs. The grammar is simplified by using some annotations that specify which grammar rules are used purely for precedence (for example, parenthesized expressions). ProgGen detects cycles of non-terminals that can each be transformed into each other ignoring precedence symbols like parenthesis. These cycles of equivalent non-terminals modulo precedence are simplified into the same non-terminal. Precedence symbols aren't important in an AST because precedence is already determined by the structure of the tree. ProgGen also removes uniquely determined string literal terminals that aren't required in an AST (as opposed to regex matching terminals like variable names and numbers). After these simplifications, the simplifed grammar used for generating and mutating ASTs include 41 nonterminals, 10 terminals and 96 rules. The original grammar is still used to parse program texts and convert program ASTS into program texts.

3.3 Program Generation Loop

ProgGen repeatedly mutates programs in a population of programs and adds valid mutations back into that population. At the start of each generation loop, ProgGen importance samples a program from its program population based on runtime and schedule stage counts (Section 3.3.1). ProgGen then mutates the sampled program using one of four mutation operations informed by the grammar (Section 3.3.2).

ProgGen then performs validity checks on the mutated program including checking that it compiles and runs (Section 3.3.3). If the mutated program is not valid, then ProgGen throws it out and starts the generation loop again. If it is valid, ProgGen benchmarks the program to get its runtime and also counts the number of Halide schedule stages in the program. ProgGen then adds the valid mutated program into its program population and starts the loop again.

3.3.1 Importance Sampling

I found that sampling programs at random from the program distribution led to a biased program distribution. Programs that were short and easy to mutate correctly, by virtue of not having very many complex computations, out competed the rest of

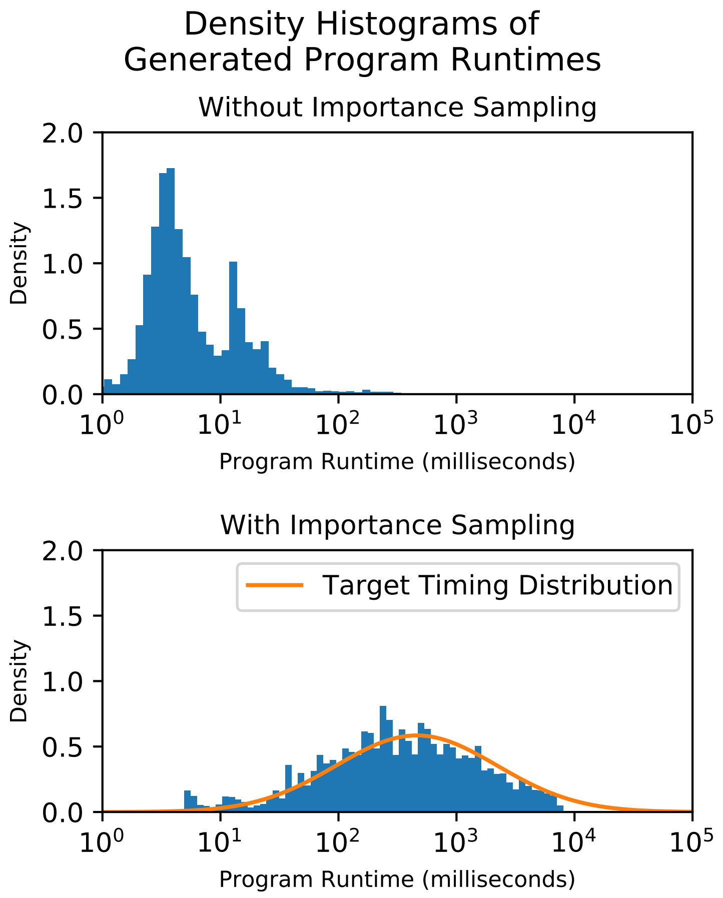


Figure 3-2: Density histograms of the log runtimes of schedules ProgGen generated without using importance sampling and with using importance sampling. I measured program runtime by benchmarking a Halide schedule from each program ProgGen generated. The graph with importance sampling also displays the log-normal probability density function that was used for importance sampling. Using importance sampling significantly changes the runtime distribution of programs generated and results in a distribution closely matching the target distribution.

the programs. This caused the generated program datasets to be mostly filled with small programs that had short runtimes.

To get around this, I biased the sampling procedure as follows. First, I defined a target program runtime and schedule stage count distribution. For program runtime, I used a base 10 log-normal distribution with a mean of 450 milliseconds and a base 10 log standard deviation of 0.7. For the case of schedule stage count, I used a normal distribution with a mean of 17 and a standard deviation of 5.

I got these distributions by measuring the runtime and schedule stage counts of the Halide random program generator and fitting probability density functions to them using SciPy distribution fitting functions. I tried fitting several different kinds of distributions, like a gamma, normal, log-normal, and more distributions, and I chose the distributions that were fitted with the smallest error. Although the target distributions are based on the Halide random program generator's distribution, we could define distributions without it. We might have an a priori idea for how program runtimes or other properties might be distributed. For example, we might specify a uniform distribution between 0 and 10 seconds for program runtime. If the generated program dataset with that target distribution leads to poor cost model performance, then the target distribution could be adjusted.

To sample a program, ProgGen samples a runtime r and schedule stage count s from the corresponding distributions. It then selects the program P_i that is closest to r and s normalized by the standard deviations of the runtime and stage count distributions, σ_r and σ_s. The following equation describes how ProgGen samples a program.

$$\arg\min_{P_i} \frac{|runtime(P_i) - r|}{\sigma_r} + \frac{|stages(P_i) - s|}{\sigma_s}$$

Sampling programs to mutate in this way makes it likely that the final distribution of runtimes and scheduling stages will match the target distributions as long as mutation operations aren't very biased in increasing or decreasing the runtime or stage count of a program. In Figure 3-2, I report the runtime distributions for program datasets generated with and without importance sampling. I show that importance sampling had a large effect on the runtime distribution and made it closely match the target distribution. In Figure 3-3, I report the schedule stage count distributions for program datasets generated with and without importance sampling. Importance sampling for schedule stage counts had a large effect on the distribution and made it more closely match the desired distribution, although it didn't match as well near the distribution's right tail. This might be because it is more difficult to generate longer programs that have many stages than it is to generate programs that run for longer.

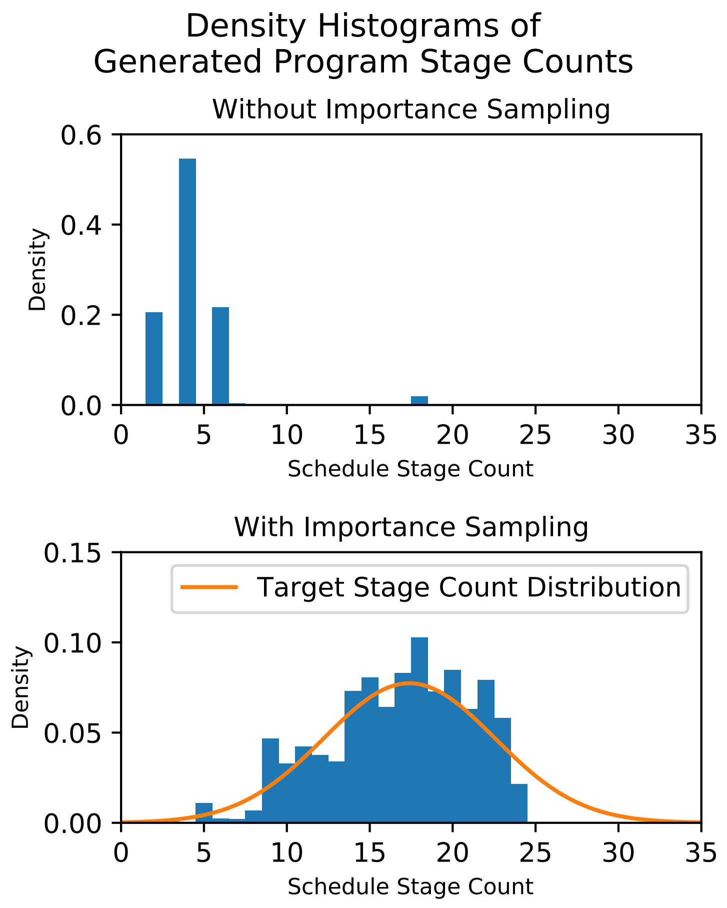


Figure 3-3: Density histograms of the stage counts of schedules ProgGen generated without using importance sampling and with using importance sampling. The schedule stage count refers to the number of computation stages in the generated algorithm. The graph with importance sampling also displays the probability density function that was used for importance sampling. Using importance sampling significantly changes the stage count distribution and makes it more closely match the target distribution. However, the match isn't as good as in the case of the runtime distribution matching, particularly near the right tail of the distribution.

Importance sampling is therefore effective at ensuring that short and easy to mutate programs don't out compete the rest of the programs. Importance sampling allows ProgGen to create programs that have a variety of runtimes and scheduling stage counts.

3.3.2 Mutation Operations

Once a program is selected to mutate, ProgGen uniformly randomly selects one of its four mutation operations to perform on that program: node replacement, node deletion, node insertion, and sub tree replacement. The mutations operate on the AST of the program and use the Halide grammar to ensure that mutations are syntactically valid according to the rules of the grammar. The mutation operations along with examples are described in detail next.

