

IMPROVING STABILITY AND PARAMETER SELECTION OF DATA  
PROCESSING PROGRAMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Wen-Chuan Lee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

**THE PURDUE UNIVERSITY GRADUATE SCHOOL**  
**STATEMENT OF DISSERTATION APPROVAL**

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Zhiyuan Li

Department of Computer Science

Dr. Tiark Rompf

Department of Computer Science

Dr. Yexiang Xue

Department of Computer Science

**Approved by:**

Dr. Clifton W. Bingham

Department of Computer Science

*Dedicated to my family for their love and support*

## ACKNOWLEDGMENTS

This dissertation would not have been possible without the support from many people in my life. First of all, I would like to express my sincerest gratitude to my advisor Professor Xiangyu Zhang for his support, patience, and listening. I really appreciate that he gave me a chance to do research with him. By working with him, I have learned how to discover a research problem, how to build a system to verify ideas, and how to take broad, high-level ideas and to be able to focus on those ideas in a more nuanced, focused way.

I would also like to thank the members of my committee: Professor Zhiyuan Li, Professor Tiark Rompf, and Professor Yexiang Xue. Their feedback about my dissertation and research has made my work significantly stronger. It was also a pleasure to be a member of an awesome research group lead by Professor Xiangyu Zhang. I thank them for their input and support of my work. Their feedback about my ideas has made my work better.

Finally, I am immensely grateful to my family, especially my wife, Yi-Shan Lin, who encouraged me to pursue the Ph.D. degree and provide her unconditional love through the whole process. I acknowledge my parents for their endless patience and understanding. The life lessons they taught me and what I have learned during my Ph.D. journey will be in my heart forever. I dedicate this dissertation to my family.

## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	ix
LIST OF FIGURES . . . . .	x
ABSTRACT . . . . .	xiii
1 INTRODUCTION . . . . .	1
1.1 Research Challenges . . . . .	2
1.2 Dissertation Statement . . . . .	4
1.3 Contributions . . . . .	4
1.4 Dissertation Organization . . . . .	6
2 RAIVE: RUNTIME ASSESSMENT OF FLOATING-POINT INSTABIL- ITY BY VECTORIZATION . . . . .	7
2.1 Introduction . . . . .	7
2.2 Background . . . . .	11
2.3 Vectorization and RAIVE Overview . . . . .	13
2.4 Design of RAIVE Runtime . . . . .	21
2.4.1 Semantics . . . . .	22
2.4.2 Understanding the Essence of RAIVE . . . . .	30
2.5 Evaluation . . . . .	31
2.5.1 Performance . . . . .	34
2.5.2 Effectiveness . . . . .	35
2.5.3 Case Studies . . . . .	39
2.6 Summary . . . . .	43
3 WHITE-BOX PROGRAM TUNING . . . . .	44
3.1 Introduction . . . . .	44
3.1.1 Key Observation of Staged Computation Paradigm . . . . .	46

	Page
3.1.2 Existing Work . . . . .	46
3.1.3 Our Work . . . . .	46
3.1.4 Properties . . . . .	47
3.1.5 Contributions . . . . .	48
3.2 Overview of White-Box Tuning Framework . . . . .	49
3.2.1 User Interface . . . . .	49
3.2.2 Running Example . . . . .	50
3.2.3 Runtime Execution Model . . . . .	51
3.2.4 Result and Comparison . . . . .	53
3.3 Execution Model: Semantics and System . . . . .	55
3.3.1 Semantics . . . . .	55
3.3.2 WBTUNER Runtime System . . . . .	58
3.4 Practical Challenges . . . . .	60
3.4.1 Overfitting . . . . .	60
3.4.2 Incremental Aggregation . . . . .	62
3.4.3 Sampling/Aggregation Strategies . . . . .	62
3.4.4 Auto-tuning Sampling Number . . . . .	63
3.5 Evaluation . . . . .	64
3.5.1 Tuning Results Summary . . . . .	65
3.5.2 Tuning Case Studies . . . . .	67
3.6 Summary . . . . .	78
4 PROGRAMMING SUPPORT FOR AUTONOMIZING SOFTWARE . . . . .	79
4.1 Introduction . . . . .	79
4.1.1 Autonomization: Bringing the Intelligence to Traditional Programs	80
4.1.2 Problems and Challenges . . . . .	81
4.1.3 Our Design . . . . .	83
4.2 Autonomization Framework Overview . . . . .	84
4.3 Execution Model: Semantics . . . . .	90

	Page
4.3.1	Definitions . . . . . 92
4.3.2	Rules . . . . . 93
4.4	Feature Variables Extraction . . . . . 95
4.5	Implementation . . . . . 99
4.6	Evaluation . . . . . 100
4.6.1	Statistics . . . . . 102
4.6.2	Effectiveness . . . . . 104
4.6.3	Case Studies . . . . . 107
4.7	Summary . . . . . 113
5	SPSA: StATISTICAL AND PROGRAM ANALYSIS AIDED SOFTWARE AUTONOMIZATION . . . . . 114
5.1	Introduction . . . . . 114
5.1.1	Existing Software Autonomization . . . . . 116
5.1.2	Our Work . . . . . 117
5.1.3	Contributions . . . . . 118
5.2	Motivation . . . . . 119
5.3	Design . . . . . 126
5.3.1	Overview . . . . . 126
5.3.2	Automatic Feature Variables Selection . . . . . 126
5.4	Evaluation . . . . . 130
5.4.1	Statistics . . . . . 130
5.4.2	Effectiveness . . . . . 132
5.4.3	Case Study . . . . . 134
5.5	Summary . . . . . 143
6	RELATED WORK . . . . . 144
6.1	Floating Point Instability . . . . . 144
6.2	Program Parameter Configuration . . . . . 145
7	CONCLUSION . . . . . 148

REFERENCES . . . . . 150

## LIST OF TABLES

Table	Page
2.1 Performance (o/h stands for overhead). . . . .	33
2.2 Instability detection. . . . .	37
2.3 Average number of unstable predicates, and forks for an execution, and output variations across samples. RSD stands for relative standard deviation.	38
3.1 Benchmark statistics and the experiment results for achieving the best tuning scores. . . . .	63
4.1 Program analysis statistics . . . . .	101
4.2 Model statistics . . . . .	101
4.3 Benchmark experimental results. . . . .	104
5.1 Statistics of models and feature variables . . . . .	131
5.2 Benchmark experimental results. . . . .	135

## LIST OF FIGURES

Figure	Page
2.1 Inevitable External Errors . . . . .	9
2.2 Floating Point Representation. . . . .	11
2.3 (Column 1) example program with <code>max()</code> inlined; (Column 2) actual execution with each entry denoting actual computed value; (Column 3) ideal execution; (Column 4) TAG [11] execution; and (Column 5) RAIVE execution. $r^F$ in the TAG approach means value $r$ is tagged with a false error bit. The shaded sub-execution denotes the new execution after the user manually annotates the benign unstable predicate (line 10). . . . .	15
2.4 Boxed statements correspond to instrumentation. Note that in RAIVE, the original floating point related statements are completely replaced by vector statements. Labels 3.1-3.5 denote instrumentation for line 3. . . . .	18
2.5 Language . . . . .	22
2.6 Operational Semantics. . . . .	23
2.7 An example for nesting unstable predicates. Symbol $r$ represents a large floating point value. Assume the input value is $r + 1$ which cannot be precisely represented and hence the represented value is $r$ at line 2. Other large values can be precisely represented. . . . .	28
2.8 The control flow graph for the example in Fig. 2.7. . . . .	29
2.9 Pseudocode snippet for k-means. . . . .	40
2.10 Clustering result variations of an unstable execution for k-means; 92 genes are grouped into five clusters; each cluster has a unique color. . . . .	40
2.11 Pseudocode snippet for pagerank. . . . .	41
2.12 Benign unstable predicate in 187.facerec. . . . .	42
3.1 Canny's results with different parameters . . . . .	45
3.2 Execution models of black-box and white-box tuning . . . . .	47
3.3 Primitives . . . . .	49
3.4 White-box tuning for Canny. The highlighted statements are added. Tuning primitives start with <code>wbt</code> . . . . .	51

Figure	Page
3.5 Execution Model . . . . .	52
3.6 Tuning Canny. TP/SP are tuning/sampling processes. . . . .	54
3.7 Tuning Canny with image coffeemaker in 90s. . . . .	54
3.8 Operational Semantics . . . . .	55
3.9 Tuning + Validation Execution Model . . . . .	61
3.10 Optimization effects on different benchmarks . . . . .	67
3.11 Canny tuning scores of 10 images. . . . .	68
3.12 Canny tuning score variation . . . . .	68
3.13 Canny results of WBTUNER and OpenTuner . . . . .	69
3.14 White-box tuning for phylip tree . . . . .	70
3.15 Phylip tree tuning scores on 10 datasets. . . . .	70
3.16 Phylip tree tuning score variation . . . . .	71
3.17 SVM tuning scores of 10 datasets w/wo validation . . . . .	72
3.18 SVM tuning scores of 10 datasets . . . . .	73
3.19 SVM tuning score variation . . . . .	73
3.20 Sphinx tuning of 10 datasets . . . . .	74
3.21 Sphinx tuning score variation . . . . .	74
3.22 Tuning mission 1 . . . . .	77
3.23 Tuning mission 2 . . . . .	77
3.24 Testing mission . . . . .	77
4.1 Primitives . . . . .	85
4.2 Autonomizing Mario. The highlighted statements are added. AUTONOMIZER primitives start with <b>au</b> . . . . .	87
4.3 Execution model . . . . .	88
4.4 Using internal data . . . . .	89
4.5 Using raw data . . . . .	89
4.6 Coverage testing . . . . .	91
4.7 Bug . . . . .	91

Figure	Page
4.8 Operational Semantics . . . . .	92
4.9 Alg.1 on Canny . . . . .	99
4.10 Alg.2 on Mario . . . . .	99
4.11 Canny. Autonomizing with the <i>Min</i> version. The highlighted statements are added. . . . .	109
4.12 Canny predictions of 10 datasets . . . . .	110
4.13 Canny prediction score variation . . . . .	110
4.14 Canny results . . . . .	110
4.15 EucDict $\approx 0$ . . . . .	111
4.16 Variance $\approx 0.007$ . . . . .	112
4.17 Driving score . . . . .	112
5.1 Time cost – Input Size . . . . .	117
5.2 Primitives . . . . .	120
5.3 Autonomizing Canny. The highlighted statements are provided by SPSA. The primitives start with <b>au</b> . . . . .	121
5.4 Execution Model . . . . .	123
5.5 Execution Result . . . . .	125
5.6 Canny Results . . . . .	125
5.7 Canny prediction of 10 datasets . . . . .	134
5.8 Canny training score variation . . . . .	136
5.9 Autonomizing Watershed. The highlighted statements are provided by SPSA. The primitives start with <b>au</b> . . . . .	137
5.10 Watershed prediction of 10 datasets . . . . .	139
5.11 Watershed tuning score variation . . . . .	139
5.12 Sphinx tuning score variation . . . . .	141
5.13 Sphinx tuning score variation . . . . .	142
5.14 Improving Sphinx autonomization. The highlighted statements are added. Autonomizing primitives start with <b>au</b> . . . . .	143

## ABSTRACT

Wen-Chuan Lee Ph.D., Purdue University, May 2020. Improving Stability and Parameter Selection of Data Processing Programs. Major Professor: Xiangyu Zhang Professor.

Data-processing programs are becoming increasingly important in the Big-data era. However, two notable problems of these programs may cause sub-optimal data-processing results. On one hand, these programs contain large number of floating-point computations. Due to the limited precision of floating-point representations, errors are introduced, propagated and accumulated in series of computations, making the computation results unreliable. We call this problem as floating-point instability. On the other hand, these programs are heavily parameterized. As no universal optimal parameter configuration exists for all possible inputs, the setting of program parameters should be carefully chosen and tuned for each input. Otherwise, the result would be sub-optimal. Manual tuning is infeasible because the number of parameters and the range of each parameter value may be big.

We try to address these two challenges in this dissertation. For floating-point instability problem, we develop a novel runtime technique to capture different output variations in the presence of instability. It features the idea of transforming every floating point value to a vector of multiple values – the values added to create the vector are obtained by introducing artificial errors that are upper bounds of actual errors. The propagation of artificial errors models the propagation of actual errors. When values in vectors result in discrete execution differences (e.g., following different paths), the execution is forked to capture the resulting output variations.

For parameterized data-processing programs, we develop a white-box program tuning framework to tune the program parameter configuration for optimal data-

processing result of each program input. To further reduce the parameter configuration overhead, we propose the first general framework to inject artificial intelligence (AI) in the program, so the intelligent program is able to predict the parameter configuration for each incoming input directly. However, similar to many other ML/AI applications, the crucial challenge lies in feature selection, i.e., selection of the feature variables for predicting the target parameter specified by the users. Thus, we propose a novel approach by combining program analysis and statistical analysis for better program feature variables selection which further helps better target parameter prediction and improves the result.

## 1. INTRODUCTION

There is an increasing need of data processing programs in the Big-data era. Their complexity is also growing at an enormous pace, involving more and more floating-point computations and program parameters.

With a large number of floating-point computations, errors can easily be introduced, propagated and accumulated, potentially leading to unreliable outputs, which is called floating point instability problem. The errors introduced include those due to precision limitations in physical instruments or human efforts in acquiring the inputs, called the external errors, and those from limited representation precision, called the internal errors. Existing works use the ideal execution with infinite precision as the ground truth to reason about instability in actual execution and then try to eliminate the gap between the two executions to produce reliable results. However, this is restricted in their scope. Even if the detected gap between ideal and actual executions are eliminated (e.g., by raising the precision) the execution may nonetheless be unstable as the same differences can be easily triggered by minor perturbations (external errors) of the input.

Furthermore, because data-processing programs are parameterized, using these programs or algorithms is challenging. The reason is that user has to configure these program parameters beforehand. More importantly, the optimal configuration is mostly dependent on the specific input. Different inputs require different configurations to achieve the optimal results. For instance, the results of K-means, a popular data clustering algorithm, heavily depends upon the choice of parameter K. It specifies the number of clusters into which the user wants to partition the input data and there is no general solution for finding K.

## 1.1 Research Challenges

**Handling Floating Point Instability.** Data processing using floating point programs is essential in the emerging big data era. During program execution, errors can be introduced, propagated and accumulated, potentially leading to unreliable outputs. We call this the *floating point instability problem*. The errors introduced include those due to precision limitations in physical instruments or human efforts in acquiring the inputs, called the *external errors*, and those from limited representation precision, called the *internal errors*. Handling instability is critical because important decisions may be based on data processing results – results of computer simulations may be used to setup expensive scientific wet bench experiments; commercial decisions may be made based upon results of mining customer data etc.

Researchers have developed various techniques to address the instability problem. Static techniques such as abstract interpretation and theorem proving [1–3] were proposed to reason about the existence or the absence of instability. Interval arithmetic [4,5] and affine arithmetic [6–8] model errors as ranges or affine formulas to reason about execution stability. Program transformation was proposed to improve precision and stability [9,10]. Recently in [11], an on-the-fly predictor was proposed to detect instability. Based on the prediction result, the execution may switch to a higher precision. While mostly focusing on internal errors, existing works use the *ideal execution* with infinite precision as the oracle to reason about instability in *actual execution* and then aim to eliminate the differences between the two executions to produce reliable results. However, we argue that these approaches may be undesirably restricted in their scope. Even if the detected differences between ideal and actual executions are eliminated (e.g., by hoisting the precision) the execution may nonetheless be unstable as the same differences may be easily triggered by minor perturbations of the input (due to external errors).

In this dissertation, we propose *RAIVE*, a novel runtime based approach that addresses instability that can be triggered by internal or external errors and captures

output variations in the presence of instability. The output variations can be used as guidance for debugging the data processing program to pinpoint the predicates that cause the output variations.

**Handling Parameter Selection.** Many data-processing programs often carry the parameters that affect the quality of the results. However, different inputs require different configurations to achieve the ideal results, i.e., no parameter configuration universally applies. Therefore, the users need to manually configure the parameters, which is difficult for normal users due to the great domain expertise required and sometimes even difficult for the experts if the parameter value space is huge.

Multiple frameworks were proposed to automate program parameter configuration, among which OpenTuner [12] is the state-of-the-art. Oblivious of the staged computation paradigm, these frameworks treat the computation as a black-box. Guided by a user-provided scoring function of the final result, they sample the parameter space to find the best parameter configuration. Internally, they adopt stochastic algorithms [13, 14] or genetic algorithms [15] as the search strategy. While the above frameworks have achieved a certain level of success, they suffer greatly from poor performance due to the *inherent* limitations of the *black-box* designs:

- Black-box tuning is not aware of the *stagedcomputingparadigm*, i.e., programs normally consist of multiple computation stages such that each stage has a unique set of tunable parameters. Thus, *all* parameters need to be tuned and set in each configuration, leading to an *exponential* number of configurations.
- A *full* execution accounts for the sampling of a *single* parameter configuration. Note that the full execution typically needs to load a large corpus of data and conduct lengthy preprocessing, which are very time consuming.

In this dissertation, we develop several techniques to achieve better and faster data processing for parameterized programs. First, we develop a program tuning framework *WBTuner* to tune the program parameter configuration for optimal data-processing result of each program input. It treats the program as a white box and

it is aware of the *stagedcomputationparadigm* and tunes each stage independently. Thus, *WBTuner* needs to sample much fewer parameter configurations than black-box tuning frameworks. Furthermore, *WBTuner* can easily achieve better tuning results by inspecting the program internal states.