Node Replacement Node replacements choose a random node in the AST and change that node with another type of node that could fit there according to grammar rules. For example, it might change an 'Addition' node to a 'Multiplication' node, change a 'sin' function to a 'sqrt' function, or change a variable or number to a different variable or number. If the node's type can't be changed without breaking a grammar rule, then a new node in the AST is chosen to be replaced.

$$Example : a + b \rightarrow a * b$$

Node Deletion Node deletions attempt to make the AST smaller by choosing a node and replacing it with one of its child nodes. For example, it might choose to delete an 'Addition' node and replace that node with its left child node. If a chosen node can't be deleted then a new node is chosen to be deleted. If no nodes can be deleted without breaking grammar rules, then a new program is sampled.

$$Example : a + b \rightarrow a$$

Node Insertion A node insertion attempts to grow the AST by performing the inverse of a node deletion. It chooses a node and attempts to insert a new node above it in the tree and generates new sibling nodes. For example, it might choose a variable node to insert above. It might then insert a multiplication above that variable node and generate a new variable node as a sibling to the selected node. It chooses a variable either from the set of variables already in the program, or it generates a new variable name to use.

$$Example : a + b \to a + (b * c)$$

Sub Tree Replacement A sub tree replacement removes an entire sub tree from the AST and generates a new sub tree to replace it of a random size that is valid according to the grammar rules. The sub tree generation is biased towards producing terminal nodes so that the sub tree generation process is more likely to terminate. If the sub tree generation doesn't terminate after 20 nodes, it attempts to generate another sub tree up to 10 times. If it fails to generate a replacement sub tree 10 times, then a new program is sampled.

$$Example : a + b \to a + (c - d + e)$$

When mutations replace or add terminals like numbers or variables, 90% of the time it chooses an existing terminal of the same type in the AST to use, and 10% of the time it uses a new number in the range of 0 to 100 or chooses a new variable name. For example, 90% of the time a node replacement would replace a variable with another variable that occurs in the AST.

Mutation Distance

Figure 3-4 shows the number of mutations each program in a generated dataset is from one of the seed programs used to start the generation process. On average, generated programs were 25.2 mutations away from a seed program. Some of these mutations are relatively small node deletions, insertions, or replacements, but some are entire sub tree replacement mutations. Therefore, two trees that are only 25 mutations away from each other can look very different.

Additionally, because Halide is designed to express image and neural network computations in a concise way, small changes to a program can result in radically different computations performed. Small changes can make programs written in general

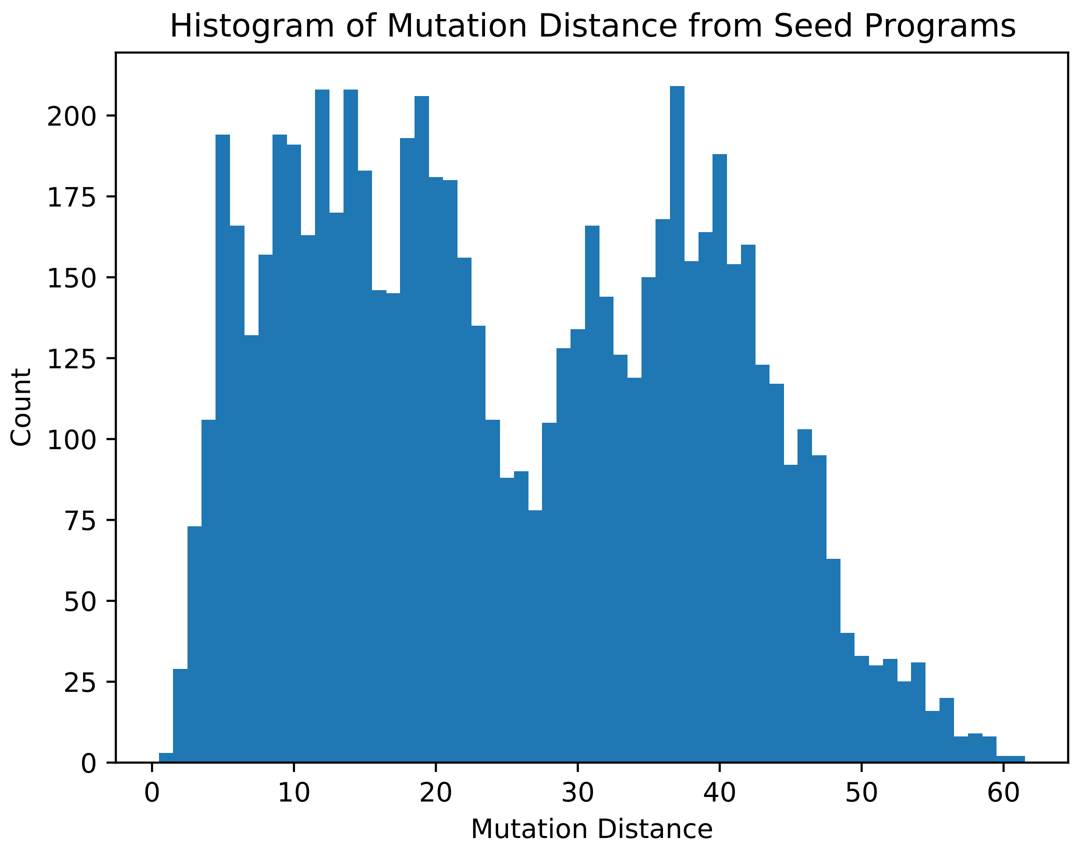


Figure 3-4: This figure shows the histogram for the number of mutations that each program is from a program in the general seed programs seed dataset. 7000 programs were generated from this seed dataset. No program was more than 62 mutations away from an program in the seed dataset. On average, a generated program was 25.2 mutations away from a seed program.

purpose languages perform very different computations, and this is even more so true in a domain specific language like Halide. Programs that are 25 mutations apart can produce output images that look very different.

3.3.3 Benchmarking and Validity Checks

Mutated programs are guaranteed to be correct according to the rules of the grammar, meaning that a compiler should be able to parse it without syntax errors. However, because ProgGen uses a context free grammar, the mutated program might not be correct according to semantic rules of the language. For example, it might use a variable before it is declared or might call a function with a variable type that isn't allowed by the type system.

Because of this, ProgGen tries to compile the mutated program. If it fails to compile, then the mutated program is thrown out, and ProgGen samples a new program to mutate. If the program does compile, then ProgGen performs additional validity checks to ensure that vacuous programs aren't generated. ProgGen checks that each program gives the same output when given the same input multiple times. This helps ensure that programs aren't returning results based on undefined or random behavior. ProgGen also checks that each program gives different outputs when different inputs are given to the program. This ensures that programs perform some computation and don't just return a fixed output.

Validity Requirements

For a program to be considered as valid by ProgGen, it must:

- Parse successfully according to the grammar.
- Compile successfully by a compiler.
- Run without throwing an error.
- Give the same output when ran with the same input multiple times.
- Give different outputs when ran with different inputs.

If these validity checks pass, then ProgGen runs the program and measures its runtime and number of scheduling stages. If the program takes longer than 40 seconds to run, then it is thrown out to prevent getting stuck running programs that don't terminate or run for a very long time. Finally, ProgGen adds the mutated program into the program population along with its runtime and schedule count. Then ProgGen repeats the program generation cycle by sampling a new program to mutate.

3.4 Program Crossover

Crossover is an operation in genetic algorithms that takes two individuals/chromosomes/solutions and combines them in some way to produce a new individual/chromosome/solution. In the case of genetic programming, a crossover operation would correspond to combining two programs in some way to produce a new program. Crossover is intended to either introduce more variety beyond what mutations introduce when generating a new generation, or to sometimes combine the best features in two individuals to create a more fit individual [7].

Crossovers for programs can often lead to invalid programs unless the crossover mechanism has significant knowledge of language semantics. Some genetic programming algorithms work around this by using more flexible languages like certain dialects of Lisp that are more likely to be valid provided the syntax of language is correct [9]. However, ProgGen uses Halide for the evaluations in this thesis, and Halide has many semantic rules that must be followed that aren't encoded in the syntax (the grammar rules) of the language. Additionally, ProgGen is designed to be able to generate programs in any language by simply using a different language grammar, so it would make ProgGen much less flexible if I were to encode knowledge of particular Halide semantic rules into the crossover mechanism for ProgGen. For these reasons, Halide crossovers that ProgGen produces are very unlikely to be correct. As a result, I tried to incorporate a learnt crossover repair mechanism into ProgGen so that ProgGen could learn to repair crossovers in any language that it generates programs in.

3.4.1 Crossover and Crossover Repair Mechanism

I attempted to implement the crossover and crossover repair mechanism in the following way. First, alongside the four mutation operations in the program generation loop of ProgGen, I added a crossover operation. This crossover operation sampled two programs from the current program population. Then, it selected a sub tree randomly from the first sampled program and selected another random sub tree from the second sampled program. It then replaced the sub tree in the first program with the sub tree selected from the second program. The output is then a new program that looks like the first program except a sub tree replaced with a sub tree from the second program. This produced a crossover-ed program that was almost always semantically invalid.

Because the program produced by crossover is almost always semantically invalid, I implemented a learnt crossover repair mechanism. This mechanism is based off of a Tree-to-Tree LSTM for program translation that uses attention in an encoder-decoder framework for trees [4]. The idea is that program translation is somewhat similar to program repair, so a neural network architecture that works well for translation might work well for repair. The Tree-to-Tree LSTM encodes the program that was produced by the crossover and then uses that encoding to, hopefully, produce a correct version of that program. In order to tokenize the program tree for ingestion into the neural network, each non-terminal and terminal in the program AST is turned into a token. Terminal symbols that have Regex matchers (and so can represent a vast, or even infinite, number of symbols), like variable names and numbers are tokenized based on the order in which they appear in the program AST. For example, if the variable 'foo' occurs first, then it would be assigned the first variable token identifier. If the variable 'bar' occurs later, it would be assigned the second variable identifier. The crossover mechanism allowed up to 15 unique variables or numbers for the evaluations in this section.