To further reduce the parameter configuration overhead, we propose the second work *Autonomizer*. It is the first general framework that allows the users to autonomize their software systems by injecting artificial intelligence (AI) in the program. The intelligent program is able to predict the parameter configuration for each incoming input on-the-fly.

However, similar to many other ML/AI applications, the crucial challenge lies in feature selection, i.e., selection of the feature variables for predicting the target parameter specified by the users. *Autonomizer* only adopts program analysis and simple heuristics to address this problem. Thus, we propose the third technique *SPSA* to further improve the feature variables selection of program autonomization. *SPSA* leverages both program analysis and statistical analysis.

## 1.2 Dissertation Statement

In this dissertation, we aim to improve the data processing results from two different perspectives: 1) program stability and 2) program parameter selection.

The thesis of this dissertation is as follows: *Data-processing program instability can be detected by leveraging program vectorization and the the parameter selection can be improved by leveraging white-box program tuning as well as artificial intelligence.*

## 1.3 Contributions

The contributions of this dissertation are as follows:

- We propose *RAIVE*, a novel vectorization based approach that addresses program instability problem that can be triggered by internal or external errors

and captures output variations in the presence of instability. Our evaluation shows that it can precisely capture output variations and its overhead (340%) is 2.43 times lower than the state of the art.

- We propose a novel white-box tuning framework *WBTuner*. It allows users to tune their program parameters inside different program computation stages. It offers the users flexible access to internal program states. Wasteful computation caused by poor internal results in an early stage can be terminated, which is infeasible in current works. Compared with the state of the art, its tuning overhead is 3.08X lower under a single core environment and 4.67X lower when multiple cores are used.
- To achieve on-the-fly parameter selection, we propose a general framework *Autonomizer* to autonomize traditional software programs, which applies to the parameter configuration of parameterized programs, the action selection of interactive programs and many other potential applications. The evaluation shows that for the data-processing programs, *Autonomizer* improves the output quality by 161% on average over the default settings. For the interactive programs such as game/driving, *Autonomizer* achieves higher success rate with lower training time than existing autonomized programs.
- We further improve software autonomization by improving the feature selection mechanism. We propose *SPSA*, which leverages both program analysis and statistical analysis to find a better set of program feature variables to predict target variable (parameter). The evaluation shows that *SPSA* substantially improves data processing results by improving the feature variables selection on ten widely used parameterized programs. The output quality is improved by 99.04% on average over the baseline with execution overhead almost the same as *Autonomizer*. Comparatively, *Autonomizer* only improved the output quality by 80.24% on average over the baseline. Furthermore, the model training

overhead of *SPSA* is 27.44X lower than *Autonomizer* and its model size is 603.5X smaller than *Autonomizer* on average.

## 1.4 Dissertation Organization

This dissertation is organized as follows: Chapter 2 discusses the design, implementation and evaluation of RAIVE, which is a novel data-processing program stability improvement approach. Then there are three chapters to gradually discuss how we improve program parameter selection for better data peocessing results. Chapter 3 discusses WBTuner, which is a programming framework that allows users to compose complex tuning tasks to tune their program parameters inside different computation stages. Then a general software autonomization technique for more efficient parameter selection is presented in Chapter 4 and an approach to further improve the software autonomization is discussed in Chapter 5. Chapter 6 discusses the related works. Last, Chapter 7 concludes the dissertation.

## 2. RAIVE: RUNTIME ASSESSMENT OF FLOATING-POINT INSTABILITY BY VECTORIZATION

Floating point representation has limited precision and inputs to floating point programs may also have errors. Consequently, during execution, errors are introduced, propagated, and accumulated, leading to unreliable outputs. We call this the *instability problem*. In this chapter, we propose RAIVE, a technique improves data processing programs stability by identifying *output variations* of a floating point execution in the presence of instability. RAIVE transforms every floating point value to a vector of multiple values – the values added to create the vector are obtained by introducing artificial errors that are upper bounds of actual errors. The propagation of artificial errors models the propagation of actual errors. When values in vectors result in discrete execution differences (e.g., following different paths), the execution is forked to capture the resulting output variations. Our evaluation shows that RAIVE can precisely capture output variations. Its overhead (340%) is 2.43 times lower than the state of the art.

### 2.1 Introduction

Data processing using floating point programs is essential in the emerging big data era. During program execution, errors can be introduced, propagated and accumulated, potentially leading to unreliable outputs. We call this the *floating point instability problem*. The errors introduced include those due to precision limitations in physical instruments or human efforts in acquiring the inputs, called the *external errors*, and those from limited representation precision, called the *internal errors*. Handling instability is critical because important decisions may be based on data processing results – results of computer simulations may be used to setup expen-

sive scientific wet bench experiments; commercial decisions may be made based upon results of mining customer data etc. Evidence suggests that widely used data processing programs suffer from instability. We will show in Section 5.4 that a widely used implementation of the *k-means* data mining algorithm [16] can produce completely different clustering results; an information retrieval program *pagerank* [17] may produce completely different rankings, both due to the instability problem.

Researchers have developed various techniques to address the instability problem. Static techniques such as abstract interpretation and theorem proving [1–3] were proposed to reason about the existence or the absence of instability. Interval arithmetic [4, 5] and affine arithmetic [6–8] model errors as ranges or affine formulas to reason about execution stability. Program transformation was proposed to improve precision and stability [9, 10]. Recently in [11], an on-the-fly predictor was proposed to detect instability. Based on the prediction result, the execution may switch to a higher precision.

While mostly focusing on internal errors, existing works use the *ideal execution* with infinite precision as the oracle to reason about instability in *actual execution* and then aim to eliminate the differences between the two executions to produce reliable results. However, we argue that these approaches may be undesirably restricted in their scope. Even if the detected differences between ideal and actual executions are eliminated (e.g., by hoisting the precision) the execution may nonetheless be unstable as the same differences may be easily triggered by minor perturbations of the input (due to external errors). Consider the example in Fig. 2.1 with a predicate (line 3) that is unstable when the input  $x$  is close to 0.0045. The curve at the bottom shows how output  $y$  varies with input  $x$ . Observe that  $y = f(x)$  is on the left and  $y = g(x)$  on the right, and there is a discontinuity right at  $x = 0.0045$ . In contrast, the curve on the top is slightly off due to internal errors. Particularly, the discontinuity is at a value close to 0.0045. Assuming the technique in [11] reports that the unstable input range is  $r_l$  so that higher precision should be used for values in the range. Unfortunately, even use of infinite precision is insufficient as it does not alert the user

that the program output may change substantially in the presence of a small external error when input  $x$  is close to 0.0045. In other words, *instability is a property of the computation performed on a given input, which cannot be completely evaded by hoisting representation precision*. Thus we argue that achieving the same results from the ideal and actual executions ignores instability due to external errors which are inevitable in the real world.

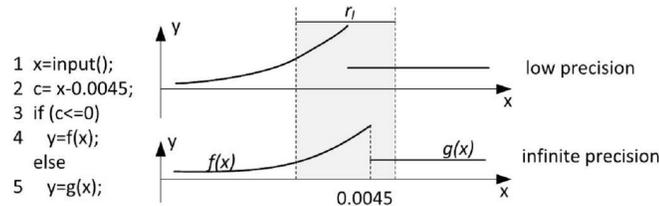


Fig. 2.1.: Inevitable External Errors

Moreover, a patch to the problem may not exist, which is different from functional bugs. In our example, the instability is due to the interface between the continuous floating point domain (i.e. variable  $c$  at line 3) and the discrete boolean domain (i.e. the branch outcome at line 3) and intrinsic to the algorithm. Changing the implementation, which is a typical method to improve stability for the floating point domain, is unlikely to completely fix the problem due to the involvement of the discrete domain.

In this paper, we observe that the key to handling runtime instabilities is not to achieve an execution close to the ideal one, but rather to *inform the user of the possible (output) effects of the instabilities* so that he/she can adjust the decision accordingly. We propose a novel technique that discloses the possible effects of errors on outputs in an actual execution for a given input  $i$ , including both internal and external errors. It does so without explicitly mutating the input  $i$ , which usually requires a large number of sample executions to expose output variations if the input is of high dimension and input correlation is complex. Instead, our technique vectorizes the subject program such that each floating point variable is represented by a vector

of multiple values. Initially, the values in a vector are identical, representing the value in the actual execution (called the *actual value*). When execution encounters operations that yield non-trivial errors, it explicitly introduces artificial errors to the actual value. The injected errors are the lower and upper bounds of the actual error (note that actual errors cannot be efficiently computed without high precision representations). Vectorization allows the mutated values to go through the same sequence of floating point operations. If they lead to discrete differences (e.g., taking different paths), the execution is forked to follow the different discrete options to capture the output variations caused by errors.

Consider the example in Fig. 2.1 . Given an input  $x_0$  within  $r_l$ , our technique detects that the predicate at line 3 is unstable. It hence forks the execution to take both branches. As such, the output variations within  $r_l$  are indicated by the differences between  $y = f(x_0)$  (from the true branch) and  $y = g(x_0)$  (from the false branch). The essence of our approach is that if internal representation errors can lead to different branch outcomes at a predicate, a small perturbation in external input is very likely to lead to the same difference. Hence we model the possible effects of external errors by modeling only internal errors.

Our key contributions include:

- We propose a novel vectorization based approach that addresses instability that can be triggered by internal or external errors and captures output variations in the presence of instability.
- We address a number of critical technical challenges, including avoiding unnecessary forks that may generate too many processes, and handling error suppressions that can properly stop error propagation.
- We develop the RAIVE (Runtime Assessment of floating point Instability by VVectorization) prototype. It precisely predicts output variations in the presence of errors, with both the precision and the recall close to 100%. The detected output changes are substantial. Its overhead is 2.43 times smaller than the

state of the art runtime instability predictor [11] that cannot predict output variations, but rather just the stability of an execution. It correctly classifies executions on over 99.99% of the inputs of the programs we studied as stable.

## 2.2 Background

**Floating Point Representation.** IEEE 754 [18] defines the format of a 64-bit floating point value as shown in Fig. 2.2. The corresponding decimal value  $f$  is given by  $f = (1 - 2s) \times (1 + m \times 2^{-52}) \times 2^{e-1023}$  where variable  $s$  is the sign bit,  $m$  the significand, which is also called mantissa, and  $e$  the exponent. There are 53 mantissa



Fig. 2.2.: Floating Point Representation.

bits, including an implicit leading bit of “1”. Any values that require more significand bits to represent cannot be precisely represented.

We use  $x$  to denote the floating point value of a variable  $x$  in the actual world (with limited precision), called the *actual value*. The corresponding value in the ideal world (with infinite precision) is called the *ideal value* and denoted as  $\hat{x}$ . The difference between the two is called the *absolute error* of  $x$ , denoted as  $\hat{\Delta}_x$ . The *relative error*  $x$ , denoted by  $\Delta_x$ , is defined as  $|\hat{\Delta}_x/x|$ . It indicates whether the actual value is reliable. Initially errors are introduced in source code constants at compile time, or when reading external inputs at runtime. The initial errors are propagated to the internal of program execution through operations.

**Discrete Factor and Instability.** A discrete factor is an operation that has floating point values as operands and produces a discrete value [19]. They are the interface between the continuous domain and the discrete domain (e.g., integer and boolean). Typical examples are control flow predicates and type casts. Discrete factors have been used to detect instability caused by representation errors [11]. If an actual

floating point value (with error) and the corresponding ideal value lead to different discrete results – one having the true branch outcome and the other the false – the execution is considered unstable. A discrete factor that yields different discrete values in the actual and the ideal worlds is called an *unstable factor* [11]. The intuition is that if we consider the output of an execution as a mathematical function over inputs, the form of the function is determined by the executed path. In the presence of unstable factors, the paths and hence the functions are different in the two worlds, indicating substantial output differences.

**Relative Error Inflation and Cancelled Bits.** Many existing works on instability detection [11,20] are built upon detecting relative error inflation, i.e., instances when the relative error of the result of a floating point operation is substantially larger than those of the operands. This is because instabilities are rooted at relative error inflation in most cases. Per IEEE 754, the result of a subtraction/addition is normalized by left-shifting to remove the leading zeros. The relative error inherited from the operands thus gets inflated. If after subtraction  $z = x - y$ , the significand bits are left-shifted by  $d$  bits, the relative error inherited from the operands,  $(\hat{\Delta}_x - \hat{\Delta}_y)/z$ , may become  $2^d$  times larger than the relative errors of  $x$  and  $y$ , because  $z$  is  $2^d$  times smaller than  $x$  or  $y$ . The left-shifted bits are also called the *cancelled bits* [20], which can be cost-effectively monitored by comparing the exponents of the result and the larger operand. In particular,  $d = \max(\varepsilon_x, \varepsilon_y) - \varepsilon_z$ , with  $\varepsilon_x$  being the exponent of  $x$ . In TAG [11], when the number of cancelled bits is larger than some pre-defined threshold  $\tau_c$ , the relative error is considered inflated. TAG further tracks the propagation of the value with an inflated relative error. An execution is considered unstable if the value can reach a discrete factor, as discrete difference (from the ideal execution), such as path difference, may be induced by the value.

### 2.3 Vectorization and RAIVE Overview

In this section, we overview RAIVE with an example and briefly explain a few important technical challenges.

Given a subject program, RAIVE leverages the compiler to transform it to a vectorized version, in which each floating point variable  $x$  is represented by a vector of four 64-bit floating point values  $\langle x_1, x_2, x_3, x_4 \rangle$ . All floating point operations are transformed to the corresponding vector operations. Initially, the vector values  $x_1, \dots, x_4$  have the same actual value of  $x$ . These values remain identical after floating point operations. A lightweight online relative error inflation detection mechanism is also inserted in the subject program. When an error inflation is detected for a variable  $x$ , the corresponding value is considered unreliable. Since we cannot precisely compute the error of  $x$ , which requires using higher precision, RAIVE computes an upper bound of the absolute error, denoted as  $\delta$ , and mutates the first two values in  $x$ 's vector, called the *left pair*, to  $x - \delta$  and  $x + \delta$ , respectively. The same perturbation is applied to the last two values, called the *right pair*. As such, the right pair is a simple duplication of the left pair. The perturbed vector further goes through floating point operations such that the value differences in  $x$ 's vector lead to value differences in other vectors, simulating the propagation of the unreliable actual value.

Upon executing a discrete factor (e.g., a predicate on floating point values), RAIVE tests if the values in the left pair lead to different discrete values (e.g., different branch outcomes) – meaning that some previously inflated relative errors have been propagated to the factor and caused discrete difference; the execution may likely become unstable. A naive solution would be to fork the execution right away so that each of the resulting executions proceeds with a unique discrete value generated at the unstable discrete factor. For instance, assuming an unstable predicate, the original execution is forked to two with one executing along the true branch and the other along the false branch. As such, the outputs generated by the forked executions denote the possible output variations in the presence of errors.

However in real-world programs, discrete differences do not necessarily lead to final output variations. For instance, even if a predicate on floating point values has different branch outcomes in the presence of errors, the two branches may have identical or highly similar effect such that the two forked executions have very little or no state differences. To address this problem, we develop a highly sophisticated runtime. Particularly, when encountering an unstable predicate, RAIVE executes both branches in sequence (in the same execution). To avoid interference between the branches (e.g. a write in one branch affects a read in the other branch, which is infeasible according to program semantics), RAIVE executes one branch on the left pairs of the vectors and the other branch on the right pairs. In other words, each variable update in a branch only modifies half of the vector. At the join point of the two branches, RAIVE compares the left and the right pairs in each vector that was updated in the branch(es). If all of them have negligible differences, the unstable predicate was benign and there is no need to fork the execution. Otherwise, RAIVE forks the execution.

**Example.** Consider an example program in the first column of Fig. 2.3. It involves both computation and decisions (i.e. lines 5, 10 and 15). All the variables are of the 64-bit double-precision type. The execution on an x86 platform is presented in the second column. The ideal execution, shown in the third column, uses infinite precision emulated via software with two to three orders magnitude of slow-down.

Initially, a very large value is assigned to  $a$  (line 1); the same large value plus one is assigned to  $b$  (line 2). In the actual execution, the increment cannot be represented and  $b$  has the same value as  $a$ . Thus, an error is introduced at line 2 in the actual execution. As such,  $c = b - a$  at line 3 has values 0 and 1 in the actual and ideal executions, respectively. Then  $c$  is used in subtraction with a large value, the result of which is further used in a conditional statement (lines 4 and 5). The error introduced earlier does not cause any problem as it is too small in comparison with the large operand value  $2^{37}$  at line 4. At line 7,  $a$  and  $b$  are passed to  $max()$ , inside which  $b - a$  is again performed. However, since the result is directly used in a conditional

	STATEMENTS	ACTUAL	IDEAL	TAG	RAIVE
1	$a = 2^{54};$	$r \dagger$	$r$	$r^F$	$\langle r, r, r, r \rangle$
2	$b = 2^{54} + 1;$	$r$	$r+1$	$r^F$	$\langle r, r, r, r \rangle$
3	$c = b - a;$	0	1	$0^T$	$\langle -64, 64, -64, 64 \rangle$
4	$d = c - 2^{37};$	$-r_1$	$-r_1+1$	$-r_1^F$	$\langle -r_1, -r_1, -r_1, -r_1 \rangle$
5	<b>if</b> ( $d \leq 0$ )	T	T	$T^F$ <b>A</b>	$\langle T, T, T, T \rangle$ <b>D</b>
6	...;				
7	$e = \max(a, b);$				
8	$\max(a, b): \{$				
9	$t = b - a;$	0	1	$1^T$	$\langle -64, 64, -64, 64 \rangle$
10	<b>if</b> ( $t \leq 0$ )	T	F	$T^T$ <b>B</b>	$\langle T, F, T, F \rangle$ <b>E</b>
11	$e = a;$	$r$		$r^F$	$\langle r, r, -, - \rangle$
12	<b>else</b> $e = b;$		$r+1$		$\langle -, -, r, r \rangle$
	<b>ret</b> $e;$				$e = \langle r, r, r, r \rangle$ <b>F</b>
	$\}$				
13	$f = e - a;$	0	1	$0^T$	$\langle -64, 64, -64, 64 \rangle$
14	$g = f - 0.5;$	-0.5	0.5	$-0.5^F$	$\langle -64.5, 63.5, -64.5, 63.5 \rangle$
15	<b>if</b> ( $g > 0$ )	F	T	$F^F$ <b>C</b>	$\langle F, T, F, T \rangle$ <b>G</b>
16	$h = 10;$		10		$\langle 10, 10, -, - \rangle$
17	<b>else</b> $h = -10$	-10		$-10^F$	$\langle -, -, -10, -10 \rangle$
					$h = \langle 10, 10, 10, 10 \rangle$ $h = \langle -10, -10, -10, -10 \rangle$ <b>H</b>
18	$o = h * 2;$	-20	20	$20^F$	$\langle 20, 20, 20, 20 \rangle$ $\langle -20, -20, -20, -20 \rangle$

† We use  $r$  to denote the large number  $2^{54}$ , or 1801439 8509481984, and  $r_1$  to denote  $2^{37}$ . Other large values are represented relative to  $r$  and  $r_1$ .