To train the learnt crossover repair neural network, I used the invalid mutations produced by ProgGen's program generation loop. Each invalid mutation represents an invalid program AST, and the program before the invalid mutation represents a valid program AST. Each training example is an invalid AST from after an invalid mutation and a valid AST from before the mutation. The neural network is trained to go from the invalid AST to the valid AST. Teacher forcing is used during the training process. Teacher forcing is a technique used during the training of neural networks that uses ground truth as the input from the previous step rather than the model output. ProgGen continuously generates a larger and larger program dataset as it trains the program repair mechanism. At some point, the program repair mechanism hopefully becomes accurate enough to succeed in repairing crossover-ed programs.

3.4.2 Crossover Repair Results

To test the crossover repair mechanism, I initially instructed ProgGen to produce simple C programs using a small subset of the C grammar shown in Listing 3.2. ProgGen generated a dataset of C programs and trained the crossover repair mechanism alongside its generation process and eventually was able to successfully crossover C programs and repair them. These repaired crossover programs were also included in the program population. In order to test the efficacy of the crossover repair, I wrote a probabilistic grammar for the simplified subset of C that was used and generated 10,000 simplified C programs.

I then performed a crossover operation on each of these 10,000 programs with another program from the set of 10,000 and used the trained crossover repair mechanism to repair the crossover-ed programs. It was able to successfully repair 71% of the crossover-ed programs. See Section 3.3.3 for the definition of a valid program and successful repair. Each repaired program had on average only 3.8 changed nodes in its AST after being repaired compared to before being repaired. Therefore, the repair mechanism wasn't significantly changing the entire program.

```
body: statement | statement body
statement: for_statement | if_statement | assignment
     | declaration | assignment_declaration
for_statement: "for" "(" assignment_declaration ";" expr ";"
   assignment ")" "{" body "}"
if_statement: "if" "(" expr")" "{" body "}"
assignment: NAME "=" expr";"
declaration: TYPE NAME ";"
assignment_declaration: TYPE NAME "=" expr ";"


expr: sum
```

```
        | expr "==" sum
        | expr ">"  sum
        | expr "<"  sum
sum: product
        | sum "+" product
        | sum "-" product
product: atom
        | product "*" atom
        | product "/" atom
atom: NUMBER
        | "-" atom
        | NAME
        | "(" expr ")"

TYPE: "int" | "bool"
NAME: /[a-zA-Z_][a-zA-Z0-9]*/ // Variable Name Regex
NUMBER: /\d+(\.\d*)?(f)?/ // Integer or floating point Regex
```

Listing 3.2: This Listing shows the grammar used for the simplified C like language used for testing the crossover repair mechanism. See Listing B.1 in the Appendix for a description of the syntax used for the grammar.

After testing the crossover repair mechanism using the simplified subset of C, I tested it using Halide. I found that once trained for long enough, the crossover repair mechanism was able to repair 40% of crossover-ed Halide programs. I then generated two program datasets using ProgGen with crossover and without crossover starting from the image seed dataset. I then trained Halide cost models using these two generated program datasets and measured the Halide Autoscheduler speedups when using these cost models. I report the results in Figure 3-5. I found that including crossover led to a program dataset that resulted in better downstream autoscheduler performance when used to train the Halide cost model.

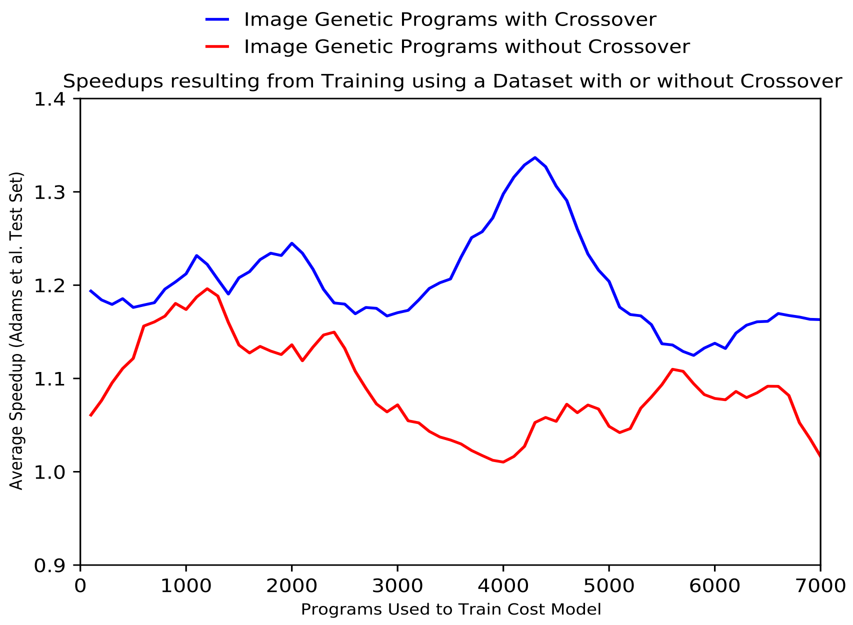


Figure 3-5: This figure shows the downstream speedups from using the Halide Autoscheduler in beam mode with a cost model trained using programs generated by ProgGen using crossover, or a cost model trained using programs generated by ProgGen without using crossover. The image seed dataset was used to seed ProgGen in both cases. I found that crossover had a small advantage over not using crossover. Importance sampling was not used for the generation of these datasets as crossover does not work well with importance sampling because programs become to large to successfully repair.

However, the results above were done without importance sampling. I found that once I included importance sampling in ProgGen, the crossover repair mechanism's repair accuracy went to 0% (although not its token level accuracy). I think this is because importance sampling selected for larger and more complex programs. For example, the average schedule stage count of the generated program datasets went from 4 to 16 when not using importance sampling vs using importance sampling. Since importance sampling led to much larger downstream autoscheduler speedups compared to including crossover, I opted to not include crossover when performing the main evaluations for this thesis.

3.4.3 Discussion and Future Work on Crossover Repair

Including program crossover and crossover repair led to better downstream autoscheduler performance. For simple C programs and small Halide programs, the crossover repair performed well. However, the crossover repair performed very poorly for large Halide programs. A future program generator might benefit from a more robust crossover repair mechanism that can repair larger and more complex programs. There are a few ways this might be done.

First, an important point to note is that although the crossover repair mechanism is being trained to fix invalid mutations, it's being used to fix invalid crossovers. An improved crossover repair mechanism might involve finding a way to train that more closely resembles crossover repair. Second, a larger Tree-to-Tree LSTM with more layers and parameters combined with more training data might allow it to successfully repair larger programs. Third, other neural architectures might perform better. In particular, rather than using an encoder-decoder framework, a repair mechanism might output patches to be performed at particular locations. That way it doesn't have to reproduce the entire program tree. Fourth, there might be a way to incorporate learning into the crossover process such that its more likely to produce correct crossovers in the first place that don't need to be repaired. Finally, even if only small programs can be successfully crossover-ed, it might still be worth performing crossovers on only small programs and ignoring crossovers for large programs.

Chapter 4

Evaluation

I first describe the datasets I use (Section 4.1) and how I evaluate ProgGen against the Halide random program generator [1] (Section 4.2). Standard cost model error metrics such as mean absolute percentage error were measured to have poor correlation with downstream Autoscheduler performance. I hypothesized that other measures of cost model error and accuracy correlate better with downstream Autoscheduler performance. Therefore, I evaluate the accuracy of the Halide Autoscheduler cost model when trained on different training datasets using different accuracy and error metrics and report which metrics correlate most strongly with downstream autoscheduler performance (Section 4.3). I show that while ProgGen's generated training datasets result in worse downstream autoscheduler performance on the Adams et al. test dataset (Section 4.4), they result in competitive downstream performance on the image processing test dataset, and slightly better downstream performance on the video processing, and linear algebra test datasets (Section 4.5). Finally, I discuss the variance we encountered in the Halide Autoscheduler system (Section 4.6).

4.1 Datasets

I use three kinds of datasets that I refer to as seed datasets, training datasets, and test datasets. Seed datasets refer to the sets of programs used to initialize ProgGen's program generation process (Section 3.2.1). Training datasets refer to the sets of

Halide program generated by ProgGen or the Halide random program generator. Test Datasets refer to the programs I measured accuracy, error, or downstream Autoscheduler performance on.

4.1.1 Seed Datasets

There are four seed datasets of 10 programs each. Tests were ran using seed datasets of 50 programs, but those tests performed similarly, so 10 was used for simplicity. Each set of seed programs are comprised of general, image processing, video processing, or linear algebra programs. Each seed dataset was used to generate one training dataset using ProgGen. The programs in each seed dataset are listed below. The programs in these seed datasets were either collected from various public GitHub repositories, from examples in the Halide repository [17], or were from programs in another language that I translated to Halide. The average length of all seed programs was 53.3 lines.

General Seed Programs Brighten filter, depth-wise convolution layer, convolution + max pool layer, average pool layer, Gaussian blur filter, inverse Daubechies wavelet filter, inverse Haar wavelet filter, sharpen filter, heat map filter, Sobel filter

Image Processing Seed Programs Adaptive contrast filter, adaptive threshold filter, bilinear scale filter, brighten filter, image dilation filter, divergence3d filter, Gaussian blur filter, RGB to greyscale filter, heat map filter, Sobel filter

Video Processing Seed Programs Video Gaussian blur filter, video brighten filter, video gamma correction filter, video horizontal flip filter, video strobe effect filter, video Sobel filter, video heatmap filter, video moving blackout filter, video speedup filter, video time reversal filter

Linear Algebra Seed Programs BLAS dcopy, BLAS ddot, BLAS dnrm2, BLAS snrm2, BLAS lnrm2, BLAS laxpy, BLAS lgemm, BLAS igemv, BLAS icopy, BLAS ldot

4.1.2 Training Datasets

I generated five training datasets for the evaluation. I generated four training datasets by running ProgGen four times with a different one of the four seed dataset each time. I refer to these datasets based on the name of the seed program they were generated from. I also ran the Halide random program generator [1] once to generate a training dataset and refer to this dataset as 'Random Programs'.

Each training dataset contains 336000 Halide schedules and their measured runtimes. Each training dataset contains 7000 unique Halide algorithms, and I used the Halide Autoscheduler schedule generation process [1] to generate 48 schedules for each algorithm. I use 7000 Halide algorithms rather than 10000 as used in the Adams et al. Halide Autoscheduler [1] because I am using 48 thread machines where it is more efficient to generate 48 schedules from each Halide algorithm, whereas they used 32 thread machines and generated 32 schedules per algorithm.

I also reserve an eighth of each training dataset to be used as a validation dataset.

4.1.3 Test Datasets

I use four test datasets that are composed of about 10 Halide algorithms each that perform computations found in real world applications. One dataset, the 'Adams et al. Test Dataset' is composed of Halide benchmarks used in the Adams et al. paper [1]. I left out a few of their benchmarks that failed to compile consistently on our machines, like a lens blur application and ResNet-50 implemented in Halide. I also constructed 3 test datasets with programs in image processing, video processing, or a linear algebra domain.

I define image processing programs as those that take as input greyscale or color images, I define a video processing programs as those that take as input a series of greyscale or color image frames, and I define linear algebra programs as those that are made up of BLAS level 1, 2, or 3 operations [11].

The names of the programs used in each test dataset can be found in Figure 4.2. No programs in the test dataset are included in the training or seed datasets.

4.2 Evaluation Methodology

I evaluate ProgGen by training the learned cost model from the Halide Autoscheduler on training datasets generated by ProgGen using different seed datasets versus the training dataset generated by the Halide random program generator (Section 4.1.2). I measure many different accuracy and error metrics of the cost models on programs in the test set. I instruct the Halide Autoscheduler to use each learned cost model to generate schedules for the Halide algorithms in the test sets. I then benchmark these schedules to get downstream performance results for each cost model trained on a different training dataset. I repeat these experiments using the Halide Autoscheduler in both beam search and greedy search mode.

I found that there is significant downstream Halide Autoscheduler performance variance depending on the cost model weight initialization and schedule training orders. Therefore, I use each training dataset to train the Halide Autoscheduler cost model 10 times; each time varying the cost model weight initialization and the order the cost model trains on the schedules. All training and evaluations are performed on a cluster of nodes that use 24 core / 48 thread Intel Ivy Bridge CPUs (Intel(R) Xeon(R) CPU E5-2695 v2) without GPU's that run on Ubuntu 18.04.

4.3 Cost Model Accuracy and Error

I found that existing metrics of cost model error on test sets, such as mean absolute percentage error, had poor correlation with downstream Halide Autoscheduler performance on test sets. I hypothesize that this is because in the Halide Autoscheduler, it doesn't matter how far off a prediction is from the true runtime as long as the cost model ranks the schedules correctly. Additionally, because the Halide Autoscheduler picks the the schedules with the best predicted time in its search path, what matters is that a cost model ranks the fastest schedules as being the fastest. It doesn't matter if slow schedules are ranked incorrectly with other slow schedules.

I proposed and evaluated several different accuracy and error metrics by measuring correlations between a cost model's accuracy/error for a given Halide program and the relative throughput for a schedule produced by that cost model in conjunction with the Halide autoscheduler. I describe the different correlation and accuracy metrics in Section 4.3.1. I describe our method for measuring the relative throughput correlations of different accuracy and error metrics in Section 4.3.2. Finally, in Section 4.3.3 I use the metric with the highest correlation with relative throughput, the 'Best 2 Prediction Error' metric, to evaluate the cost models trained with different training datasets.

4.3.1 Description of Accuracy and Error Metrics

For these definitions, I first define some notation. I define SR_i to be the schedule with the ith fastest runtime. I define SP_i to be the schedule with the ith fastest predicted runtime. N is the number of test schedules I have for a given program. The function $r(schedule)$ maps a schedule to its measured runtime and the function $pr(schedule)$ maps a schedule to its predicted runtime

Pairwise Accuracy I define pairwise accuracy as the fraction of all unique pairs of schedules for a given program that a cost model orders correctly. For example, if schedule 1 is faster than schedule 2, then this pair would only be ordered correctly if the cost model's predicted runtime of schedule 1 is faster than the predicted runtime of schedule 2.

Intuitively, if a cost model orders orders more pairs of schedules correctly, it is more likely to order the fastest schedules ahead of slower schedules. This metric performs poorly if a cost models orders most pairs correctly, but orders a slow schedules ahead of all other schedules.

$$\frac{2}{N(N-1)} \sum_{i=1}^{N-1} \sum_{j>i}^{N} pr(SR_i) < pr(SR_j)$$

Fastest Pairwise Accuracy I define fastest pairwise accuracy as the fraction of schedules that are predicted to be slower than the true fastest schedule. For example, if the fastest schedule has a runtime of 1 second, and 75% of the other schedules are predicted to have a runtime slower than 1 second and the rest predicted to be faster than 1 second, then the fastest pairwise accuracy would be 0.75.

This metric performs poorly if a cost model predicts that the fastest schedule is faster than most schedules, but predicts that a couple of slow schedules are faster.

$$\frac{1}{N-1}\sum_{i=2}^{N} pr(SR_i) > pr(SR_1)$$

Mean Absolute Percentage Error I define mean absolute percentage error as the mean percentage that each predicted runtime is off from its true runtime.

$$\frac{1}{N}\sum_{i=1}^{N}\frac{|pr(SR_i) - r(SR_i)|}{r(SR_i)}$$

Mean Squared Error I define mean squared error as the average squared percent that each predicted runtime is off from its true runtime. For example, if every schedule's runtime was predicted to be 10% slower than it actually is, then the squared percent error would be 1.0.

$$\frac{1}{N}\sum_{i=1}^{N}\frac{(pr(SR_i) - r(SR_i))^2}{r(SR_i)}$$

Ranking Error I define ranking error as the average percent error the ith fastest predicted runtime is from the ith actual fastest runtime. Note that with this metric, it doesn't matter how far off the predicted runtimes are from the true runtimes as long as the cost model correctly orders the schedules. Also note that it penalizes misorderings less if the schedules that it misorders have similar true runtimes.

$$\frac{1}{N}\sum_{i=1}^{N}\frac{|r(SP_i) - r(SR_i)|}{r(SR_i)}$$

Best K Error I define best K error as how many times slower the schedules with the best K predicted runtimes are from the true fastest runtime on average. For

Correlations of Accuracy/Error Metrics to Throughput	
Accuracy/Error Metric	Correlation Coefficient
Pairwise Accuracy	+0.43
Fastest Pairwise Accuracy	+0.25
Mean Abs. Percentage Error	−0.15
Mean Squared Error	−0.18
Ranking Error	−0.36
Best 1 Prediction Error	−0.13
Best 2 Prediction Error	−0.48
Best 5 Prediction Error	−0.45
Best 10 Prediction Error	−0.43
Best 200 Prediction Error	−0.28
Best 400 Prediction Error	−0.35

Table 4.1: This table displays the correlation of each accuracy or error metric with the relative measured throughput for all programs in the test datasets. 2000 schedules were generated from each program in all four test datasets, and then I measured each accuracy or error metric for those schedules for each cost model. I then used each cost model with the Halide Autoscheduler to generate a schedule for each program and measured its throughput and normalized it by dividing it by the throughput of the fastest known schedule for that program. For each accuracy metric, I then measured the Pearson correlation coefficient for all the cost model accuracy and measured throughput pairs.

example, if the fastest schedule has a runtime of 2 seconds, and the schedules with the 2 fastest predicted runtimes have actual runtimes of 4 and 6 seconds, then the best 2 error would be 2.5.

The intuition behind this metric is that what matters most for a cost model is that when presented with many schedule choices, the cost model predicts that the fastest schedule, or a schedule with a similar runtime to the fastest, is the fastest. It doesn't matter if the cost model misorders all the slow schedules, or even if the cost model mispredicts runtimes by orders of magnitude, as long as its able to tell the autoscheduler which of the possible schedules is fastest. This metric measures how close to the fastest schedule the best predicted schedules are, which should correspond well with how the autoscheduler works.

$$\frac{1}{K}\sum_{i=1}^{K}\frac{r(SP_i)}{r(SR_1)}$$

4.3.2 Measuring Accuracy and Error Correlations

Good accuracy and error metrics allow us to evaluate cost models without using the Halide Autoscheduler to schedule and run programs in the test datasets, which can be time consuming. Because I have many proposed accuracy and error metrics, I want to know which metric correlates most with downstream runtimes of schedules that the Halide autoscheduler produces when using a given cost model.

Our process for measuring these correlations went as follows. First, I generated 2000 schedules for each program in the test datasets using the Halide Autoscheduler with a high exploration setting. I generated many schedules with a high exploration parameter so that I have large schedule diversity. This matters because I expect the Halide Autoscheduler to examine a large variety of schedules during its search process. Then, for each of the 50 cost models I trained (10 cost models trained for each training dataset), I measured an accuracy/error metric of that cost model on each set of 2000 schedules. Then, I used that cost model in conjunction with the Halide autoscheduler in beam mode to generate a schedule for each program. I then measured the throughput of each schedule and divided it by the highest known throughput for that schedule to normalize it.

This gave us 39 accuracy/error and relative throughput pairs for each cost model (1 pair for each test program). This means, for each accuracy/error metric, I had 1950 accuracy/error and normalized throughput pairs. I then measured the Pearson correlation coefficient for the 1950 pairs for each accuracy/error metric. I reported the correlation coefficients in Figure 4.1.