Fig. 2.3.: (Column 1) example program with  $\max()$  inlined; (Column 2) actual execution with each entry denoting actual computed value; (Column 3) ideal execution; (Column 4) TAG [11] execution; and (Column 5) RAIVE execution.  $r^F$  in the TAG approach means value  $r$  is tagged with a false error bit. The shaded sub-execution denotes the new execution after the user manually annotates the benign unstable predicate (line 10).

(line 10), the error manifests itself by yielding different branch outcomes in the two executions, leading to different return values, which are used in the subtractions with  $a$  (line 13) and then with a constant value 0.5. The resulting different values cause different branch outcomes at line 15, and eventually different outputs.

*Working of RAIVE.* Initially, the vector of variable  $a$  holds four identical 64-bit value  $2^{54}$ . At line 2, the floating point addition is transformed to a vector addition. Due to the same precision limitation as the non-vectorized actual execution (in the second column),  $b$ 's vector holds four identical values. Inside box  $\textcircled{D}$ , the subtraction at line 3 causes relative error inflation (Section 2.2). In particular, the result of the operation is 0 whereas the operands are very large values. RAIVE detects the inflation such that it introduces artificial errors in the vector that denote the bounds of the absolute errors. The first value -64 is a lower bound; the second value 64 denotes an upper bound. The right pair is a simple duplication of the left pair. We will discuss how the bounds are computed in Section 2.4.1. At line 4, the vector of  $d$  becomes four identical values despite the differences in  $c$ . In other words, the errors are *suppressed* by the subtraction with the large operand  $2^{37}$  (due to the precision limitation of the operation). Therefore, the four values of  $d$ 's vector yield the same branch outcome (at line 5), correctly modeling the fact that both the actual (column 2) and the ideal (column 3) executions take the same branch.

In box  $\textcircled{E}$  inside the function  $\text{max}()$ , the same relative error inflation is observed at line 9. However, since the variable  $t$  with inflated error is directly used in the predicate at line 10, different branch outcomes are observed. The predicate is an unstable discrete factor. Observe that in this case, the actual and ideal executions do differ at line 10. RAIVE does not fork the execution at this point as the instability may be benign. Instead, inside box  $\textcircled{F}$ , it first executes the true branch with a vector mask that enables only the left pairs of vectors. As a result, the left pair of  $e$  is assigned the left pair of  $a$ , which holds two identical values of  $2^{54}$  (line 11). After that, it further executes the false branch with a mask enabling the right pairs. Hence, the right pair of  $e$  is assigned the right pair of  $b$ , which also holds the same two values.

At the join point of the two branches, RAIVE tests if the left and right pairs of  $e$  are identical or have negligible differences. In this case, since they are identical, the instability at line 10 is benign. No forking is needed. Intuitively, although the actual and ideal executions have different control flow inside the `max()` method due to the error, the relative error of the return value is very small (i.e.  $1/r$ ). In other words, the predicate at line 10 only becomes unstable when  $a$  and  $b$  are very close; however in such a case, returning either  $a$  or  $b$  does not make much difference and thus the instability is benign. RAIVE continues execution with the full vectors.

In box  $\textcircled{G}$ , RAIVE detects error inflation at line 13 and introduces artificial errors. The errors cannot be suppressed by the operation at line 14. As such, another unstable predicate is detected at line 15. However in this case, the two branches yield different left and right pairs in  $h$ . RAIVE forks the execution (box  $\textcircled{H}$ ). In one execution, the left pair of  $h$  is copied to the right pair so that  $h$  holds four identical value of 10 for the continuation. In the other execution, the right pair is copied to the left. As such, both executions can proceed with full vectors. Eventually, the two executions report two possible outputs,  $-20$  and  $20$ , which precisely capture the possible output variations in the presence of any internal error or external error (e.g., error on input variable  $a$ ).

**Key Advances Over the State-of-The-Art.** RAIVE has the following advantages over the state-of-the-art runtime instability detector TAG [11]. TAG detects relative error inflation, i.e., instances when the relative error of the result of a floating point operation is substantially larger than those of the operands. It taints a variable when its relative error is inflated. It further monitors the propagation of the taint bit. The bit may be reset when a tainted operand is used in a binary operation with a much larger untainted operand. If a taint bit reaches a discrete factor, the execution is considered unstable and terminated. Re-execution with a higher precision is needed.

- *Capturing Output Variations Caused by Both Internal and External Errors.* TAG cannot detect output variations. Instead, when TAG detects an unstable discrete factor, it simply terminates the execution and switches to a higher precision. Further-

TAG	RAIVE
$x'$ the error bit of $x$	$\langle x_1, x_2, x_3, x_4 \rangle$ denoted as $x^v$ ,
1 $a = 2^{54}$ ;	$a^v = \langle 2^{54}, 2^{54}, 2^{54}, 2^{54} \rangle$ ;
1.1 <span style="border: 1px solid black; padding: 2px;"><math>a' = F</math></span> ;	
...	...
3 $c = b - a$ ;	$c^v = b^v - a^v$ ;
3.1 <span style="border: 1px solid black; padding: 2px;"><b>if</b> <math>(b' \wedge a')</math> <math>c' = T</math>;</span>	<b>if</b> $(\max(\varepsilon_b, \varepsilon_a) - \varepsilon_c > \tau_c)$ $\delta = 2^{\max(\varepsilon_{b_1}, \varepsilon_{a_1}) - \tau_c + 1}$ ; $c^v = c^v + \langle -\delta, \delta, -\delta, \delta \rangle$ ;
3.2 <span style="border: 1px solid black; padding: 2px;"><b>elseif</b> <math>(b')</math> <math>c' = \neg(\varepsilon_a - \varepsilon_b &gt; \tau_s)</math>;</span>	
3.3 <span style="border: 1px solid black; padding: 2px;"><b>elseif</b> <math>(a')</math> <math>c' = \neg(\varepsilon_b - \varepsilon_a &gt; \tau_s)</math>;</span>	
3.4 <span style="border: 1px solid black; padding: 2px;"><b>else</b></span>	
3.5 <span style="border: 1px solid black; padding: 2px;"><math>c' = (\max(\varepsilon_b, \varepsilon_b) - \varepsilon_c &gt; \tau_c)</math>;</span>	
...	...
18 $o = h * 2$ ;	$o^v = h^v * 2^v$ ;
18.1 <span style="border: 1px solid black; padding: 2px;"><math>o' = h' \vee 2'</math></span>	

Fig. 2.4.: Boxed statements correspond to instrumentation. Note that in RAIVE, the original floating point related statements are completely replaced by vector statements. Labels 3.1-3.5 denote instrumentation for line 3.

more, as we discussed in Section 2.1, if external errors are possible, even using the infinite representation precision cannot address the problem that the execution may produce different outputs due to the errors. In contrast, RAIVE does not terminate an unstable execution, but rather forks multiple executions to capture output variations. It handles both internal and external errors.

The third column in Fig. 2.3 shows the execution of TAG. In box  $\textcircled{A}$ , value 0 in  $c$  is tainted due to the error inflation at line 3. But the taint bit is reset at line 4. Later, TAG detects that a tainted value reaches a discrete factor in box  $\textcircled{B}$  and terminates. In contrast, RAIVE reports the possible output variations. Assume the value  $2^{54}$  of  $a$

at line 1 is loaded from a file. Even if we used infinite precision in the execution, the same output variation would still occur if  $a$  has some small external error.

- *Handling Benign Differences.* TAG cannot automatically determine if an unstable factor is benign or harmful. It simply terminates when an unstable factor is detected. Or, the user can choose to manually annotate some discrete factors beforehand such that instability warnings at those factors are ignored. The predicate in box  $\textcircled{\text{B}}$  in Fig. 2.3 is unstable but benign. TAG cannot handle this. In contrast, RAIVE leverages a novel runtime that evaluates both branches to overcome the problem.

- *Avoiding Undesirable Error Suppression.* TAG uses a single bit to denote the presence of an inflated relative error. However, this may become problematic in error suppression. In box  $\textcircled{\text{C}}$ , the subtraction  $e - a$  causes a relative error inflation. Because  $f$  is 0 at line 14, substantially smaller than 0.5, the error bit is reset. The predicate at line 15 is hence considered stable. This is problematic: as shown in the second and the third columns, the actual and ideal executions have different branch outcomes at line 15, leading to different final outputs.

In normal cases, absolute errors are much smaller than the actual values even when the relative errors are inflated. However at line 14, the absolute error of  $f$  is 1.0 (from the actual and ideal values), larger than the value of  $f$  itself (i.e.  $f = 0$ ) and even the other operand 0.5. Unfortunately, this information cannot be represented by the single error bit.

In contrast, RAIVE injects artificial errors that denote the bounds of the absolute errors (box  $\textcircled{\text{G}}$ ). It easily supports error propagation in which operand(s) with inflated error(s) lead to a result with inflated error, since the different values in the operand vectors often lead to different values in the result vector. Furthermore, error suppression can occur implicitly and appropriately during floating point operations. For example, in box  $\textcircled{\text{D}}$ , the errors are suppressed by the subtraction whereas in box  $\textcircled{\text{G}}$ , the operand 0.5 is not large enough to suppress the errors.

- *Lower Runtime Overhead by Vectorization.* Although TAG features much lower overhead compared to techniques based on high precision libraries [10] and affine anal-

ysis [8], it is still very expensive (827% overhead according to Section 5.4). This is due to the expensive instrumentation and the poor instruction pipeline performance. The left column of Fig. 2.4 shows part of the TAG instrumentation for the example in Fig. 2.3. Lines 1.1, 3.1-3.5, 18.1 denote instrumentation for lines 1, 3, and 18, respectively. Line 3.1 means that the result is tagged true if both operands are tagged (true). Lines 3.2-3.3 handle the case when only one operand is tagged. In this case, if the difference between the operand exponents  $\varepsilon_a$  and  $\varepsilon_b$  is larger than a threshold  $\tau_s$ , the relative error is suppressed and the result tag  $c'$  is false. Line 3.5 detects relative error inflation. Instrumentation similar to lines 3.1-3.5 is added for each subtraction/addition. For multiplications and divisions, since neither inflation nor suppression could happen, the result tag is simply the union of the operand tags (e.g. line 18.1). The nesting branches in instrumentation lead to poor instruction pipeline performance. The fine-grained interleaving of the boolean type error bit propagation and the floating point type computation also prevents aggressive instruction scheduling, causing performance penalty.

RAIVE leverages the native support for vectors. In particular, each original floating point instruction is rewritten to a vector instruction. Note that the operations on individual vector values are performed simultaneously on separate FPUs such that they do not cost additional cycle(s). Such vectorization is shown in the right column in Fig. 2.4. At line 3, the original subtraction is replaced with a vector subtraction. In addition, we only need instrumentation for detecting error inflation and checking discrete factors. For example, the instrumentation for line 3 in RAIVE is much simpler than that in TAG. No instrumentation is needed for multiplications or divisions. This not only reduces the number of instructions, but also avoids interleavings of integer/boolean instructions and floating point instructions.

## 2.4 Design of RAIVE Runtime

RAIVE is a runtime technique. The given floating point program is transformed using the compiler. This transformed program has a special execution model that is supported via vectorization. The execution may fork multiple processes and produce a set of outputs that denote possible variations in the presence of errors. The execution model can be intuitively described as follows. The program state (for floating point variables) is a set of pairs, each representing an interval for the possible values of the variable (e.g.,  $\langle x_1, x_2 \rangle$  for variable  $x$ ). Initially, each pair has two identical values (e.g.,  $\langle x, x \rangle$ ). Floating point operations are performed on the pairs. RAIVE monitors these operations. When it detects relative error inflation, it introduces artificial errors to the pair so that it becomes  $\langle x - \delta, x + \delta \rangle$ . The introduced error  $\delta$  over-approximates the error incurred by the operation and hence the pair denotes the lower and upper bounds of  $x$  at this operation. Upon encountering a conditional, if there exists a pair of values  $\langle x_1, x_2 \rangle$  for  $x$  such that one of the values ( $x_1$  or  $x_2$ ) satisfies the condition and the other does not, RAIVE executes both branch outcomes of the conditional. The states for the branch outcomes are managed separately. At the join point of the branch we get two different sets of (output) pairs –  $\langle y_1^t, y_2^t \rangle$  for the true branch and  $\langle y_1^f, y_2^f \rangle$  for the false branch. If these pairs agree, RAIVE joins them, knowing that there is no output variation induced by the branch deviation. If they do not agree, RAIVE separates the pairs (from the two branches) permanently via forking.

Observe that the aforementioned execution model requires maintaining a store for pairs and supporting non-interference when evaluating both branches of a conditional. In order to achieve these goals, we develop a vector based semantics that can be implemented using the latest vector instruction support. In particular, we use a vector of four values to denote each floating point variable in the original program. Normally, the first two values (called the *left pair*) denote the interval of the variable value and the right pair is simply a duplication of the left pair. When both branches

of a conditional need to be evaluated, the left and right pairs are used/updated in isolation to achieve non-interference.

### 2.4.1 Semantics

<i>Program</i> $P$	$::= s$
<i>Stmt</i> $s$	$::= s_1; s_2 \mid \mathbf{skip} \mid x := e \mid x := \mathbf{f2i}(y) \mid$ $\mathbf{if} \ x \bowtie y \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2 \mid$ $\mathbf{while} \ x \bowtie y \ \mathbf{do} \ s$
<i>Expr</i> $e$	$::= x \mid v \mid e_1 \ op \ e_2 \mid \mathbf{sin}(e)$
<i>BinOp</i> $op$	$::= + \mid - \mid * \mid /$
<i>Value</i> $v, w$	$::= n \mid r \mid b$
$Var \ x, y \in Identifier \quad n \in \mathbb{Z} \quad r \in Real \quad b \in Boolean$	

Fig. 2.5.: Language

We use a language in Fig. 2.5 to facilitate discussion. We model two kinds of discrete factors: **f2i** denoting a type cast from floating point to integer and  $\bowtie$  denoting a relational operation on two floating point variables. Mathematical functions are modeled by a representative function **sin**( $e$ ).

The semantics is presented in Fig. 2.6. The related definitions are presented close to the top. In particular, the store  $\sigma$  is a mapping from a variable to a vector of four values. It is constituted by two disjoint stores,  $\sigma_l$  and  $\sigma_r$ , denoting the mappings from a variable to the first two values (i.e., *left pair*), and to the last two values (i.e., *right pair*), respectively. The execution mode is denoted by  $\omega$ , which has three possible values: *REGULAR* denoting *regular execution* in which both the left and right stores are updated, *LEFT* denoting *left execution* in which only the left store is updated, and *RIGHT* denoting *right execution* in which only the right store is updated. When an unstable predicate is encountered, that is, the values in the vectors (involved in



the predicate) yield non-uniform branch outcomes, RAIVE needs to determine if the instability is benign by executing both branches in sequence. The executions of the two branches need to be isolated so that they do not interfere with each other and their results can be properly compared at the join point. In particular, the first branch execution only operates on the left pairs, called the *left mode*, whereas the second branch execution only operates on the right pairs, called the *right mode*. Execution modes and mode changes are implemented using the *mask* instruction provided by the CPU. The instruction defines which values in a vector are visible and operatable. The variable join set  $\Phi$  contains the variables defined during the branch executions of an unstable predicate. At the branch join point,  $\Phi$  is scanned to determine if forking is necessary.

To make presentation easier, we extend the syntax of expression to represent a vector of values, and the syntax of statement to include a few new commands: **mode\_switch** to switch the current execution mode; **join** is to commit the updates from the two branches of a predicate and determine if the execution should be forked; **spawn** is to fork an execution. These statements are auxiliary and only present during evaluation.

## Expression Rules

Expression rules evaluate a floating point expression to a vector of four values. The evaluation may be moderated by the execution mode. During expression evaluation, relative error inflation is also detected. Rule [CONST] shows that a floating point value  $v$  is expanded to a vector of four identical values.

Subtraction of two vectors (Rule [SUB]) is performed by subtracting the corresponding values and adjusting the resulting values with  $\delta$ , which is computed as follows: (1) if the current execution mode is *REGULAR* or *LEFT* (for branch execution using the left store) and there is relative error inflation (or cancellation) in the subtraction of the actual values  $v_1$  and  $w_1$ , or  $v_2$  and  $w_2$ ,  $\delta$  is computed from the

exponents by  $\delta = 2^{\max(\varepsilon_{v_1}, \varepsilon_{w_1}, \varepsilon_{v_2}, \varepsilon_{w_2}) - \tau_c + 1}$ . Intuitively,  $\tau_c$  is the threshold to detect inflation, meaning that an addition/subtraction causes relative error inflation if the result is left-shifted by at least  $\tau_c$  bits, suggesting the first  $\tau_c$  significant bits of the two operands are identical. Since we consider that the result value cannot be trusted in this case, it is equivalent to that the absolute error of the result can be as large as the value represented by the  $\tau_c$ -th significant bit of the largest operand (i.e. the first bit that differs in the operands), which can be computed by the aforementioned formula. (2) If the current mode is *RIGHT*, the values in the right pairs are used to detect inflation and compute  $\delta$ . (3) If there is no inflation,  $\delta = 0$ . Additions are handled in the same way. The first line in box  $\textcircled{D}$  in Fig. 2.3 shows an example of condition (1), with  $\tau_c = 49$ .

In Rule [MULT], the values in the operand vectors are multiplied respectively, denoting the propagation of errors, if injected previously. The rule for division is similar. Note that although multiplication or division can enlarge absolute errors, they do not inflate relative errors. Intuitively, when an operand  $x$  with an absolute error  $\hat{\Delta}_x$  is multiplied with another operand  $y$ , both  $x$  and  $\hat{\Delta}_x$  are enlarged by  $y$  so that the resulting relative error (i.e., the ratio between the absolute error of result and the actual result) is unchanged.

For non-library function calls, parameter vectors are directly passed to callees and used there. In contrast, an external library function is generally evaluated on the respective vector values (Rule [SIN]). We cannot vectorize library functions as we do not have their source code. For better efficiency, we re-implement some frequently used library functions to directly support vectorization.

## Statement Rules

RAIVE has three execution modes. Statement semantics may be different in these modes.

**Regular Mode.** The first set of statement rules is for the regular execution mode. Rule [ASSIGN] describes that all the four fields of the left hand side variable are updated.

Most of the complexity of RAIVE lies in the handling of conditional statements. It first determines if a predicate is stable. If so, the execution proceeds with the uniform branch outcome. Otherwise, it executes the two branches in sequence to detect if the instability is benign. If not, RAIVE forks the execution to capture the different effects of the instability.

Rules [IF-Stable-True] and [IF-Stable-False] describe that during regular execution, if the left pair has the identical true/false value, the execution proceeds to the true/false branch. Note that only the left pair needs to be inspected as the right pair is a duplication during regular execution.

Rules [IF-Unstable-T] and [IF-Unstable-F] specify the semantics when the left pair does not concur. If the first value  $b_1$  is false (Rule [IF-Unstable-F]), it first executes the false branch in the left mode, and then executes the **mode\_switch** statement to change to the right mode for true branch execution (Rule [MODE]). After executing the two branches, the updates in them are inspected by the **join** statement.

Rules [JOIN] and [JOIN-Split] specify the semantics of the **join** statement. In Rule [JOIN], the left and right pairs of all live updated variables are identical or have trivial differences, suggesting benign instability. In particular, RAIVE inspects each *live* variable in  $\Phi$ , by comparing its left and right pairs. If the comparison of two values incurs cancellation (i.e. the number of cancelled bits is larger than the threshold  $\tau_c$ ), we consider the two values under comparison have trivial difference. If the differences are always trivial for all live variables, the instability is benign, the execution is not forked. During inspection, only variables that are live at the join point of the branches are considered (i.e. those that may be used beyond the join point). We use a standard static live variable analysis. After joining,  $\Phi$  is reset. In Rule [JOIN-Split], if the differences are not trivial, the execution is split. In the parent process, the updates during right execution are discarded, by overwriting the

right store values with the left store values. In the child process, the updates during left execution are discarded, by overwriting the left store values with the right store values.

Rules [F2I] and [F2I-Split] specify the semantics of type casts from floating point to integer in the regular mode. If the left pair yields different integer values, the execution forks based on the different discrete integer values.

The evaluation of while loops (Rule [WHILE]) is standard, which unrolls the loop once each time. Since loops are essentially unrolled during evaluation, unstable loop predicates are handled like normal predicates. To prevent potential infinite forking, we limit the number of forks allowed for a loop predicate (to 10). In practice, such a limit is never reached. But if the limit is reached simply continue with one of the executions.

*Example.* Box  $\textcircled{\text{H}}$  in Fig. 2.3 shows an example of Rule [JOIN-Split]. At the join point,  $\Phi = \{h\}$  and the differences between the left and right pairs of  $h$  are substantial and the execution is forked. In the continuation of the original execution,  $h^v = \langle 10, 10, 10, 10 \rangle$  after copying the left pair to the right, whereas in the spawned execution,  $h^v = \langle -10, -10, -10, -10 \rangle$  after copying the right to the left.  $\square$

**Right Mode.** The next set of rules is for the right execution mode. According to Rule [ASSIGN-Right], in right execution, only the right pairs are updated, while the left pairs retain their values. Moreover, the left-hand-side variable  $x$  is inserted to the variable set  $\Phi$  for inspection at the join point.

Rules [R-IF-Stable-T], [R-IF-Stable-F], and [R-IF-Unstable] evaluate conditional statements in the right mode. Rule [R-IF-Unstable] specifies the case in which another unstable predicate is encountered, which suggests nesting unstable predicates. Since we use only the right pairs in the right mode, we cannot afford evaluating the two branches of the inner unstable predicate in sequence. Therefore, we fork the execution right away. Particularly, the original execution proceeds with the true branch (of the inner predicate) with the same right mode. The spawned execution proceeds with the false branch in the regular mode, discarding all the updates during the for-

Program	Exec. I			Exec. II		
	$\sigma$	$\omega$	$\Phi$	$\sigma$	$\omega$	$\Phi$
1 $a = r;$	$\sigma[a] = \langle r, r, r, r \rangle$	REGULAR	$\{\}$			
2 $b = \text{input}();$	$\sigma[b] = \langle r, r, r, r \rangle$	REGULAR	$\{\}$			
3 $c = a - b;$	$\sigma[c] = \langle -\delta, \delta, -\delta, \delta \rangle$	REGULAR	$\{\}$			
4 <b>if</b> ( $c < 0$ )	$\langle T, F, T, F \rangle$	LEFT	$\{\}$			
5 $t = a + 6;$	$\sigma[t] = \langle r + 6, r + 6, -, - \rangle$	LEFT	$\{t\}$			
6 <b>else</b> {		RIGHT	$\{t\}$			
7 $c = b + 2;$	$\sigma[c] = \langle -, -, r + 2, r + 2 \rangle$	RIGHT	$\{t, c\}$			
8 $d = c - a;$	$\sigma[d] = \langle -, -, 2 - \delta, 2 + \delta \rangle$	RIGHT	$\{t, c, d\}$			
9 <b>if</b> ( $d < 0$ )	$\langle -, -, T, F \rangle$	RIGHT	$\{t, c, d\}$	$\sigma[c] = \langle r + 2, \dots, r + 2 \rangle$	REGULAR	$\{\}$
				$\sigma[d] = \langle 2 - \delta, 2 + \delta, 2 - \delta, 2 + \delta \rangle$		
				$\sigma[t] = \langle 0, 0, 0, 0 \rangle$		
10 $t = c + 6;$	$\sigma[t] = \langle -, -, r + 8, r + 8 \rangle$	RIGHT	$\{t, c, d\}$			
11 <b>else</b> $t = a * 2;$				$\sigma[t] = \langle 2r, 2r, 2r, 2r \rangle$	REGULAR	$\{\}$
}	$\sigma[t] = \langle r + 6, r + 6, r + 8, r + 8 \rangle$	REGULAR	$\{\}$			
12 $o = t + a;$	$\sigma[o] = \langle 2r + 6, 2r + 6, 2r + 8, 2r + 8 \rangle$	REGULAR	$\{\}$	$\sigma[o] = \langle 3r, 3r, 3r, 3r \rangle$	REGULAR	$\{\}$

Fig. 2.7.: An example for nesting unstable predicates. Symbol  $r$  represents a large floating point value. Assume the input value is  $r + 1$  which cannot be precisely represented and hence the represented value is  $r$  at line 2. Other large values can be precisely represented.

mer branch evaluation (in the left mode). As such, the **join** operation at the join point of the outer unstable predicate has no effect. The left mode rules are similar and hence omitted.

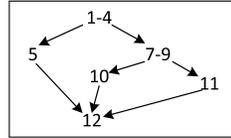


Fig. 2.8.: The control flow graph for the example in Fig. 2.7.

*Example.* Fig. 2.7 shows an example for nesting unstable predicates. The first column shows the program. The next three columns show its original execution. The last three columns show the spawned execution. Initially,  $a$  has a large value  $r$  and the input to  $b$  is  $r + 1$ . However due to the limited precision, the represented value in  $b$  is  $r$ . At line 3, the actual value of  $c$  is 0, and artificial errors are introduced due to the relative error inflation. Since line 4 is unstable, line 5 is executed in the left mode and  $\Phi$  contains  $t$ . The execution is switched to the right mode at line 6. Another inflation is detected at line 8 so that errors are introduced to the third and fourth values, leading to non-uniform branch outcome. The execution is forked according to Rule [R-IF-Unstable]. Note that the updates on the left pairs of all the variables in  $\Phi$  are discarded and replaced with the right pairs in the spawned execution, which proceeds with the regular mode and an empty  $\Phi$ . The original execution continues with the right mode and the join operation is performed before line 12. Since variables  $c$  and  $d$  are not live beyond line 12, only  $t$ 's vector values are compared. Since the differences are trivial, the execution is not forked.

As in Fig. 2.8, RAIVE captures the effects of the leftmost path (1-5→12) and the middle path (1-4→7-10→12) through the original execution, and the effect of the rightmost path (1-4→7-9→11-12) through the spawned execution.  $\square$

### 2.4.2 Understanding the Essence of RAIVE

Next, we informally discuss a few properties of RAIVE

First, the goal of RAIVE is to expose output variations caused by discrete differences. Such differences are caused by discrete factors (i.e. operations that have floating point operands and discrete type result such as integer or boolean) instead of floating point operations. Hence, RAIVE is most suitable for programs with both floating point and discrete operations (e.g., the example in Section 2.1), and less effective for mathematical cores composed of floating point operations only (e.g., a code snippet that computes  $y = x^2 + 2x + 3$ ). The real world floating point programs we study contain non-trivial number of discrete operations. As such, output variations are mainly due to discrete differences (caused by errors). Our experiments will illustrate this later. Intuitively, errors through floating point operations cause continuous output changes, whereas errors through discrete operations may cause discontinuous differences that are usually much more substantial.

Second, RAIVE introduces artificial errors that are bounds of the absolute error when relative error inflation occurs. The resulting value differences in the vector are further propagated via the following vectorized floating point operations. However, the values in the vector of a variable are not guaranteed to stay as the bounds of the variable value, as during binary operations errors from multiple operands may interfere with each other. In other words, at the moment when the artificial errors are introduced, the values in the vector denote the lower and upper bounds of the actual value. However, when these values (with errors) are propagated to other variables through operations, especially binary operations, the values in the vectors of result variables may not represent the bounds. A very important point, however, is that RAIVE *does not need to guarantee these values to be the bounds*. Instead, the (different) values in a vector are essentially samples in the error range. *These samples are sufficiently distant so that they expose discrete differences*. This is because the artificial errors RAIVE introduces are very conservative. In contrast, affine analysis [8]

focuses on modeling *continuous* changes caused by errors through affine formulas. Hence, they need to compute the bounds of values, which is very expensive (i.e. 3-4 magnitude of slowdown according to [8]).

Note that an approach that tries to use multiple sample runs to expose output variations is inferior to RAIVE. This is because RAIVE essentially not only packs multiple sample executions into a vectorized execution, but also avoids unnecessary samples by detecting benign unstable predicates. According to our experiment (Section 5.4), while an execution may encounter many unstable predicates, most of them are found to be benign.

Third, similar to existing work [11, 20], RAIVE uses a threshold to detect relative error inflation, which is key to detecting unstable discrete factors. In theory, the detection is neither sound nor complete due to the use of threshold. However, RAIVE does not aim to detect unstable discrete factors, but rather expose output variations in the presence of errors. It has a sophisticated runtime mechanism to determine if an unstable predicate is benign. Therefore, we use a conservative threshold for relative error inflation detection. The resulting false positives of unstable predicates are effectively suppressed by the runtime mechanism. Furthermore, as shown in Section 5.4, all inputs falling into the unstable range tend to cause the same discrete differences and hence the same (or highly similar) output variations. The net effect of having false positives in detecting unstable factors is to report the same output variations for a larger set of inputs. We also show in Section 5.4 that the unstable ranges are very tiny such that even though RAIVE reports output variations for a larger range, such output variations are well possible because they can be easily induced by external errors.

## 2.5 Evaluation

We implement RAIVE using GCC-4.7.2. to support C/C++ and Fortran. We leverage the *Advanced Vector Extensions* (AVX) on x86.64 architecture to support

vectors. It features instructions operating on 256-bit vector registers, called `ymm` registers, which can store up to four double precision floating point values. The execution mask in our semantics is also natively supported by the CPU. We modify the lexical analysis of GCC. Each floating point variable is replaced with a 256-bit type supported by GCC. Floating point operations are also replaced accordingly. After lexical analysis, we further instrument the GIMPLE IR code to support functionalities such as error inflation detection and the execution modes.

Since the Fortran frontend of GCC does not support AVX natively, some language features are difficult to vectorize. One example is the primitive complex type in Fortran. Since a double precision complex value consists of two 64-bit values (the real and the imag parts), a complex value cannot be directly transformed to a vector. Hence we transform a complex value to a C struct that consists of two 256-bit vectors. This transformation is more challenging than vectorizing scalar floating point variables because we need to further replace the operations on complex values with operations on the C struct values. RAIVE handles all the language features that we have encountered in the benchmark set.

We evaluate efficiency and effectiveness of RAIVE and compare it with HPL [10,11] and TAG [11]. HPL uses 128-bit quadruple precision. For fair comparison, we re-implemented HPL using GCC. Our implementation is faster than [10]. We use the programs in [11] for comparison, including SPEC CFP2000 and a biochemical data processing program `deisotope`. In addition, we include two widely used data-mining programs `k-means` and `pagerank`. Note that these programs are much more complex than those used in studies that focus on numerical programs (e.g., [8]). They contain a lot of discrete operations. Their LOCs are shown in Table 2.1 column 2. All experiments were run on a machine with Intel i7-2640M 2.80GHz processor and 8GB RAM.

Table 2.1.: Performance (o/h stands for overhead).

Program	LOC	Native Time(s)	HPL o/h	TAG o/h	Vec-only		RAIVE	
					Time(s)	o/h	Time(s)	o/h
168.wupwise	2.1k	79.70	6043%	292%	141.6	178%	179.5	225%
171.swim	0.4k	122.5	7356%	359%	163.7	133%	178.5	146%
172.mgrid	0.4k	39.27	35031%	1639%	89.9	228%	355.7	905%
173.applu	4k	39.91	21112%	1458%	108.6	272%	163.9	410%
177.mesa	63k	8.60	3364%	538%	19.97	232%	27.63	321%
178.galgel	15.3k	28.26	23867%	1592%	124.5	441%	209.2	740%
179.art	1.2k	10.08	15786%	735%	21.13	210%	28.00	277%
183.quake	1.3k	12.55	21876%	1525%	52.37	417%	65.28	520%
187.facerec	2.4k	35.72	10784%	1492%	145.8	408%	180.8	506%
188.amp	13.4k	53.02	16263%	822%	78.76	148%	160.6	303%
189.lucas	3k	24.24	23536%	1333%	74.52	307%	87.56	361%
191.fma3d	60k	38.33	13169%	1110%	142.9	373%	155.9	406%
200.sixtrack	47.2k	59.50	47540%	1056%	95.34	160%	214.6	360%
301.apsi	7.5k	51.47	13220%	719%	104.6	203%	166.1	322%
deisotope	2.2k	11.82	469%	205%	15.01	127%	18.69	158%
k-means	7k	12.69	925%	329%	14.12	111%	16.23	127%
pagerank	0.25k	13.29	5491%	1653%	41.17	309%	66.83	502%
AVERAGE			9826%	827%		229%		340%

### 2.5.1 Performance

In the first experiment, we evaluate the runtime overhead of RAIVE. We use the reference inputs from SPEC. For `deisotope`, `k-means` and `pagerank`, we use the inputs that come with the programs. The results are shown in Table 2.1. Column 3 shows the native execution time. Columns 4 and 5 present the overhead for HPL and TAG. Observe that the average overhead of HPL exceeds **98x**. The average overhead for TAG is **827%**.

The last two columns present the time and overhead of RAIVE. We collect the data with  $\tau_c = 48$ , which is the threshold used in detecting relative error inflation (Section 2.2). The average overhead is **340%**, which is **2.43** times smaller than that in the TAG approach. The higher overhead in some of the programs (e.g., `178.mgrid`) is due to the exceptionally large number of additions and subtractions in the hot loops. These operations have to be instrumented for error inflation detection.

We further study the breakdown of overhead for RAIVE. We run the programs with vector instructions but without detecting instability. It means that a program is transformed to its vector version, where floating point values are stored in 256-bit vectors and operated with AVX instructions. The four values in a vector are always identical. This is to study the overhead of vectorization. The results are shown in the `vec-only` columns. The average overhead is **229%**. While theoretically AVX instructions should not cost additional cycles, there are a few possible reasons for the slow-down. First, the processor we use (CPUID: 06\_2AH) is an early version supporting AVX. According to the Intel Manual, the latencies for AVX in our processor are higher than later versions [21]. Second, we suspect the compiler is not able to perform aggressive optimizations for AVX instructions because they are relatively new. We anticipate RAIVE will have lower overhead in the future with new hardware and better compiler support. We argue that the overhead is acceptable given the capability of reporting output variations. Note that without our technique, achieving

the same capability may entail a large number of sample executions, especially when the input dimension is high.