4.3.3 'Best 2 Prediction Error' of Cost Models

I found that the best 2 prediction error of a cost model had the highest correlation with the runtimes of schedules generated by the Halide Autoscheduler with that cost model. I hypothesize that best 1 prediction error had poor correlations with downstream results because the signal it provides is far too noisy. There's no guarantee that the autoscheduler will find all of the schedules in the test set, or there won't be other

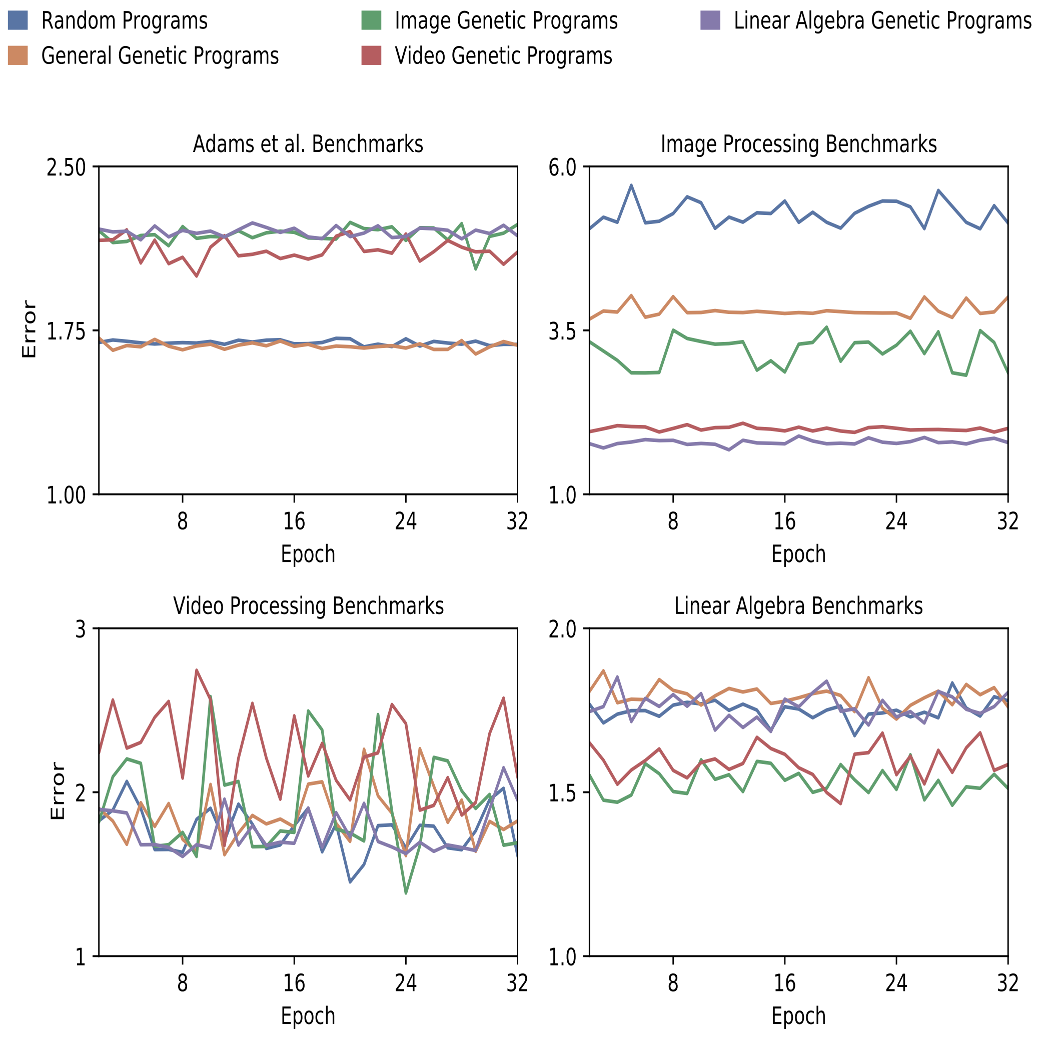


Figure 4-1: Average 'best 2 prediction error' of each cost model during training for each of the test datasets. Each cost model was trained for 32 epochs across all of its training data. At the end of each epoch, I measured the 'best 2 prediction error' for the 2000 schedules generated for each program in each test dataset. I then averaged the error across each program in a test dataset to get the error for that test dataset. The 'best 2 prediction error' seems to be relatively flat during the training duration. Although not plotted, other measures of accuracy and error show a similar trend.

schedules it finds when performing its search. Therefore best K prediction error performs better when it averages the prediction error for more than 1 of the fastest predicted schedules because the signal it provides is less noisy.

I report the best 2 prediction error on test sets from each epoch of training, averaged across each of the 10 cost models trained on each of the five training datasets

Autoscheduler (Beam Mode) Speedups					
Test Set	Random Programs	General Genetic Programs	Image Genetic Programs	Video Genetic Programs	Linear Algebra Genetic Programs
Adams et al.	1.46 ± 0.06	1.18 ± 0.06	1.07 ± 0.05	0.84 ± 0.07	0.70 ± 0.06
Image	2.88 ± 0.07	2.83 ± 0.10	2.20 ± 0.10	1.93 ± 0.12	1.77 ± 0.09
Video	0.85 ± 0.05	0.77 ± 0.05	0.92 ± 0.03	0.84 ± 0.04	0.63 ± 0.03
Linear Algebra	4.26 ± 0.21	3.15 ± 0.28	4.68 ± 0.26	3.60 ± 0.46	3.36 ± 0.22

Table 4.2: This table displays the speedup (higher is better) of the Adams et al. Halide Autoscheduler when used in beam search mode compared to the classic Halide Autoscheduler [14] when using a cost model trained on either programs from the Halide random program generator, or when trained on programs generated by ProgGen seeded with real programs from a general, linear algebra, image, or video seed dataset. It shows speedups for four different test datasets: the Halide benchmarks used in the Adams et al. paper [1], or benchmarks I put together in an image, video, or linear algebra domain. To the right of each speedup, I also show the standard error.

in Figure 4-1. I find that the test errors for each of the test sets don't decrease during training regardless of the training dataset that was used to train the cost model. This is in contrast to the errors on the validation and training datasets which do decrease during training. We're not sure why this would be the case, but one explanation could be that the cost models are over-fitting to their training datasets.

4.4 Autoscheduler Performance on Adams et al. Test Set

Autoscheduler (Greedy Mode) Speedups					
Test Set	Random Programs	General Genetic Programs	Image Genetic Programs	Video Genetic Programs	Linear Algebra Genetic Programs
Adams et al.	1.06 ± 0.04	0.82 ± 0.04	0.74 ± 0.05	0.65 ± 0.03	0.59 ± 0.05
Image	1.93 ± 0.05	1.86 ± 0.09	1.45 ± 0.11	1.51 ± 0.05	1.22 ± 0.12
Video	0.78 ± 0.04	0.74 ± 0.05	0.88 ± 0.03	0.86 ± 0.05	0.57 ± 0.03
Linear Algebra	3.49 ± 0.27	2.74 ± 0.23	4.36 ± 0.33	3.42 ± 0.45	3.50 ± 0.37

Table 4.3: This table displays the same information as Table 4.2, except the Halide Autoscheduler is used in greedy search mode instead of beam search mode.

In addition to measuring various accuracy and error metrics for the cost models, I also measured downstream autoscheduler performance using each of the cost models I trained. I ran the Halide Autoscheduler in both greedy mode, where it greedily searches the schedule space, and in beam mode, where it performs a beam search of