### 2.5.2 Effectiveness

**Instability Detection.** The experiment is setup as follows. For each program, we collect  $1E+14$  samples within an input range. We execute the program on these samples with HPL, TAG, and RAIVE. We extend HPL to also compute the actual floating point values with 64-bit precision such that they can be compared with their high precision version to collect the ground truth (of instability). For TAG and RAIVE, we collect data for three configurations with different  $\tau_c$  values. The results are shown in Table 2.2. We only focus on the programs with instability reported. For each program, the input sample range and the total number of samples are shown in the first row labeled OVERALL. The six following rows present the detection results for the configurations. The second column shows the configuration. The third column shows the number of samples in which instability is reported (i.e. at least one unstable discrete factor has been encountered), and its percentage over the total sample number is presented in the fourth column. The last column shows the detected problematic range that contains the unstable samples.

We have the following observations. (1) RAIVE has comparable or better effectiveness than TAG in instability detection. The detected ranges by RAIVE are similar to or smaller than those by TAG for the same configuration. Both of them can correctly determine that over 99.99% of the inputs lead to stable executions. However, the overhead of RAIVE is **2.43x** smaller than TAG. (2) Threshold  $\tau_c = 52$  is the best configuration for most benchmarks considered. It reports the smallest number of unstable samples without any false negatives except for **pagerank**. The maximum possible threshold value is 53. Threshold  $\tau_c = 44$  is safe (for the programs we considered) as it does not cause any false negatives. Note that using a larger  $\tau_c$  means that we have a stricter condition in determining relative error inflation. (3) With

$\tau_c = 44$ , although the number of detected unstable samples is 3.75-2258 times larger than the ground truth, these samples only denote a trivial part of the input range. This implies it is unlikely for RAIVE to have false warnings in instability detection. In contrast, according to [11], interval analysis and techniques based on solely detecting error inflations report a lot more false positives. More importantly, as we will show later, the false positives in detecting instability have little effects on the main results, output variations.

**Handling Benign Unstable Predicates and Forking.** An important advantage of RAIVE is the capability of handling benign unstable predicates. The results are in Columns 2-4 in Table 2.3. Column 2 lists the average number of unstable predicates encountered in *a single sample execution*. Note that column 3 in Table 2.2 shows *the number of sample runs* in which unstable factors were detected. They have different meanings. Column 3 in Table 2.3 shows how often RAIVE can proceed without forking after executing the two branches separately. Column 4 shows the number of forks. Observe that 2 of the 6 programs (with instability detected) encounter benign unstable predicates. If these predicates were not properly handled, there would be a lot of unnecessary forks. Also observe that since most unstable predicates are benign, the number of forks is very small. Intuitively, it is unlikely for a program to have multiple sources of instability for *a given input*.

**Output Variations.** The experiment is set up as follows. We first identify the input range that is reported as unstable by HPL. Then we collect two samples at the boundary of the range, denoted as *lb* and *ub*, and use the output variations between the two executions as the ground truth as they denote the two boundary *stable* executions. Observe that the range is mostly very small such that external errors can easily cause input variations in the range. Then we execute the program on RAIVE for all the unstable samples (with  $\tau_c = 52$ ) and collect the output variations for each sample. We then compute the output coverage for a sample *i* as follows. Let  $O_1, \dots, O_n$  be the *n* output variables and  $O_t(ub)$  the value of a *t*'th variable in the *ub*

Table 2.2.: Instability detection.

	approach	# of cases	%	detected range
equake	OVERALL	1E+14		[0.8650, 0.8750]
	HPL	2	2.00E-12%	[0.8690799016130847, 0.8690799016130848]
	TAG( $\tau_c=44$ )	4516	4.52E-09%	[0.86907990160 <b>26333</b> , 0.8690799016130848]
	TAG( $\tau_c=48$ )	279	2.79E-10%	[0.86907990161305 <b>70</b> , 0.8690799016130848]
	TAG( $\tau_c=50$ )	20	2.00E-11%	[0.86907990161308 <b>29</b> , 0.8690799016130848]
	RAIVE( $\tau_c=44$ )	4516	4.52E-09%	[0.86907990161 <b>28591</b> , 0.8690799016133106]
	RAIVE( $\tau_c=48$ )	280	2.80E-10%	[0.8690799016130 <b>709</b> , 0.8690799016130 <b>988</b> ]
	RAIVE( $\tau_c=52$ )	20	2.00E-11%	[0.86907990161308 <b>29</b> , 0.8690799016130848]
facerec	OVERALL	1E+14		[0.6694, 0.6695]
	HPL	30700	3.07E-08%	[0.6694295316764218, 0.6694295316764524]
	TAG( $\tau_c=44$ )	37267458	3.73E-05%	[0.6694295316 <b>577891</b> , 0.6694295316 <b>950752</b> ]
	TAG( $\tau_c=48$ )	2314523	2.31E-06%	[0.669429531675 <b>1556</b> , 0.6694295316777159]
	TAG( $\tau_c=52$ )	162943	1.63E-07%	[0.669429531676 <b>2312</b> , 0.6694295316766320]
	RAIVE( $\tau_c=44$ )	115423	1.15E-07%	[0.669429531676 <b>3000</b> , 0.6694295316766643]
	RAIVE( $\tau_c=48$ )	70847	7.08E-08 %	[0.669429531676 <b>2782</b> , 0.6694295316766333]
	RAIVE( $\tau_c=52$ )	55712	5.57E-08%	[0.669429531676 <b>2792</b> , 0.6694295316765709]
galgel	OVERALL	1E+14		[0.8184, 0.8185]
	HPL	57695	5.77E-08%	[0.8184459012000007, 0.8184459012253359]
	TAG( $\tau_c=44$ )	37972131	3.80E-05%	[0.818445 <b>8998299998</b> , 0.81844590 <b>39575860</b> ]
	TAG( $\tau_c=48$ )	3728089	3.28E-06%	[0.81844590 <b>02196792</b> , 0.81844590 <b>20723309</b> ]
	TAG( $\tau_c=52$ )	1233455	1.23E-06%	[0.818445901 <b>9084903</b> , 0.81844590 <b>20723309</b> ]
	RAIVE( $\tau_c=44$ )	43930919	4.39E-05%	[0.81844 <b>37893753897</b> , 0.81844 <b>81724703180</b> ]
	RAIVE( $\tau_c=48$ )	3258832	3.26E-06%	[0.818445901 <b>1753019</b> , 0.8184459 <b>399554056</b> ]
	RAIVE( $\tau_c=52$ )	229573	2.30E-07%	[0.818445901 <b>1900151</b> , 0.818445901 <b>4121039</b> ]
deisotope	OVERALL	1E+14		[1.11, 1.12]
	HPL	2	2.00E-12%	[1.1156381266106556, 1.1156381266106557]
	TAG( $\tau_c=44$ )	653	6.53E-10%	[1.115638126610 <b>5905</b> , 1.1156381266106557]
	TAG( $\tau_c=48$ )	40	4.00E-11%	[1.11563812661065 <b>18</b> , 1.1156381266106557]
	TAG( $\tau_c=52$ )	5	5.00E-12%	[1.11563812661065 <b>53</b> , 1.1156381266106557]
	RAIVE( $\tau_c=44$ )	315	3.15E-10%	[1.1156381266106 <b>398</b> , 1.1156381266106712]
	RAIVE( $\tau_c=48$ )	22	2.20E-11%	[1.11563812661065 <b>45</b> , 1.1156381266106566]
	RAIVE( $\tau_c=52$ )	2	2.00E-12%	[1.1156381266106556, 1.1156381266106557]
k-means	OVERALL	1E+14		[0.5640, 0.5650]
	HPL	233	2.33E-10%	[0.56446068002405417, 0.56446068002405649]
	TAG( $\tau_c=44$ )	100819	1.01E-07%	[0.5644606800 <b>2355170</b> , 0.5644606800240 <b>55988</b> ]
	TAG( $\tau_c=48$ )	6064	6.06E-09%	[0.5644606800240 <b>2601</b> , 0.5644606800240 <b>8664</b> ]
	TAG( $\tau_c=52$ )	325	3.25E-10%	[0.56446068002405417, 0.56446068002405 <b>741</b> ]
	RAIVE( $\tau_c=44$ )	50572	5.06E-08%	[0.5644606800 <b>2380185</b> , 0.564460680024 <b>30756</b> ]
	RAIVE( $\tau_c=48$ )	3140	3.14E-09%	[0.5644606800240 <b>3901</b> , 0.5644606800240 <b>7040</b> ]
	RAIVE( $\tau_c=52$ )	325	3.25E-10%	[0.56446068002405417, 0.56446068002405 <b>741</b> ]
pagerank	OVERALL	1E+14		[1.10, 1.11]
	HPL	122	1.22E-10%	[1.1026503685992210, 1.1026503685992331]
	TAG( $\tau_c=44$ )	593	5.93E-10%	[1.102650368599 <b>1619</b> , 1.1026503685992211]
	TAG( $\tau_c=48$ )	38	3.80E-01%	[1.1026503685992 <b>174</b> , 1.1026503685992211]
	TAG( $\tau_c=52$ )	2	2.00E-12%	[1.1026503685992210, 1.1026503685992211]
	RAIVE( $\tau_c=44$ )	593	5.93E-10%	[1.102650368599 <b>1912</b> , 1.1026503685992 <b>504</b> ]
	RAIVE( $\tau_c=48$ )	38	3.80E-11%	[1.1026503685992 <b>190</b> , 1.1026503685992227]
	RAIVE( $\tau_c=52$ )	2	2.00E-12%	[1.1026503685992210, 1.1026503685992211]

sample. Since an execution in RAIVE may fork, we use  $range(O_t(i))$  to denote the range of  $O_t$  for the  $i$ th sample.

$$recall = \left( \sum_{t=1}^n \frac{range(O_t(i)) \cap [O_t(lb), O_t(ub)]}{|O_t(ub) - O_t(lb)|} \right) / n$$

$$precision = \left( \sum_{t=1}^n \frac{range(O_t(i)) \cap [O_t(lb), O_t(ub)]}{range(O_t(i))} \right) / n$$

Table 2.3.: Average number of unstable predicates, and forks for an execution, and output variations across samples. RSD stands for relative standard deviation.

program	# of unstable preds.	# preds (%) that merge	# fork	# output var.	precision	recall	%RSD
178.galgel	253385	253382 (99%)	3	7	100%	71%	-
183.equake	1	0 (0%)	1	12	99%	99%	69%
187.facerec	14	9 (64%)	5	10	100%	100%	6.4%
deisotope	1	0 (0%)	1	30	97%	97%	43%
k-means	1	0 (0%)	1	92	100%	100%	55%
pagerank	1	0 (0%)	1	10	100%	100%	16%

*Recall* denotes how much ground truth output variation is covered by RAIVE; *precision* represents how much output variation reported by RAIVE denotes true variation.

The results for  $\tau_c = 52$  are in columns 5-7 in Table 2.3. Column 5 shows the number of output variables. The last two columns show the average precision and recall over all samples. Observe that RAIVE has close to 100% precision and recall for most cases, meaning that *any sample* within the range can precisely predict the same output variations in the presence of (both internal and external) errors. This is because the output variations caused by discrete differences are much more substantial than those by continuous numerical operations and RAIVE can precisely simulate discrete differences caused by errors. Since the discrete differences are stable within the range, the output variations are mostly stable across sample runs too. The precision and recall are not 100% in some cases because of the continuous differences. `Galgel` has the lowest recall as its continuous differences are non-trivial compared to the discrete differences.

The last column shows the maximum relative standard deviation (i.e., standard deviation divided by mean) of the output values for the same output variable. We only present the maximum as there are outputs whose values are almost identical across all the forked runs as they are not affected by the path differences. Observe that there are substantial output variations. The RSD for `galgel` cannot be computed as many of its forked executions do not produce any output. Note that existing techniques

focusing on numerical behavior [8, 22] cannot report output variations caused by discrete differences.

We have also repeated the same experiment on  $\tau_c = 44$ . In this case, we have much more unstable sample runs. However, the results are very similar to those in Table 2.3. The precision and recall are still close to 100%. The only difference is that RSD values are smaller due to the larger sample size. The observation is hence that the output variations are not sensitive to the threshold. The reason is that all these unstable sample runs lead to the same path differences and hence the same output variations. In other words, using a more conservative (i.e., smaller) threshold only means that the same output variations are exposed by a larger set of inputs (falling in the unstable range). On the other hand, as long as an input falls into the unstable range, any errors, including internal and external errors, cause the same output variations.

### 2.5.3 Case Studies

**K-means** [16] implements a widely used clustering algorithm, which partitions inputs into  $k$  clusters by the given distance metrics. The core algorithm is shown in Fig. 2.9. In each iteration, it first computes the centroids for the current  $k$  clusters (line 2) and then the sum of the distances from each element to its centroid (line 3). It then traverses all elements to check if the current partition can be further improved (lines 5-14). This is done by checking for an element  $e$ , if there is a cluster  $J$  whose centroid is closer than  $e$ 's current cluster  $K$ . If so,  $e$  is moved to  $J$ . The algorithm repeats until the overall distance stabilizes (line 15).

Since there is an element  $e$  with a very similar distance to the centroids of  $J$  and  $K$ , leading to an unstable predicate  $dist_{new} < dist$  at line 10, RAIVE detects the instability and evaluates both branches. But it cannot merge the two branches and hence forks, yielding two different clustering results as shown in Fig. 2.10. While we used simple input data in the case study, real world data set could be very complex

```

1  do {
2    getclustermeans(cluster, data); /* Find the centroids */
3    total_dist = ...; /* Compute current total dist */
4    total_dist_new = 0;
5    foreach element e {
6      K = cluster[e]; /* Element e belongs to cluster K */
7      dist = distance[e];
8      foreach cluster J {
9        dist_new = euclid(e, J);
10       if (dist_new < dist) {
11         /* Move element e from cluster K to J. */
12         cluster[e] = J;
13         distance[e] = dist_new;
14         dist = dist_new;
15       } }
16     total_dist_new += dist;
17   }
18 } while (total_dist_new < total_dist);

```

Fig. 2.9.: Pseudocode snippet for k-means.

and can hardly be manually inspected. RAIVE can automatically identify the possible clustering outputs.

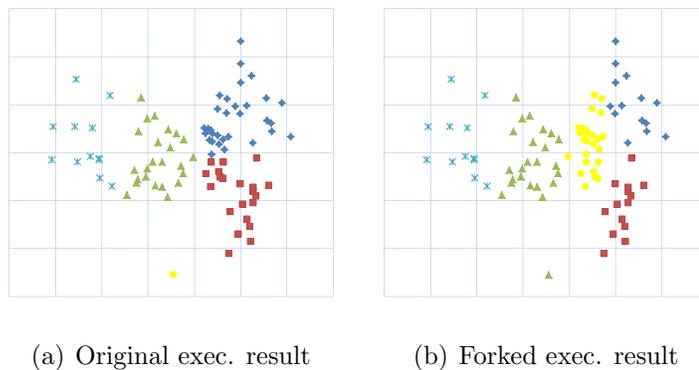


Fig. 2.10.: Clustering result variations of an unstable execution for k-means; 92 genes are grouped into five clusters; each cluster has a unique color.

```

1 while (cont) {
2   cont = 0;
3   foreach page p{
4     foreach neighbor n
5       neighbor_sum+ = score(n)/num_out_links;
6     new_score = ... + (... * neighbor_sum);
7     diff = abs(new_score - score(p));
8     if (diff > threshold) /* unstable */
9       cont = 1
10    }
11    score(p) = new_score;

```

Fig. 2.11.: Pseudocode snippet for pagerank.

**Pagerank** [17] is one of the most widely used algorithms in information retrieval. It was invented by Google and used to rank result pages for a search request. It computes a score for each page according to the number and quality of other pages linked to it. The score represents the probability that a random surfer will visit a page. A page with many high-score pages linked to it also has a high score. Part of the algorithm is shown in Fig. 2.11. In each iteration, a new score is computed for each page according to its incoming neighbors' previous scores. The algorithm repeats until the absolute difference between the new score and the old score is smaller than a threshold for all pages.

In this experiment, we use a set of similar pages. RAIVE reports instability at line 8 as `diff` and `threshold` are very close to each other. Hence, the execution is split to two and one of them iterates more. The iteration differences substantially change the final rankings of the pages. While the program printed the top 10 pages, we showed the top 3 in the following table.

Orig. (35 iterations)		Forked (34 iterations)	
page id	score	page id	score
722	0.999997000...	968	0.99999499...
723	0.999997000...	969	0.99999499...
724	0.999997000...	970	0.99999499...

Observe that the results are completely different. Also, there may not be a patch to the code that can fix the instability problem, which does not lie in the numerical core of the algorithm, but rather in the interface between the floating point computation and the discrete logic. It is a property of the algorithm and the provided input. Furthermore, generating inputs that expose such instabilities may not be as useful as in exposing functional bugs due to the infeasibility of fixing them. Therefore, we argue that showing the possible outputs to the user provides a reasonable solution.

```

/*coarse-grained search*/
142  Position = (LLX, LLY)
...
/*fine-grained search*/
168  CurSimilarity = GraphSimFct(LLX, LLY, ...)
169  If (CurSimilarity > Similarity) Then
170    Similarity = CurSimilarity
171    Position = (LLX, LLY)
172  EndIf

```

Fig. 2.12.: Benign unstable predicate in 187.facerec.

**Benign Unstable Predicate in 187.facerec.** *Facerec* is a Fortran program for face recognition. It consists of two phases of search. The first one is coarse-grained and the second one is fine-grained. It is possible that both phases identify the same object. The related code snippet is shown in Fig. 2.12. The object identified in the first phase is saved in `Position` at line 142. Lines 168-172 are for the second phase. In particular, the newly computed `CurSimilarity` is compared with the current `Similarity` (line 169). If the new similarity is larger, the current similarity and position are updated. If both phases identify the same object, the difference between the two similarity metrics is very small, leading to instability at line 169. RAIVE executes both branches: lines 170-171 and the fall-through. At the join point 172, the vector for `Similarity` has trivial differences inside. Intuitively, since the two similarity metrics are very close, the update in the true branch has little effect.

Moreover, the integer variable `Position` has the same values in the two branches as the same object was identified in the two phases. As such, the instability is benign.

## 2.6 Summary

In this chapter, we propose a runtime technique `RAIVE` to improve the stability of data processing programs. `RAIVE` detects output variations caused by errors (both internal and external) in floating point computation. It transforms a floating point value to a vector of four values and encodes the presence of an error by injecting value differences into the vector. Error propagation and suppression are performed implicitly by vectorized floating point operations. Instability is detected by checking if all vector elements lead to the same discrete result at discrete factors. Evaluation shows that `RAIVE` can precisely identify output variations. Compared to the state-of-the-art, `RAIVE`'s overhead is 2.43 times lower, averaging 340%, and it has the new capability of reporting output variations.

Next, we will discuss a series of projects to improve parameter selection of data processing programs for better results.

### 3. WHITE-BOX PROGRAM TUNING

Starting from this chapter, we are focused on another perspective for better data processing, which is improving programs parameter selection. Many programs or algorithms are largely parameterized, especially those based on heuristics. The quality of the results depends on the parameter setting. Different inputs often have different optimal settings. Program tuning is hence of great importance. Existing tuning techniques treat the program as a black-box and hence cannot leverage the internal program states to achieve better tuning. In this chapter, we propose a white-box tuning technique that is implemented as a library. The user can compose complex program tuning tasks by adding a small number of library calls to the original program and providing a few callback functions. Our experiments on 13 widely-used real-world programs show that our technique substantially improves data processing results and outperforms OpenTuner, the state-of-the-art black-box tuning technique.

#### 3.1 Introduction

More and more highly parameterized programs or algorithms are being used to solve different problems. Their complexity is also growing at an enormous pace, involving more and more computation stages. A prominent challenge for using these programs or algorithms is that the user has to configure a set of parameters beforehand. More importantly, the optimal configuration is mostly dependent on the specific input. Different inputs require different configurations to achieve the optimal results.

For instance, the results of *K-means* [16], a popular data clustering algorithm, heavily depends upon the choice of parameter  $K$ . It specifies the number of clusters into which the user wants to partition the input data. A lot of research [23–27] has

aimed at automatically deriving the appropriate  $k$  value from the input. However, there is no general solution for finding  $K$ . Another example relates to object detection in satellite image processing [28]. The parametrized algorithm processes a large volume of images in a time unit to generate the detection results. However, the parameter configuration that yields the best results for one image may produce sub-optimal results for another image (e.g., missing objects and broken edges). Consider, **Canny** [29], one of the most widely used image processing algorithms that detect edges. It is a multi-staged algorithm with three important parameters upon which **Canny**'s results heavily depend. According to [30], each input image may require a specific parameter setting to produce the best edge detection result. Fig. 3.1 shows the results on two different images using **Canny**. The left two are the original images. The other images show the results from two respective parameter configurations. Observe that configuration (0.6, 0.5, 0.9) produces the better result for the airplane whereas configuration (1.8, 0.2, 0.7) produces the better result for the trashcan. Thus, automated parameter tuning becomes critical in data processing as manual tuning is not realistic.

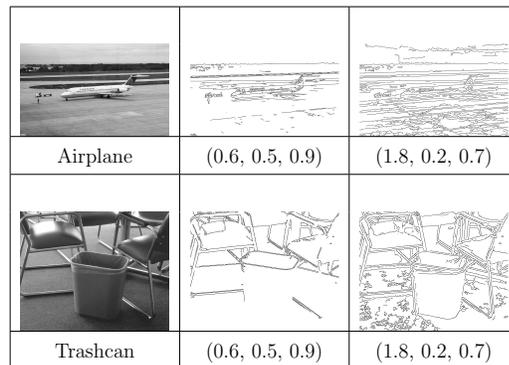


Fig. 3.1.: Canny's results with different parameters

### 3.1.1 Key Observation of Staged Computation Paradigm

By observing **Canny** and many other real world parameterized programs, we find that they typically follow the *staged computing paradigm*, i.e., they consist of multiple computation stages such that each stage has a unique set of tunable parameters.

### 3.1.2 Existing Work

Multiple frameworks were proposed to automate program tuning, among which OpenTuner [12] is the state-of-the-art. Oblivious of the staged computation paradigm, these frameworks treat the computation as a black-box. Guided by a user-provided scoring function of the final result, they sample the parameter space to find the best parameter configuration. Internally, they adopt stochastic algorithms [13, 14] or genetic algorithms [15] as the search strategy. While the above frameworks have achieved a certain level of success, they suffer greatly from poor performance due to the *inherent* limitations of the *black-box* designs:

- *All* parameters need to be tuned and set in each configuration, leading to an *exponential* number of configurations.
- A *full* execution accounts for the sampling of a *single* parameter configuration. Note that the full execution typically needs to load a large corpus of data and conduct lengthy preprocessing, which are very time consuming.

### 3.1.3 Our Work

In this paper, we propose a novel white-box tuning framework called WBTUNER. It is aware of the staged computation paradigm and tunes each stage independently. Specifically, WBTUNER spawns multiple processes to sample different parameter configurations involved in a stage. At the end of each stage, it aggregates the sampled internal results of that stage through a default or custom aggregation strategy. The

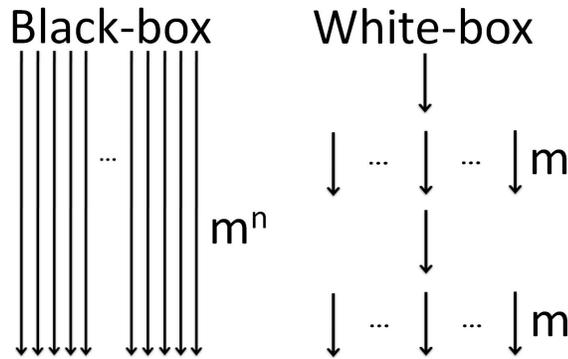


Fig. 3.2.: Execution models of black-box and white-box tuning

aggregation step reduces the spawned processes to fewer or one process (with desirable internal results achieved by tuning), which will proceed to tune the next stage. Intuitively, the aggregation strategy may either select the min/max value from the internal results (from various processes) or take the average value (Sec. 3.4.3).

Consider an application with  $n$  stages of computation, each stage having a unique parameter to tune. The parameter domain has  $m$  unique values. Initially, WBTUNER spawns  $m$  sampling processes to cover the  $m$  configurations of the first stage. Assume only one of the processes in a stage is selected to proceed to the next stage (after aggregation). WBTUNER needs only  $m$  sampling processes in each stage. Overall, WBTUNER only needs to cover  $m * n$  configurations and achieves so with a single full execution that keeps at most  $m$  live processes in any stage. Comparatively, OpenTuner needs to cover the  $m^n$  unique parameter configurations with  $m^n$  full execution instances. Fig. 3.2 illustrates the comparison.

### 3.1.4 Properties

WBTUNER features the following properties.

- By leveraging the independence between stages, WBTUNER needs to sample much fewer parameter configurations than OpenTuner. In the above example,

it needs to sample only  $m * n$  configurations, while OpenTuner needs to sample  $m^n$  configurations.

- Wasteful computation caused by poor internal results in an early stage can be terminated through the aggregation step for efficiency/efficacy. This is infeasible in black-box tuning.
- A full execution is reused for sampling different configurations and tuning different stages. Through the reused execution, WBTUNER greatly reduces the number of full execution instances needed. Note that every full execution may need to load and pre-process a large corpus of data, which only have to be done once in WBTUNER.

### 3.1.5 Contributions

The following shows our key contributions.

- We propose a novel white-box tuning technique. As discussed above, it features a set of salient properties compared to the existing black-box tuning.
- We develop a prototype WBTUNER in the form of a library that offers the users flexible access to internal program states. The realization of the library incurs great challenges related to process management and data store management. Our technique addresses these problems through a novel runtime transparently to the end users. Besides, we formalize the semantics of the runtime execution.
- We release our implementation of WBTUNER for the community at [31]. We use WBTUNER to tune 12 widely used parameterized programs. Our experiments show that WBTUNER substantially improves their results with reasonable overhead. The comparison with OpenTuner shows that OpenTuner takes 3.08X time to achieve the same results under a single core environment and 4.67X when multiple cores are used.

- We use WBTUNER to tune the parameters of a complex drone controller software (278K LOC) to mimic the behavior of a different controller with a better configuration (Sec. 3.5.2). Changing the configurations manually is infeasible because the numbers of parameters are large (612 and 426 respectively) and the meanings of these parameters are quite different between the two controllers.

## 3.2 Overview of White-Box Tuning Framework

We present the interface and show how to use it to tune **Canny**, a popular image processing algorithm.

### 3.2.1 User Interface

WBTUNER provides the user with an intuitive interface, which consists of multiple tuning primitives shown in Fig. 5.2. They are essentially library calls in the same programming language as the original program, rather than annotations in some specification language. We use the following *Canny* example to intuitively explain how to use the interface.

<p><i>Library Calls :</i></p> <p><b>@sampling</b>(<math>n, cbStrgy</math>)  </p> <p><b>@aggregate</b>(<math>x, cbAggr</math>)  </p> <p><b>@sample</b>(<math>x, cbDist</math>)   <b>@expose</b>(<math>x</math>)  </p> <p><b>@load</b>(<math>x</math>)   <b>@loadS</b>(<math>x, i</math>)   <b>@split</b>()  </p> <p><b>@sync</b>(<math>cbBarrier</math>)   <b>@check</b>(<math>cbChk</math>)</p> <p><i>CallBack :</i>     <math>cbStrgy, cbAggr, cbDist, cbChk, cbBarrier</math></p>
---

Fig. 3.3.: Primitives

### 3.2.2 Running Example

**Canny** has four stages: the *Gaussian smoothing* stage (line 22 in Fig. 5.3) which removes noise from the input image, the *image transformation* stage (line 30) which performs non-maximal suppression, the *edge traversal* stage (line 37) which leverages the hysteresis analysis to track all potential edges in the image, and the *visualization* stage which visualizes the final results.

It takes three parameters: `sigma`, `low`, and `high`. Specifically, the Gaussian smoothing stage relies on the parameter `sigma` and the edge traversal stage relies on the `low` and `high` thresholds. Based on our observation, **Canny** is a representative of real world data processing applications, which usually follow the *staged computing paradigm*, i.e., they consist of multiple computation stages such that each stage has a unique set of tunable parameters.

Fig. 5.3 shows how the interface is used (symbol @ is replaced with `wbt_`). Primitive `wbt_sampling()` (line 20) denotes the start of a sampling code region. It specifies the number of samples that should be collected within this region and a callback function that implements a sampling strategy. WBTUNER has a few built-in callbacks including `random` in this example. Primitive `wbt_aggregate()` (line 27) marks the end of a sampling region. It specifies a callback function (e.g., `AggregateGaussian()`) that aggregates the values of `sImage` across sample runs. Primitive `wbt_sample()` (line 21) indicates that a program variable, e.g., `sigma`, is a variable to tune (sample). It also specifies the distribution of the variable from which sample values are taken.

A callback function `AggregateGaussian()` is provided by the user to facilitate tuning. In this example, we implement it following an existing approach [32] to prune the poorly smoothed ones. Specifically, it loads (line 6) the images denoted by `sImage` which are computed according to different sampled values of `sigma` and determines (line 7) whether each image is properly smoothed given the image size `imgSize`. We will explain the relevant primitives `wbt_load()`, `wbt_loadS()` and `wbt_expose()` in Section 3.3.1. For each properly smoothed image, a new process is spawned by the

primitive `wbt_split` (line 9) to continue to tune `low` and `high` in the edge traversal stage (lines 34-41), while preserving the sampled `sigma` value and the produced image, i.e., `sImage`. Next we will discuss the execution model that underlies the user interface.

```

1 /* User provided callback */
2 void AggregateGaussian() {
3   for (int i=0; i<sampleNum; i++) {
4     // Prune smdImage if over/under smoothed
5     // Spawn a new process if not pruned
6     result = wbt_loadS(sImage, i);
7     if (properSmooth(result, wbt_load(imgSize))) {
8       memcpy(sImage, result, wbt_load(imgSize));
9       wbt_split();
10    }
11  }
12 }
13 /* Source program */
14 void canny() {
15   // Initializations
16   image = input(file);
17
18   /* STAGE ONE */
19   // Begin uniform sampling in 0.1 <= sigma <= 10
20   wbt_sampling(600, random);
21   sigma = wbt_sample(uniform(0.1,10));
22   sImage = gaussianSmooth(image, sigma);
23
24   //End sampling and Spawn new processes for those
25   //samples with appropriately smoothed images
26   wbt_expose(imgSize);
27   wbt_aggregate(sImage, AggregateGaussian);
28
29   /* STAGE TWO */
30   // Gradient computation and suppression
31
32   /* STAGE THREE */
33   // Begin sampling, 0.1 <= low, high <= 1
34   wbt_sampling(20, random);
35   low = wbt_sample(uniform(0.1,1));
36   high = wbt_sample(uniform(0.1,1));
37   finalImage = hysteresis(low, high);
38
39   // End sampling and aggregate finalImage from
40   // each sample to do majority vote
41   wbt_aggregate(finalImage, MajorityVote);
42
43   /* STAGE FOUR */
44   visualize(finalImage);
45 }

```

Fig. 3.4.: White-box tuning for Canny. The highlighted statements are added. Tuning primitives start with `wbt`.

### 3.2.3 Runtime Execution Model

The runtime execution framework is shown in Fig. 3.5. Initially, the original main process executes normally until it reaches the start of a tuning region (①). At this point, its role is switched to a *tuning process*. Intuitively, a tuning process is the “manager” of a pool of *sampling processes* that it spawns. A sampling process is the “worker” that conducts the computation within the region, and emits its result at the end of the region. The tuning process invokes the *sampling driver* (②) to spawn

a pool of child sampling processes (③). The driver determines how many sampling processes to be spawned and exercises a given sampling strategy. In some cases, the sampling strategy is feedback driven and relies on previous tuning results.

After spawning, the tuning process pauses. The sampling processes carry out the computation within the tuning code region (④), orchestrated by a scheduler (Sec. 3.3.2). When a sampling process encounters a *tuning variable* (i.e., X), it acquires a sample value from the variable's distribution. The sampling processes have different states afterwards. Upon reaching the end of the tuning region, a sampling process calls the *child aggregation driver* (⑤) to commit its own computation result from the *sample result variable* (i.e., Y) and terminates. Note that although the sampling process also calls the primitive *wbt\_aggregate()*, it only submits its sampling outcome. After all sampling processes commit, the tuning process resumes and invokes the *parent aggregation driver* to aggregate the sampling results (⑥). It then continues to execute normally with the aggregated results (⑦).

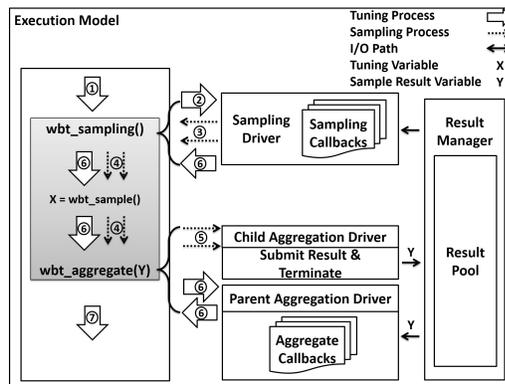


Fig. 3.5.: Execution Model

The above simplified model assumes a single tuning process in the runtime system. It is usually necessary to have multiple tuning processes. For example, consider the aggregation at line 27 in Fig. 5.3, the user may want to spawn multiple (independent) tuning processes each continuing with one from a subset of good internal results, i.e., properly smoothed images referred to by `sImage`, rather than a single

tuning process that continues with exactly one internal result. To achieve this, the user can use primitive *wbt\_split()* (line 9) to explicitly spawn a new tuning process (not sampling process) if the image is properly smoothed (line 7). Our runtime system fully supports multiple tuning processes (Section 3.3.1).

### 3.2.4 Result and Comparison

Initially we use 200 samples (line 20). At the end of the Gaussian smoothing stage (line 27), the invoked function `AggregateGaussian()` prunes 78 samples that are not properly smoothed, and keeps 122 samples. WBTUNER further spawns a tuning process for each remaining sample. When each of these processes reaches the edge traversal stage (line 34), it triggers a new sampling procedure which explores 90 samples (with different configurations of the parameters `low` and `high`) for each smoothed image. Hence the total number of samples is  $122 \times 90 = 10980$ . Fig. 3.6 shows the tuning model of WBTUNER for `Canny`.

The sampling results are aggregated by majority voting (line 41), that is, a pixel is set if it is set in the majority of sample runs. WBTUNER supports voting by default. Hence, the user can aggregate results through one line of function call. Finally, the aggregated image is visualized at line 44.