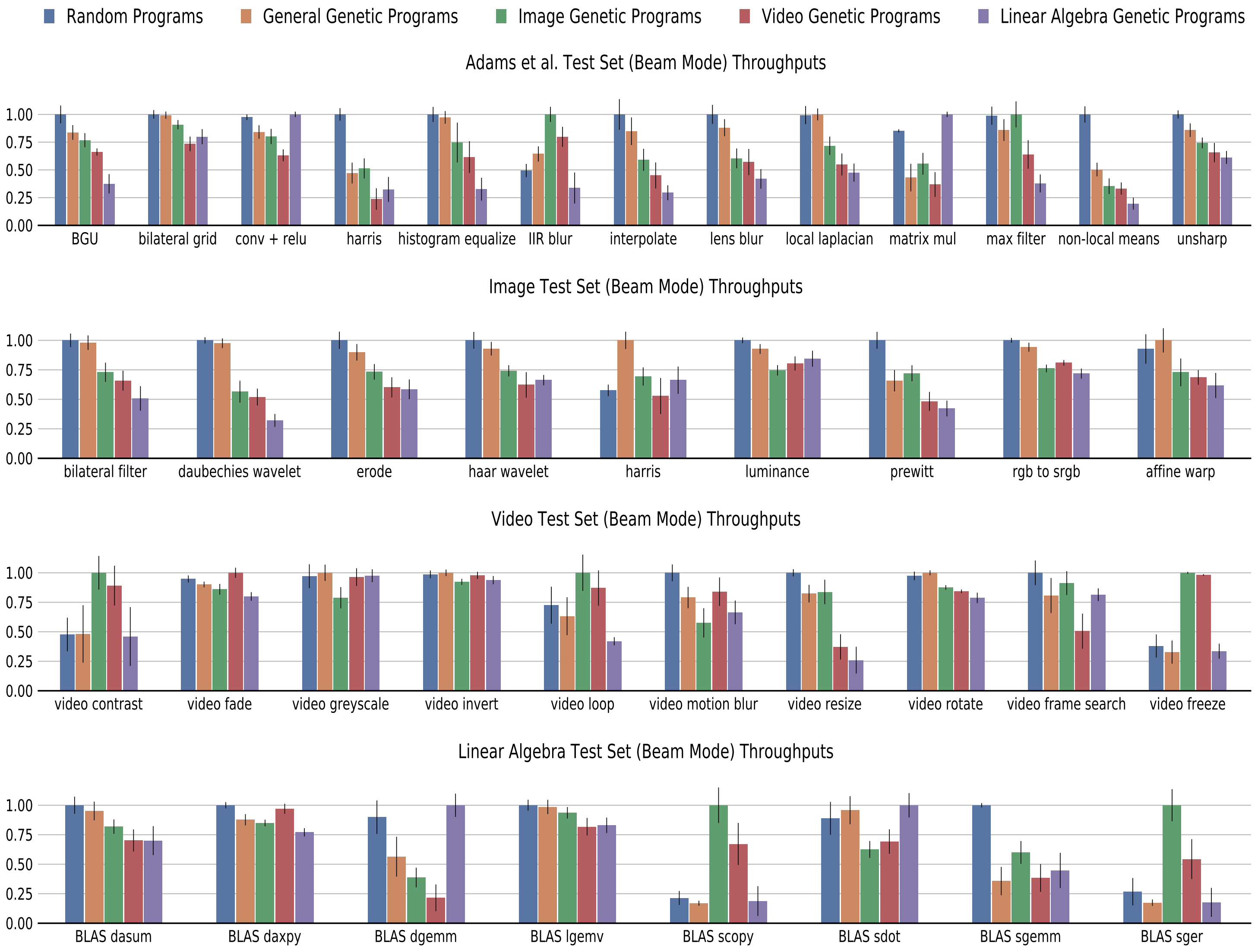


Figure 4-2: Throughput of Halide schedules generated by the Halide Autoscheduler using different cost models where the best throughput is normalized to 1 for each program in four different test datasets. The image, video, and linear algebra test datasets include Halide programs that perform operations on images, videos, and tensors respectively. The Adams et al. test set includes programs they used as benchmarks. Each Halide program is autoscheduled using a cost model trained on either programs from the Halide random program generator, or from programs generated with ProgGen seeded with 10 programs from a general, image, video, or linear algebra seed dataset. The autoscheduler was used in beam search mode for all cases. The Halide random program generator produces a training dataset that results in on average 24% faster performance than the next best performing training dataset on the Adams et al. benchmarks. On average the cost models trained on our image training dataset perform 10% and 8% better on the video and linear algebra test datasets respectively compared to the cost models trained on the Halide random program generator training dataset.

the schedule space with multiple passes. I also generated and measured the runtimes of schedules for programs in test datasets using the classic Halide Autoscheduler [14], the autoscheduler in place before the Adams et al. autoscheduler [1] that doesn't use a learned cost model.

I report speedups using beam mode relative to the classic Autoscheduler in Figure 4.2, and speedups using greedy mode in Figure 4.3. The first row in the table includes geometric means of the speedups of all programs in that test dataset relative to the classic autoscheduler. Each program in each test dataset was autoscheduled a total of 50 times, 10 times for each retrained cost model for each of the 5 training datasets generated. Beam mode program specific results from each test program are shown in Figure 4-2.

For the Adams et al. test dataset, I found that using the Halide Autoscheduler with a cost model trained on programs from their random program generator resulted on average in schedules that ran 24% faster than our best performing cost model. The training datasets generated by ProgGen using the general and image seed datasets resulted in similar downstream performance to each other. The training datasets ProgGen generated using video or linear algebra seed datasets performed much worse on the Adams et al. test dataset. The speedups when using the Halide Autoscheduler in greedy mode are generally lower but show the same trends amongst the different training datasets.

4.5 Autoscheduler Performance on Other Test Sets

Generally, program generators are over-fitted manually to their domain. For instance, the Halide random program generator [1] includes local convolutions, max pools, histogram operators, and other patterns that resemble some of the local patterns found in their benchmarks. Because of this, I hypothesized ProgGen would perform relatively better in computation domains that their random program generator wasn't designed for.

Results for each test dataset when using the Halide Autoscheduler in beam mode with a cost model trained on one of the five training datasets is shown in Figure 4.2. Similar results using the Halide Autoscheduler in greedy mode are shown in Figure 4.3. Beam mode program specific results from each test set are shown in Figure 4-2.

I found that the training dataset ProgGen generated using the general seed dataset performed best on the image processing test set. When using a cost model trained on that training dataset, it resulted in only 2% worse average downstream performance when using beam mode compared to a cost model trained with programs from the Halide random program generator. This difference was also within a single standard error.

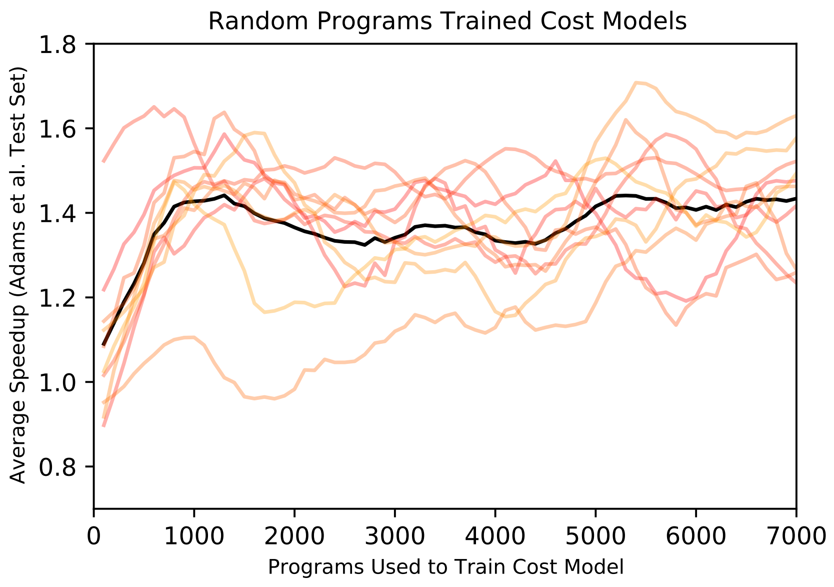
For the video and linear algebra test datasets, the cost model trained on the training dataset ProgGen generated using the image seed dataset performed best. In beam mode, it resulted in 8% and 10% better average downstream performance for the video processing and linear algebra test datasets, respectively, compared to a cost model trained with programs from the Halide random program generator. In greedy mode, it resulted in 13% and 25% better performance for the video processing and linear algebra test datasets, respectively. This shows ProgGen can be competitive with or slightly better than the Adams et al. random program generator [1] in image processing, video processing, and linear algebra domains. However, ProgGen generates program datasets automatically without having deep knowledge of Halide built in. It just requires a language grammar and seed programs to generate programs of a particular programming language.

4.6 Halide Autoscheduler Variance

I found that the Halide Autoscheduler and Halide Cost Model training pipeline had a large amount of variance. Small changes in the training of the cost model could lead to very different downstream speedup results for the Autoscheduler. This variance was far larger than benchmarking variance, which was less than 1% because benchmarks could easily and quickly be repeated many times to get an accurate average. In Figure 4-3, I shuffled the random programs and image genetic programs training datasets 10 times each and trained a cost model on the first 100, first 200, first 300, and so on up until the first 7000 programs of the shuffled dataset. Figure 4-3 shows that the variance of downstream performance is quite large for cost models trained on either the random programs dataset from the Halide Random Program generator or the image genetic programs dataset from ProgGen.

I found it odd that there was a large variance on downstream performance if the only change was the order in which the cost model trained on the programs in the training set. I was unable to find out why exactly this might be the case. However, I have a few hypothesis. First, the training datasets might not be diverse or large enough and the cost model is over-fitting to its training sets. The validation accuracies and training losses had far smaller variance than the end to end downstream results. Second, the cost model is a relatively simple neural network with only 2 fully connected layers. Perhaps the model is too simple to really be able to accurately predict runtimes and has a lot of variance when moving to new domains as a result. Finally, the Autoscheduler system might just be chaotic such that small changes in the cost model can lead to large changes in autoscheduler performance, i.e. the kinds of schedules that are generated.

To account for this variance, I made sure to retrain each cost model 10 times and take averages and measure variances when reporting results.



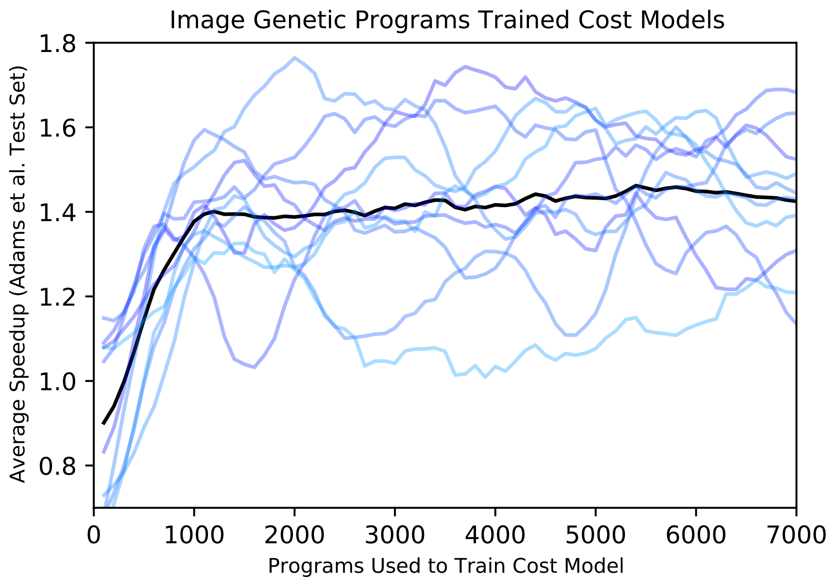


Figure 4-3: These graphs show the downstream Halide Autoscheduler speedups in beam search mode on the Adams et al. test set for cost models trained with either the random programs or image genetic programs training sets. For each graph, 700 cost models were trained and evaluated. For the upper graph, the random programs training dataset was shuffled randomly 10 times, then a cost model was trained using the first 100 programs from a shuffled set, then the first 200, then the first 300, and so on until a cost model was trained using all 7000 programs from the shuffled training set. Therefore, the 10 cost models trained using 7000 programs were trained with the same 7000 programs, but trained on those programs in a different order. The lower graph was generated in the same way but using the image genetic programs training dataset. Each coloured line corresponds to a cost model trained with the first 'x' number of programs from a shuffled dataset. The black lines are the averages of the coloured lines.