For comparison, we also apply OpenTuner to tune `Canny` with its default search strategy (i.e., Multi-armed bandit). Since no algorithm exists for computing a score for the output quality, we use simple heuristics to determine the poor samples, such as those that have very few or too many pixels in the final image. We use the execution time of WBTUNER as the timeout for OpenTuner. The images generated by OpenTuner through its sampling runs are aggregated by the same voting procedure in WBTUNER.

The tuning results for the coffeemaker image are shown in Fig. 3.7. Observe that WBTUNER spent 90 seconds on 9040 samples whereas OpenTuner can only finish 842 samples within the same amount of time, because most of its computation time was

spent on the expensive image loading, Gaussian smoothing, and gradient computation stages as it has to repeat such computation for each sample run. In addition to the visual result, we use the SSIM score [33] to compare the result with the ground truth result hand-picked by experts [30]. Both visual and scoring results demonstrate that that WBTUNER outperforms OpenTuner.

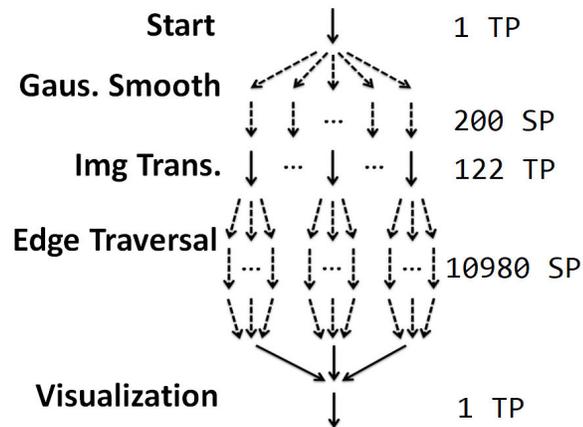


Fig. 3.6.: Tuning Canny. TP/SP are tuning/sampling processes.

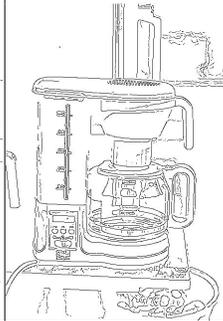
			
Origin	Ground Truth	OpenTuner	WBTUNER
samples	-	842	10980
SSIM	1	0.592	0.794

Fig. 3.7.: Tuning Canny with image coffeemaker in 90s.

### 3.3 Execution Model: Semantics and System

In order to achieve white-box tuning, we need to overcome a number of prominent challenges related to the management of *stores* and *processes*. First, an original process will spawn many sampling processes, which may need to be terminated (if the sampling result is poor), communicate with each other, further spawn their own child sampling processes, and join at specific execution points. Second, as the sampling processes produce a lot of sample data from internal states, managing such data (i.e., storing, accessing, and aggregating results across processes) is also challenging. All these complexities should be transparent to the users. In Section 3.3.1, we describe the formal execution model of our runtime system with the operational semantics. In Section 3.3.2, we present the implementation details of our runtime system.

#### 3.3.1 Semantics

DEFINITIONS: $Store \ \sigma ::= Var \rightarrow Value \quad SmpStore \ \delta ::= Var \rightarrow Value \mid Var \rightarrow (Index \rightarrow Value) \quad Mode \ \omega ::= T(pid) \mid S(pid)$ $Stmnt \ s ::= \dots \mid \mathbf{spawn}(\sigma, \delta, \omega, s) \mid \mathbf{notify}(pid) \mid \mathbf{wait}(pid) \mid \mathbf{invoke}(cb)$	
STATEMENT RULES: $\boxed{\sigma, \delta, \omega : s \xrightarrow{s} \sigma', \delta', \omega', s'}$ <span style="float: right;">Let <math>CPID = \{Child \ Process \ ID\}</math>, <math>PPID = Parent \ Process \ ID</math> in the following rules:</span>	
$\sigma, \delta, \omega : x := v$	$\xrightarrow{s} \sigma[x \mapsto v], \delta, \omega, \mathbf{skip}$ <span style="float: right;">[ASSIGN]</span>
$\sigma, \delta, T(pid) : @\mathbf{sampling}(n, cbStrgy); s$	$\xrightarrow{s} \sigma, \delta, T(pid), \forall i \in [1, n], \mathbf{spawn}(\sigma, \delta, S(i), \mathbf{invoke}(cbStrgy); s); \mathbf{invoke}(cbStrgy); s$ <span style="float: right;">[SAMPLING]</span>
$\sigma, \delta, T(pid) : @\mathbf{aggregate}(x, cbAggr); s$	$\xrightarrow{s} \sigma, \delta, T(pid), \mathbf{invoke}(cbAggr, x); s$ <span style="float: right;">[AGGR - T]</span>
$\sigma, \delta, S(pid) : @\mathbf{aggregate}(x, cbAggr); s$	$\xrightarrow{s} \sigma, \delta[x[pid] \mapsto \sigma(x)], S(pid), \mathbf{skip}$ <span style="float: right;">[AGGR - S]</span>
$\sigma, \delta, S(pid) : @\mathbf{sample}(x, cbDist); s$	$\xrightarrow{s} \sigma, \delta, S(pid), x := \mathbf{invoke}(cbDist); s$ <span style="float: right;">[SAMPLE]</span>
$\sigma, \delta, T(pid) : @\mathbf{split}(); s$	$\xrightarrow{s} \sigma, \delta, T(pid), \mathbf{spawn}(\sigma, \{\}, T(\mathbf{newPid}()), s); s$ <span style="float: right;">[SPLIT]</span>
$\sigma, \delta, T(pid) : @\mathbf{sync}(cbBarrier); s$	$\xrightarrow{s} \sigma, \delta, T(pid), \forall i \in CPID, \mathbf{wait}(i); \mathbf{invoke}(cbBarrier); \forall i \in CPID, \mathbf{notify}(i); s$ <span style="float: right;">[SYNC - T]</span>
$\sigma, \delta, S(pid) : @\mathbf{sync}(cbBarrier); s$	$\xrightarrow{s} \sigma, \delta \mapsto \sigma(x), S(pid), \mathbf{notify}(PPID), \mathbf{wait}(PPID); s$ <span style="float: right;">[SYNC - S]</span>
$\sigma, \delta, S(pid) : @\mathbf{check}(cbChk); s$	$\xrightarrow{s} \sigma, \delta, S(pid), \mathbf{if} \ \mathbf{invoke}(cbChk) \equiv \mathbf{true} \ \mathbf{then} \ s \ \mathbf{else} \ \mathbf{skip}$ <span style="float: right;">[CHECK]</span>
$\sigma, \delta, T(pid) : @\mathbf{expose}(x); s$	$\xrightarrow{s} \sigma, \delta[x \mapsto \sigma(x)], T(pid), s$ <span style="float: right;">[EXPOSE]</span>
$\sigma, \delta, T(pid) : y = @\mathbf{load}(x); s$	$\xrightarrow{s} \sigma[y \mapsto \delta(x)], \delta, T(pid), s$ <span style="float: right;">[LOAD]</span>
$\sigma, \delta, T(pid) : y = @\mathbf{loadS}(x, i); s$	$\xrightarrow{s} \sigma[y \mapsto \delta(x)[i]], \delta, T(pid), s$ <span style="float: right;">[LOADSAMPLE]</span>

Fig. 3.8.: Operational Semantics

The semantics are presented in Fig. 3.8. The related definitions are presented at the top of the figure.

## Stores

WBTUNER has two stores, the *store*  $\sigma$  for original program states and the *sample store*  $\delta$  that is shared across all processes to store sampling outputs. The two are isolated. State transferring between the two are performed explicitly by the programmer. In  $\delta$ , states can be further divided into two classes: (1) **exposed store**, a store for exposed variables, (2) **aggregation store**, a store for sampled results from the child sampling processes.

**Exposed Store** Exposed store is a mapping from variables to values. A local variable is exposed by the primitive *wbt\_expose()*. The exposed local variable is saved to the exposed store and can be retrieved with the primitive *wbt\_load()*. Different from common local variables, the exposed local variable is available outside its local scope (e.g., function). Thus, the exposed local variable can be used to pass the value across different scopes. For instance, in Fig. 5.3, the local variable `imgSize` from the `canny` function is exposed at line 26 and then loaded at line 7 in the `AggregateGaussian` function.

**Aggregation Store** Aggregation store of a tuning process stores the sampled outcomes. It maps each program variable  $x$  to a vector  $\delta(x)$ , of which the  $i$ th entry holds the value of the variable from the  $i$ th child process. Note that vector abstracts the mapping from index to values. At the semantic level, the primitive *wbt\_aggregate(x, ...)* forces each child sampling process to write/commit the value of  $x$  from its regular store to the aggregation store of the parent tuning process, as illustrated by line 27 in Fig. 5.3. The primitive *wbt\_loadS(x, i)* loads the value of  $x$  from the  $i$ th child process, as illustrated by line 6 in Fig. 5.3.

## Processes

WBTUNER supports two execution modes,  $\mathbb{T}\langle pid \rangle$  denotes the a tuning process and  $\mathbb{S}\langle pid \rangle$  is a sampling process.  $pid$  denotes the process id. To facilitate discussion, we extend the statements to include a **spawn**( $\sigma, \delta, \omega, s$ ) statement that forks a process with the specified stores, execution mode, and the process body  $s$ , a **notify**( $pid$ ) statement that notifies a process  $pid$ , a **wait**( $pid$ ) statement that waits for a notification from the process  $pid$ , and an **invoke**(**cb**) statement that invokes a callback function  $cb$ .

## Statement Rules

Rule [*SAMPLING*] forks  $n$  sampling processes (indicated by the  $\mathbb{S}\langle i \rangle$  mode) through the **spawn**() primitive. Observe that the last parameter of the primitive is the body of the child process, which contains the same statements as the parent, namely, “**invoke**( $cbStrgy$ );  $s$ ”. After forking, callback  $cbStrgy()$  is called to initialize the sampling strategy in the children. Note that Rule [*SAMPLING*] only applies in a tuning process. It is a NOP in a sampling process.

Rule [*AGGR-T*] specifies that a tuning process invokes the callback  $cbAggr()$  to aggregate the sampling results for variable  $x$ . In the callback, the user can implement various aggregation strategies. For example, the values of sample target variable  $x$  from all sample processes can be averaged and written back to  $x$  in the tuning process, which can proceed with the aggregated value. In contrast, Rule [*AGGR-S*] specifies that upon aggregation, a sampling process stores its sampling outcome of  $x$  to the element of the sampling vector corresponding to the process id and then terminates. Recall that only the tuning process aggregates results and sampling processes only produce results.

Rule [*SAMPLE*] only applies to sampling processes. It specifies that the callback  $cbDist()$  is invoked to acquire a sample value for variable  $x$ , which denotes a parameter to tune. Rule [*SPLIT*] specifies that a tuning process can explicitly spawn a child

tuning process. The child process is for tuning the next phase. Function `newPid()` returns a new *pid*. The child process inherits the regular store but not the sample store from the parent. Rule `[SYNC-T]` indicates that the tuning process waits for all the child sampling processes to reach the barrier, and then it invokes `cbBarrier()` to perform some operations that access results across multiple sample runs. After that, the tuning process notifies all its child sampling processes to proceed. Compared to `@aggregate`, `@sync` is usually used in the middle of a sampling region. Rule `[SYNC-S]` specifies that a sampling process notifies its parent tuning process after it has reached the barrier. It then waits for the tuning process to finish the callback and notify it to proceed. Notifications from child processes are queued to avoid message lost which may lead to deadlocks.

Rule `[CHECK]` specifies that a sampling process invokes a callback `cbChk()` to check its local states. If the check returns *false*, the sampling process is terminated. This feature allows us to terminate useless sample runs long before they get to the aggregation point (e.g., `k-means` in Sec. 3.5.2), which improves not only the performance but also the final results. Note that such improvements are impossible to achieve in black-box tuning.

Rule `[EXPOSE]` exposes the value of  $x$  from the regular store to the sample store, which is accessed by tuning callbacks. The rule only applies to tuning processes. Observe that it allows callbacks to access program variables outside their scopes. Rule `[LOAD]` loads an exposed variable  $x$  (from the exposed store of  $\delta$ ) inside some callback function in a tuning process. Rule `[LOADSAMPLE]` loads the  $i$ th sample outcome of  $x$  (from the aggregation store of  $\delta$ ).

### 3.3.2 WBTUNER Runtime System

We present the implementation details of WBTUNER runtime by following the same structure as Section 3.3.1.

## Stores

**Exposed Store** We implemented the exposed store as follows. Our system encodes a local variable with its name and its scope information (e.g., the function name) before mapping it to the value in the exposed store. Similarly, our system uses the name and the scope information of a variable to retrieve the associated value. The encoding guarantees we can access the value of the exposed variable throughout the whole execution. Note that the scope information is required to distinguish the local variables with the same name from different scopes.

**Aggregation Store** Our system achieves the semantics by leveraging the file system in disk. In particular, all sampled outcomes (WBTUNER supports multiple sample result variables aggregation) of a sampling process are stored in a file. The file name is in the form *pid*, which specifies the sampling process that submits the results of the variables. All the files are stored in a directory owned by the tuning process. To load the *i*th outcome of *x* from disk, our system searches in the directory owned by the tuning process for the related file based on the information in primitive *wbt\_loadS(x, i)*.

## Processes

**Process Scheduling** In practice there will be large number of tuning and sampling processes executing concurrently at runtime. Thus, WBTUNER provides a scheduler to manage the creation and termination of processes. It prevents excessive process creation for better performance (Fig. 3.10). Using a uniform process pool is not optimal because of the difference between the two kinds of processes (tuning and sampling). Instead, we prioritize a sampling process over a tuning process because the former conducts the real computation. In addition, we want to finish all the sampling processes belonging to a tuning process as soon as possible so that the tuning process can finish its work and yield the resource.

The scheduler works as follows. Upon a spawn request, it checks if there are enough resources. If not, the current process is put in a priority queue. Upon a process termination event, the highest priority process in the queue is woken up.

Algorithm 1 shows the details. The *Schedule* procedure is called with *pid* (i.e., process id), *event* and *todo*. There are three possible events: *SPAWN\_S* (i.e., spawning a sampling process), *SPAWN\_T* (i.e., spawning a tuning process), and *EXIT*. The parameter *todo* denotes the number of samples remained for the current (tuning) process. Sampling processes are ordered inside the priority queue based on the *todo* values of their parent tuning processes. Lines 2-7 correspond to process termination, which wakes up the process with the highest priority. At line 8, the computed threshold denotes the remaining resources. If the number of available resources is below it, the current process will be put back into the priority queue (lines 9-12). Otherwise, it is allowed to proceed (line 14). Since real tuning is done by sampling processes, the threshold is always 0 for sampling processes so that they don't have to wait if there is any available resource. A configurable variable is used to prevent spawning too many tuning processes because they would inevitably lead to decreasing the tuning efficiency. In Algorithm 1, we set the configurable threshold of tuning process to 75% (i.e., it has to wait if 25% processes are occupied).

For benchmarks requiring a large number of samples and consuming lots of memory (e.g., *Canny*), the scheduler limits the number of concurrent samples and reduces the memory consumption and execution time significantly (Fig. 3.10). Too much memory consumption will result in excessive page fault which degrades the runtime performance.

## 3.4 Practical Challenges

### 3.4.1 Overfitting

Since machine learning algorithms normally produce models as their output, the tuning task of these hyper-parameters is usually guided by the execution results of

the models (e.g., lower classification errors). However, it might lead to *overfitting*, meaning that the model with the tuned hyper-parameters produces optimal results with training data but poor results with testing data. Note that programs such as **Canny** do not have this problem as they are tuning for the final result but not models tested by new data. Cross-validation can help to mitigate such hyper-parameters tuning problem according to existing studies [34, 35]. Specifically, it searches for the model hyper-parameters that generalize, rather than fitting to the training dataset. WBTUNER provides intrinsic support to address overfitting by combining its execution model with *k-fold cross-validation* [36], a widely used technique. Specifically, to tune the parameters in a machine learning algorithm, the user only indicates the  $k$  value in the `wbt_sampling()` primitive and provides a validation callback. WBTuner will then transparently include  $k$ -fold cross validation during its tuning process. The experimental results in Fig. 3.17 demonstrate the necessity of cross-validation.

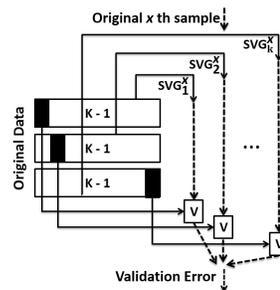


Fig. 3.9.: Tuning + Validation Execution Model

The tuning-validation model is shown in Fig. 3.9. First, the input data is transparently divided to  $k$  datasets for each sample run. Moreover, for the  $x$ th original sample run, WBTUNER spawns  $k - 1$  more processes, that form a *sampling and validation group* (SVG). If the user intends to collect  $n$  samples originally, WBTUNER internally creates  $n$  SVGs, that is,  $n * k$  processes. All the  $k$  processes in a SVG share the same sample values for the tuning variables but use different datasets for training and validation to prevent overfitting. As illustrated in the figure, the  $i$ th process in the SVG uses the  $i$ th dataset for validation and the remaining  $k - 1$  datasets

for training. At the end of the execution of an SVG process, WBTUNER invokes the user-supplied validation callback to apply the produced model on its validation dataset and computes the validation error. The validation errors from all SVG processes are then aggregated to drive the remaining steps of the tuning procedure. The experimental results in Fig. 3.17 demonstrate the necessity of cross-validation.