Chapter 5

Related Work

Program Dataset Generation In order to train cost models for compilers, program datasets are needed. In some cases like Ithemal [13] and AutoTVM [3], a large collection of real world programs are able to be collected to form a program dataset. In the case of the Halide Autoscheduler [1] and the Tiramisu Compiler [2], cost models are trained using a program dataset generated by a hand written program generator. These program generators both work by piecing together random expressions in particular formats and produce programs that are correct by construction. ProgGen differs from these systems because it is generalized to use a program grammar to allow it to easily generate programs in other domains, and because it uses techniques inspired by genetic programming.

A system used for translating programs from one language to another using a Tree-to-Tree LSTM generated training data using a probabilistic grammar for relatively simple Javascript and Coffeescript programs [4]. However, when testing the system on real world C# and Java programs, they collected their training data by crawling open source projects that had both a C# and Java implementation.

Genetic Programming Genetic programming has been used for generating test cases for programs [23, 19, 22, 26, 15, 20]. In these systems, test cases are often evolutionary evolved by performing mutation and crossover operations on program inputs and saving inputs that cause program assertions to fail. Systems have been designed to use genetic programming to automatically repair broken programs using geneting programming [16]. Genetic programming has also been used for program synthesis [8, 24, 25, 5, 10, 6]. ProgGen differs from program repair and synthesis because it is generating an entire training dataset of programs. Additionally, ProgGen is not optimizing towards a specific program specification or solution.

Stoke and MCMC Search In some ways, ProgGen is similar to Stoke [21]. Stoke uses MCMC search to find optimized programs in a search space. In its MCMC search, it is constantly mutating a program to try and find the the most optimized version of that program. Although ProgGen also mutates programs, ProgGen doesn't have an objective function that it attempts to minimize. Additionally, the program generation loop performed by ProgGen isn't described well by MCMC because ProgGen stores all programs it generates and can mutate any generated program to generate new candidates.

Chapter 6

Conclusion

6.1 Limitations

ProgGen requires writing a language grammar in EBNF form for the target language and collecting a set of seed programs in that target language. Although this is intended to take far less time than engineering a hand crafted program generator, it is still a limitation. ProgGen also requires selecting program features, like runtime and schedule stage count, and a distribution of those features to be sampled from in the importance sampling step. These distributions might be known a priori by an expert, but they might take trial and error to find good distributions otherwise. ProgGen also requires programs it generates to specify their own inputs, or at least the shape and dimensions of their inputs.

ProgGen doesn't have a good way of measuring its generated program dataset's diversity or quality without training a downstream system and measuring the downstream performance. ProgGen's crossover mechanism also only worked for small Halide programs and programs written in a small subset of C. Additionally, although ProgGen is designed to take less engineering time to produce programs, it takes more computation time to produce programs. For example, it takes ProgGen about 24 hours to generate 7000 valid Halide programs. Though, I didn't spend significant effort optimizing the speed of ProgGen.

6.2 Future Work

ProgGen could be extended to automatically determine distributions of program features to use in its importance sampling. It could do this by analyzing the distributions of program features in its seed dataset and fitting a distribution automatically to them. ProgGen's crossover mechanism that I attempted to implement could be improved so that it can successfully crossover and repair large Halide programs, or large programs in any language. The mutation operations of ProgGen could be improved, perhaps even by including learned mutation operations that increase the chance of successful mutations.

A set of measurements could be created that measure the quality or diversity of a program dataset, at least relative to other program datasets. I evaluated the quality of a program dataset by using it to train the Halide cost model and measuring the downstream autoscheduler performance. However, other measures of program dataset quality would have allowed me to iterate and test ProgGen much more quickly. Additionally, measures of program similarity and diversity could be used to improve ProgGen's generation process.

6.3 Conclusion

I present ProgGen, a program generator inspired by genetic programming for automatically generating program datasets used in training compiler cost models. ProgGen generates program datasets that can lead to better downstream performance in some computation domains than traditional program generators. ProgGen demonstrates that future program generation systems can be made more general and automatic by utilizing genetic programming inspired techniques.

Appendix A

Additional Figures and Tables

Correlations of Accuracy/Error Metrics to Throughput (Greedy Mode)	
Accuracy/Error Metric	Correlation Coefficient
Pairwise Accuracy	+0.33
Fastest Pairwise Accuracy	+0.16
Mean Abs. Percentage Error	−0.11
Mean Squared Error	−0.13
Ranking Error	−0.36
Best 1 Prediction Error	−0.15
Best 2 Prediction Error	−0.41
Best 5 Prediction Error	−0.42
Best 10 Prediction Error	−0.41
Best 200 Prediction Error	−0.34
Best 400 Prediction Error	−0.40

Table A.1: This table displays the same information as Table 4.1, except the throughput values were measured using the Halide Autoscheduler in greedy search mode as opposed to beam search mode. 2000 schedules were generated from each program in all four test datasets, and then I measured each accuracy or error metric for those schedules for each cost model. I then used each cost model with the Halide Autoscheduler in greedy mode to generate a schedule for each program and measured its throughput and normalized it by dividing it by the throughput of the fastest known schedule for that program. For each accuracy metric, I then measured the Pearson correlation coefficient for all the cost model accuracy and measured throughput pairs.

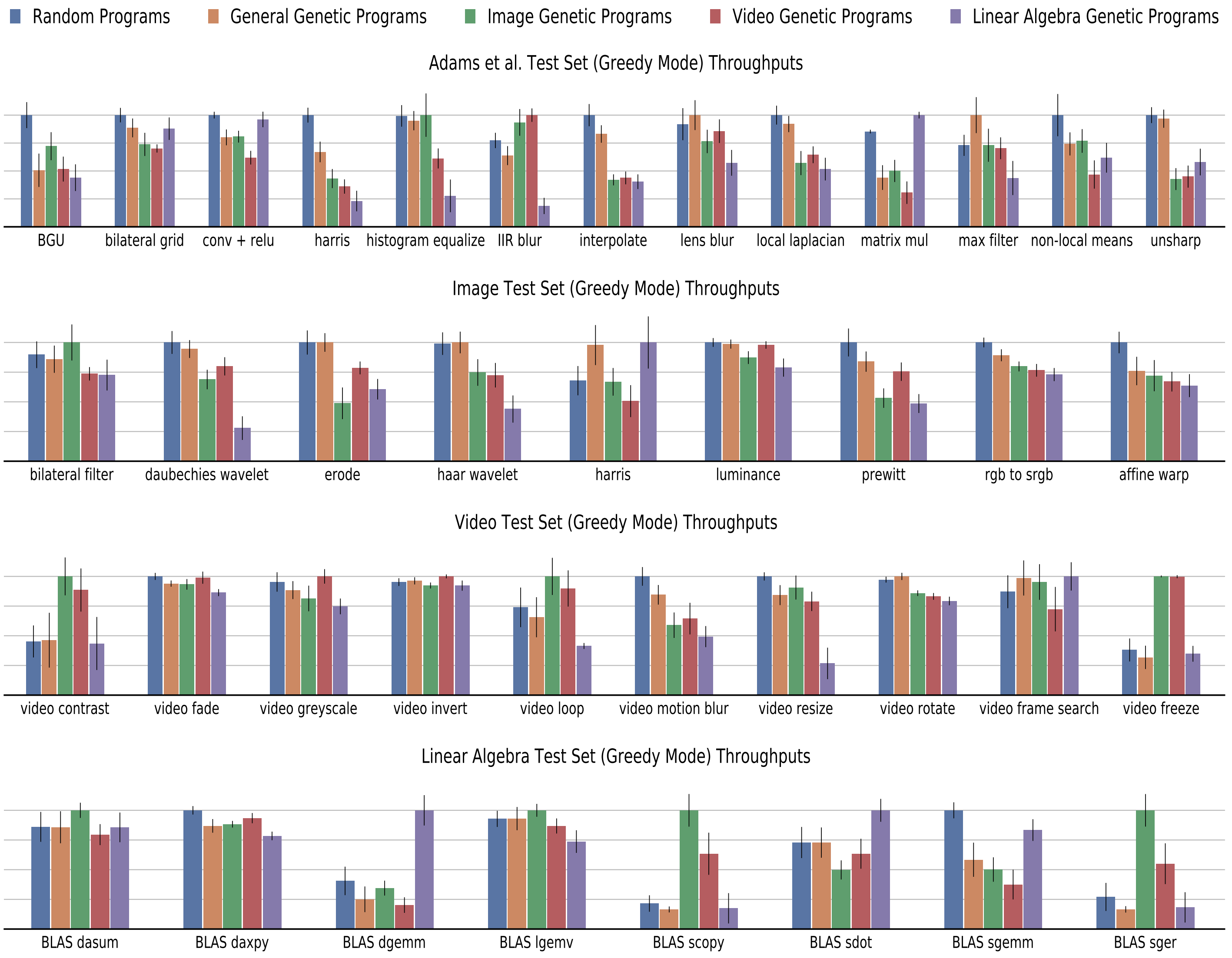


Figure A-1: Throughput of Halide schedules generated by the Halide Autoscheduler using different cost models where the best throughput is normalized to 1 for each program in four different test datasets. The image, video, and linear algebra test datasets include Halide programs that perform operations on images, videos, and tensors respectively. The Adams et al. test set includes programs they used as benchmarks. Each Halide program is autoscheduled using a cost model trained on either programs from the Halide random program generator, or from programs generated with ProgGen seeded with 10 programs from a general, image, video, or linear algebra seed dataset. The autoscheduler was used in greedy search mode for all cases. The Halide random program generator produces a training dataset that results in on average 24% faster performance than the next best performing training dataset on the Adams et al. benchmarks. On average the cost models trained on our image training dataset perform 10% and 8% better on the video and linear algebra test datasets respectively compared to the cost models trained on the Halide random program generator training dataset.

Appendix B

Halide Grammar

```
start: inputs "void generate() {" body
inputs:
    "Input<Buffer<" PLAIN_TYPE ">> input{'input', 3};"
    "Input<Buffer<" PLAIN_TYPE ">> filter{'filter', 3};"
    "Output<Buffer<" PLAIN_TYPE ">> output{'output', 3};"
body: statement | statement body
statement: for_statement
    | if_statement
    | assignment ";"
    | declaration
    | assignment_declaration
    | call_statement
    | declaration_constructor

assignment: NAME ASSIGN_TYPE expr | function_call
  ASSIGN_TYPE expr
declaration: type name_list ";"
declaration_constructor: type NAME "(" arg_list ");"
assignment_declaration: type NAME "=" expr ";"
```

```
if_statement: "if" "(" expr ")" "{" body "}"
    | "if" "(" expr ")" "{" body "}" "else" "{" body "}"
    | "if" "(" expr ")" "{" body "}" "else" if_statement

call_statement: function_call ";"
for_statement: "for" "(" assignment_declaration expr ";"
   assignment ")" "{" body "}"
expr: sum
    | expr "==" sum
    | expr "!=" sum
    | expr ">" sum
    | expr ">=" sum
    | expr "<" sum
    | expr "<=" sum
sum: product
    | sum "+" product
    | sum "-" product
    | sum ">>" product
    | sum "<<" product
product: atom
    | product "*" atom
    | product "/" atom
    | product "%" atom
atom: NUMBER
     | UNARY atom
     | NAME
     | function_call
     | structure_constructor
     | "(" expr ")"
```

```
ASSIGN_TYPE: "=" | "+=" | "-=" | "*=" | "/="
UNARY: /-/ | "!"

type: PLAIN_TYPE | casted_type
casted_type: PLAIN_TYPE "<" PLAIN_TYPE ">"
PLAIN_TYPE: "int" | "bool" | "RDom" | "Buffer" | "Func" |
    "Var" | "Expr" | "float" | "long" | "double"

structure_constructor: "{" arg_list "}"

function_call: halide_function | var_call | method_call
method_call: NAME "." method

halide_function: cast | min | max | ceil | floor | round
   | abs | sum_func | pow | fast_exp | exp | fast_log |
   sqrt | sin | tanh | cos | undef | repeat_edge

cast: "cast" "<" PLAIN_TYPE ">" "(" expr ")"
min: "min" "(" arg_list ")"
max: "max" "(" arg_list ")"
ceil: "ceil" "(" expr ")"
floor: "floor" "(" expr ")"
round: "round" "(" expr ")"
abs: "abs" "(" expr ")"
sum_func: "sum" "(" expr ")"
pow: "pow" "(" expr "," expr ")"
fast_exp: "fast_exp" "(" expr ")"
exp: "exp" "(" expr ")"
fast_log: "fast_log" "(" expr ")"
```

```
sqrt: "sqrt" "(" expr ")"
sin: "sin" "(" expr ")"
tanh: "tanh" "(" expr ")"
cos: "cos" "(" expr ")"
undef: "undef" "<" PLAIN_TYPE ">" "(" ")"
repeat_edge: "BoundaryConditions::repeat_edge(" NAME ")"

method: halide_method |  halide_method "." method
halide_method: WIDTH | HEIGHT | X | Y | Z

WIDTH: "width()"
HEIGHT: "height()"
X: "x"
Y: "y"
Z: "z"

var_call: NAME "(" arg_list ")" | NAME "()"
arg_list: expr | expr "," arg_list
name_list: NAME | NAME "," name_list

NAME: /[a-zA-Z_][a-zA-Z0-9]*/ // Variable Name Regex
NUMBER: /\d+(\.\d*)?(f)?/ // Integer or floating point
    Regex


%import common.WS_INLINE // Commands to ignore whitespace
%ignore WS_INLINE
%import common.WS
%ignore WS
```

```
COMMENT: "//" /[^\\n]*/ // Regex for comments
%ignore COMMENT // Command to ignore comments

PREAMBLE:
   "#include "Halide.h"
    #include <iostream>
    #include <iomanip>
    #include <random>
    #include <cstdlib>
    #include <unordered_map>

    using namespace Halide;
    using namespace Halide::Internal;

    class RandomPipeline : public Halide::Generator<
       RandomPipeline> {
    public:"

POSTAMBLE:
  "if (auto_schedule) {
      int dims[3] = {2000, 2000, 3};
      int filter_dims[3] = {7, 7, 3};

      for (int i=0; i < input.dimensions(); i++) {
          input.dim(i).set_bounds_estimate(0, dims[i]);
      }

      for (int i=0; i < filter.dimensions(); i++) {
          filter.dim(i).set_bounds_estimate(0,
             filter_dims[i]);
```

```
        }

        for (int i=0; i < output.dimensions(); i++) {
            output.estimate(output.args()[i], 0, dims[i]);
            output.dim(i).set_bounds_estimate(0, dims[i]);
        }
    }}};"
```

Listing B.1: Full Halide Grammar written using the Lark package from Python. Terminals are in all caps; non-terminals are all lower-cased. Literal string values are surrounded by double quotation marks. Regex is surrounded by forward slashes. A grammar rule consists of a terminal or non-terminal on the left, followed by a colon, then the expansion on the right. A vertical bar in an expansion ("|") represents an or operation. For example, "a: B | C" would represent a grammar rule that says non-terminal a can be expanded into terminal B or terminal C. The PREAMBLE and POSTAMBLE terminals contain a fairly large amount of literal text that acts a preamble or postamble to set up the Halide programs used. Comments in the grammar are denoted by a double forward slash ("//"). The Lark Python package contains a few directives that begin with percent symbols ("%") that are used.

Bibliography

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-kelley. Learning to optimize halide with tree search and random programs. In *Association for Computing Machinery Transactions on Graphics*, 2019.

[2] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. A deep learning based cost model for automatic code optimization. In *Machine Learning and Systems*, 2021.

[3] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Conference on Neural Information Processing Systems*, 2018.

[4] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *International Conference on Learning Representations*, 2018.

[5] Stefan Forstenlechner, David Fagan, Miguel Nicolau, and Michael O'Neill. Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In *Institute of Electrical and Electronics Engineers Congress on Evolutionary Computation*, 2018.

[6] Thomas Helmuth. General program synthesis from examples using genetic programming with parent selection based on random lexicographic orderings of test cases. 2015.

[7] Colin G. Johnson. Genetic programming crossover: Does it cross over? In *European Conference on Genetic Programming*, 2009.

[8] Gal Katz and Doron Peled. Synthesis of parametric programs using genetic programming and model checking. In *International Workshop on Verification of Infinite-State Systems*, 2014.

[9] John R. Koza and James P. Rice. Automatic programming of robots using genetic programming. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[10] Krzysztof Krawiec, Iwo Błądek, Jerry Swan, and John H. Drake. Counterexample-driven genetic programming: stochastic synthesis of provably correct programs. In *International Joint Conference on Artificial Intelligence*, 2018.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. In *Association for Computing Machinery Transactions on Mathematical Software*, 1979.

[12] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. In *Association for Computing Machinery Transactions on Graphics*, 2018.

[13] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, 2019.

[14] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. In *Association for Computing Machinery Transactions on Graphics*, 2016.

[15] M. Nosrati, H. Haghighi, and M. Vahidi Asl. Test data generation using genetic programming. In *Information and Software Technology*, 2021.

[16] Vinicius Paulo L. Oliveira, Eduardo F. D. Souza, Claire Le Goues, and Celso G. Camilo-Junior. Improved crossover operators for genetic programming for program repair. In *International Symposium on Search Based Software Engineering*, 2016.

[17] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Association for Computing Machinery SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[18] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from 'big code'. In *Association for Computing Machinery Principles of Programming Languages*, 2015.

[19] J. C. B. Ribeiro, A. F. Nogueira, F. F. de Vega, and M. A. Zenha-Rela. ecrash: a genetic programming-based testing tool for object-oriented software, 2015.

[20] Vahid Raf Sajad Esfandyari. A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy. In *Information and Software Technology*, 2018.

[21] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

[22] Akshat Sharma, Rishon Patani, and Ashish Aggarwal. Software testing using genetic algorithms. In *International Journal of Computer Science and Engineering Survey*, 2016.

[23] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm, 2014.

[24] Thomas Weise and Ke Tang. Evolving distributed algorithms with genetic programming. In *Institute of Electrical and Electronics Engineers Transactions on Evolutionary Computation*, 2011.

[25] Thomas Weise, Mingxu Wan, Ke Tang, and Xin Yao. Evolving exact integer algorithms with genetic programming. In *Institute of Electrical and Electronics Engineers Congress on Evolutionary Computation*, 2014.

[26] Xiangjuan Yao and Dunwei Gong. Genetic algorithm-based test data generation for multiple paths via individual sharing. In *Computational Intelligence and Neuroscience*, 2014.