### 3.4.2 Incremental Aggregation

According to the execution model of WBTUNER, the sampling results are submitted by the sampling processes and aggregated by the tuning processes once the sampling is completed. However, the whole execution entails massive storage and I/O overhead. According to our observation, aggregation can be performed incrementally in many benchmark programs as their aggregations involve functions such as finding the min, max, average, or majority. For instance, for the aggregation strategy min/max, each sampling process updates a shared global variable min/max by comparing its local outcome with the global variable. For incremental averaging, WBTUNER uses a shared ring buffer to which sampling processes copy their results. The tuning process then consumes the data from the buffer to perform incremental averaging. Majority voting is handled in a similar fashion. In our experiment section (Fig. 3.10), we will show that incremental aggregation substantially reduces the tuning time and memory consumption.

### 3.4.3 Sampling/Aggregation Strategies

In addition to custom strategies provided by the user, WBTUNER supports several common sampling/aggregation strategies by default. The user only needs to indicate the strategy name inside the *wbt\_sampling/wbt\_aggregate* primitive to use it. Currently, the supported sampling strategies are random (RAND) and Markov Chain Monte Carlo (MCMC). For aggregation strategies, WBTUNER supports min, max, majority vote (MV), averaging (AVG), and duplicate elimination (DEDUP). Based

Table 3.1.: Benchmark statistics and the experiment results for achieving the best tuning scores.

Program	LOC	#P	#PR	Sampling	Aggregation	Ext LOC	Single Core						Multi Core									
							Native		WBTuner		OpenTuner		o/h(x)		Native		WBTuner		OpenTuner		o/h(x)	
							time(s)	Score	time(s)	Score	time(s)	Score	OT/WB	Score	time(s)	Score	time(s)	Score	OT/WB	Score	time(s)	Score
[29] ↑ Canny <sup>1</sup>	1.1k	3	8	RAND	CUSTOM/MV	151	0.159	0.29	51.53	0.636	t/o <sup>2</sup>	0.44	-	0.061	17.75	0.636	t/o	0.44	-			
[38] ↑ Watershed <sup>1</sup>	270k	3	5	RAND	MV	34	1.03	0.41	26.1	0.65	31.5	0.65	2.11	0.93	56.1	0.65	221.59	0.65	3.95			
[16] ↑ Kmeans	1.2k	1	5	MCMC	MAX	56	0.165	0.46	1.57	0.523	9.7	0.523	5.79	0.057	0.56	0.523	2.49	0.523	4.45			
[39] ↑ DBScan	908	2	7	MCMC	MAX	80	0.657	0.299	25.41	0.502	124.08	0.502	3.21	0.021	2.94	0.502	15.7	0.502	5.34			
[40] ↓ Face Rec	9.6k	3	7	RAND	MIN	92	4.788	17	578.62	7.3	1203.25	7.3	2.07	4.6	33.47	7.3	684.12	7.3	4.44			
[41] ↑ Speech Rec <sup>1</sup>	19.8k	16	18	RAND	MV	89	4.263	1	313.25	5	t/o	4.2	-	4.12	19.54	5	t/o	4.2	-			
[42] ↓ Plylip	12.6k	4	12	RAND	DEDUP/MIN	95	4.67	20.4	1021.4	0.84	1910.23	0.84	1.87	2.4	211.15	0.84	693.21	0.84	3.28			
[43] ↑ FASTA	77.5k	2	4	RAND	CUSTOM	108	0.12	40	1.56	523	4.91	523	3.54	0.02	0.25	523	t/o	461	-			
[44] ↑ TOPN Rec	33.5k	3	5	RAND	MAX	3	6.16	0.1	273.45	0.126	560.5	0.126	3.04	5.9	81.2	0.126	513.1	0.126	6.32			
[45] ↓ METIS	44.3k	3	5	RAND	MAX	30	0.16	6952	4.77	6706	20.57	6706	4.31	0.06	1.2	6706	7.34	6717.7	6.12			
[46] ↓ C4.5	17.8k	2	4	RAND+CV	MIN	58	0.059	2.46	7.23	0.082	21.54	0.082	3.18	0.036	1.68	0.082	6.54	0.082	3.89			
[47] ↓ SVM	11.3k	8	10	RAND+CV	MIN	44	6.172	87	233.72	9.5	438.12	9.5	1.96	5.314	66.98	9.5	288.23	9.5	4.3			
[48] ↓ Ardupilot	278k	40	44	RAND	CUSTOM	204	-	1954k	-	-	-	-	-	192.3	151k	1074k	-	-	-			

↑: Higher scores are better; ↓: lower scores are better.

1. These benchmarks do not have default scoring functions.

2. "t/o" means OpenTuner cannot achieve the similar score (i.e., difference < 10%) of WBTUNER.

on our experience, these strategies are usually sufficient for most of the tuning tasks. Observe that only four benchmarks (out of 13 benchmarks) use custom aggregation strategies in our experiments.

### 3.4.4 Auto-tuning Sampling Number

Since the number of samples varies from one tuning region to another, WBTUNER provides an automatic way similar to exponential backoff [37] to determine the optimal number of samples. For the sampling number of each primitive `wbt_sampling()`, WBTUNER doubles it and compares the aggregated results between the samples from original set and the samples from doubled set according to the scoring function. If the doubled result is better, the number of samples is doubled again until the no further improvements.

---

**Algorithm 1** Process Scheduling
 

---

```

1: procedure SCHEDULE(pid, event, todo)
2:   if event = EXIT then
3:     poolSize  $\leftarrow$  poolSize + 1
4:     if PQueue not EMPTY then
5:       p  $\leftarrow$  PopPQueue()
6:       signal(p.pid)
7:     return
8:   threshold  $\leftarrow$  (event = SPAWN_S) ?
       0 : MAX_POOL_SIZE  $\times$  0.75
9:   while poolSize  $\leq$  threshold do
10:    PushPQueue(new P(pid, event, todo))
11:    poolSize  $\leftarrow$  poolSize + 1
12:    wait()
13:    poolSize  $\leftarrow$  poolSize - 1
14:  poolSize  $\leftarrow$  poolSize - 1
15:  return

```

---

### 3.5 Evaluation

WBTUNER is implemented in C and publicly available at [31]. We evaluate the efficiency and effectiveness of WBTUNER and compare it with OpenTuner. Experiments were run on a machine with Intel i7-2640M 2.80GHz processor and 16GB RAM.

**Benchmarks.** We use a wide variety of C/C++ benchmarks in our experiments, including 12 widely used data processing programs and a complex open-source controller software for commercial drones. These are heavily parameterized applications.

All programs have multiple datasets that can be found online or come with the program. We have selected only the datasets that have the outcome ground truth for comparison. On average, we used 10 datasets for each program. The results are

summarized in Table 3.1. Benchmarks either come with their own scoring functions or use publicly available scoring functions, so the callbacks for them are implemented accordingly. Comparison results of benchmarks without scoring functions (i.e., with superscript 1 in Tab. 3.1) are explained in section 3.5.1.

Columns 1-2 show the programs names and the lines of code. Column 3 shows the number of tunable parameters and column 4 shows the number of WBTUNER primitives added to the source. The next two columns (5-6) describe the sampling and the aggregation strategies. Most programs use random sampling. `DBScan` and `K-means` demonstrate the use of a different sampling strategy (MCMC). `C4.5` and `SVM` use random sampling together with cross-validation, which is also implemented in OpenTuner for these two benchmarks (for comparison). Column 7 presents the LOC in tuning callback functions. Observe that the number of primitives is small, yet, it allows to represent complex tuning models as we will demonstrate in Section 3.5.2. The LOCs for callbacks are small compared to the source code LOCs. They mainly implement scoring functions or checks.

### 3.5.1 Tuning Results Summary

In the first experiment, we ran each benchmark with the largest dataset under three settings – (1) native run without tuning; (2) white-box tuning with WBTUNER; (3) black-box tuning with OpenTuner and its default search strategy (i.e., multi-armed bandit [49]).

We ran WBTUNER with the number of samples auto-tuned (Sec. 3.4.4) by WBTUNER until converging, then we collected the tuning time. For OpenTuner, we gradually increased the timeout parameter until it either reaches similar results as WBTUNER (difference  $< 10\%$ ) or could not reach similar results after spending 10 times WBTUNER’s tuning time. We measured the quality of the tuning results by comparing with the ground truth that comes with the datasets. Note that these ground truths are only used in measuring quality, *but not in tuning*. As stated before,

OpenTuner requires scoring functions to guide the search; however, a few benchmarks do not have a standard scoring function (marked with the superscript 1 in Table 3.1). To achieve fair comparison, for these benchmarks, we implemented the same domain-specific heuristics from WBTUNER in OpenTuner to distinguish good and bad samples, and to use the same aggregation method from WBTUNER to aggregate the good sample results. To quantify the results for these programs, we compute their scores based on the comparison with the ground truths. Such scores are not used in tuning.

Since OpenTuner does not support parallel sampling by default, which requires substantial engineering effort, we conducted the comparison in both single-core and multi-core. The single-core results are shown in columns 8-14 in Table 3.1, while the multi-core results are shown in columns 15-20. Columns 8 and 9 present the native execution time and the score without tuning. Note that for the programs with  $\uparrow$ , the higher the scores the better, and for the others with  $\downarrow$ , the lower the scores the better. Column 10 presents the tuning time of WBTUNER upon convergence. Column 11 shows the converged score. Column 12 shows the tuning time for OpenTuner. Those with “t/o” mean that those scores are apparently worse (difference  $> 10\%$ ) than WBTUNER after spending 10x more tuning time. Column 13 shows the final tuning score of OpenTuner. Column 14 shows the overhead comparison. Columns 15-20 are the results for multi-core.

Observe that for single-core environment, OpenTuner times out in 2 out of the 13 cases. For the other cases, the average tuning overhead of OpenTuner is 3.08X higher than WBTUNER. For multi-core environment, 3 cases time out and the overhead ratio is 4.67X.

Observe that WBTUNER substantially improves the results quality compared to those without any tuning. It is more effective than OpenTuner. For the cases that OpenTuner can reach the scores of WBTUNER, we also allow more tuning time; still, it did not produce better results.

Fig. 3.10 shows the effect of the optimizations discussed in Sec. 3.3.2 and 3.4.2. Observe that the incremental aggregation is highly effective for several cases, espe-

cially for reducing the memory usage as it prevents reading a large number of results for one-shot aggregation. Observe that the scheduler further improves the performance in several cases, especially Canny and K-means. Before optimization, Canny’s execution time and memory overhead are about 4X higher.

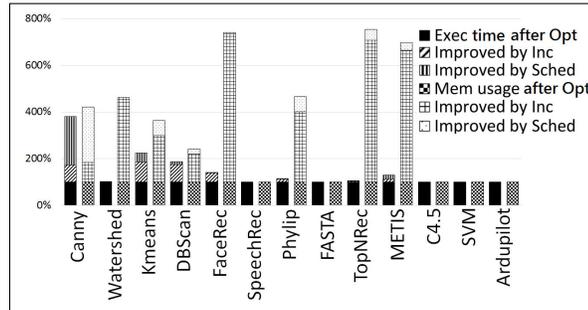


Fig. 3.10.: Optimization effects on different benchmarks

### 3.5.2 Tuning Case Studies

In this section, we study the details of tuning several representative programs under the single core environment and tuning Ardupilot (a drone controller) in the multi-core environment.

#### Image Processing

**Canny.** In Section 3.2, we have already shown the tuning results of **Canny**. Here we used 10 different images from [30], where each image has a ground truth result image hand-picked by experts.

Since no general scoring function exists, we use majority vote for results aggregation, meaning the result with the largest number of supports from the sample runs is reported. Then we use the SSIM [33] score to compare the voting result with the ground truth. The higher the score the better. We extended OpenTuner with the

majority voting capability to achieve fair comparison. For each image, we ran WBTUNER and OpenTuner 10 times and took the average. Fig. 3.11 shows the tuning score when WBTUNER converges, the corresponding OpenTuner score after it runs the same amount of time, and the score without tuning. Observe that WBTUNER almost always produces the best results.

On average, OpenTuner has 119% improvement over no-tuning, whereas the improvement of WBTUNER is 178%. The reason is that WBTUNER can prune a lot of sample runs that will not yield promising results after stage one ( Fig. 5.3).

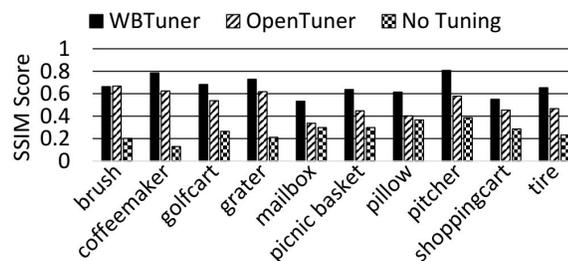


Fig. 3.11.: Canny tuning scores of 10 images.

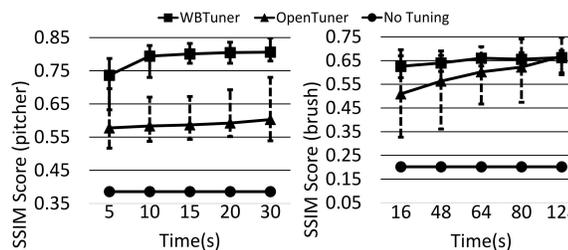


Fig. 3.12.: Canny tuning score variation

The score variation with the tuning time is shown in Fig. 3.12 for the **pitcher** and **brush** images, which represent the maximum and minimum improvement over OpenTuner, respectively. Observe that for **pitcher**, even 5-second tuning in WBTUNER yields much better results for 30 seconds tuning in OpenTuner. The visualization in Fig. 3.13 shows that the result by WBTUNER is very close to the ground truth but the result by OpenTuner is not. For **brush**, WBTUNER has a very close but lower

score at the end, although the two have very comparable performance all the time. Fig. 3.13 shows that the WBTUNER's result is not inferior.

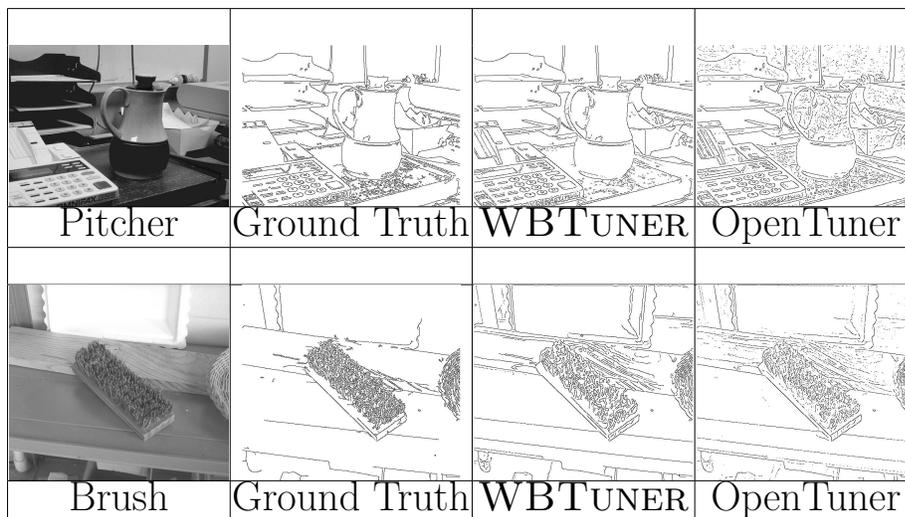


Fig. 3.13.: Canny results of WBTUNER and OpenTuner

## Bioinformatics

**Phylip.** Phylip [42, 50] generates the phylogenetic tree of given protein or DNA sequences by calculating the distances. It shows the evolutionary relationships between various biological species. Phylip consists of five stages of computation as shown in Fig. 3.14.

Stage 1 generates the transition probability matrix and has a tunable parameter `ease`. Stage 2 loads data and performs preprocessing. Stage 3 generates the distance matrix based on the transition probability matrix and the input. It has two tunable parameters `invarfrac` and `cvi`. Stage 4 initializes the phylogenetic tree. Stage 5 generates the tree based on the distance matrix from stage 3. It has a tunable parameter `power`. WBTUNER tunes stages 1, 3 and 5. The `wbt_aggregation()` primitive is called at the end of stages 1 and 3 with the duplicate-elimination (DEDUP) strat-

egy to prune the sample runs that have similar matrices. Thus, new tuning processes are only spawned for unique matrices. At the end of stage 5, the aggregation selects the tree with the lowest sum of squares, which is the default scoring function. Lower score means the better result.

```

1 void PhyloTreeGeneration() {
2   /* STAGE ONE */
3   wbt_sampling(numberOfSamples, random);
4   ease = wbt_sample(uniform(0.1,0.9));
5   maketrans();
6   qregigen(prob);
7   wbt_expose(probMatrixSize);
8   wbt_aggregate(probMatrix, DEDUP);
9
10  /* STAGE TWO: read in data and preprocessing */
11  /* STAGE THREE */
12  wbt_sampling(numberOfSamples, random);
13  cvi = wbt_sample(uniform(0.5,5));
14  invarfrac = wbt_sample(uniform(0.1,0.9));
15  makedists();
16  wbt_expose(distMatrixSize);
17  wbt_aggregate(distMatrix, DEDUP)
18
19  /* STAGE FOUR: phylogenetic tree initialization*/
20  /* STAGE FIVE */
21  wbt_sampling(numberOfSamples, random);
22  power = wbt_sample(uniform(1.5,2.5));
23  maketree();
24  wbt_aggregate(phyTree, MIN, scoring);
25  output(phyTree);
26}

```

Fig. 3.14.: White-box tuning for phylip tree

Fig. 3.15 shows tuning score comparison for ten datasets from [51] when WB-TUNER converges. Observe that tuning is critical for this program. On average, WBTUNER can reduce the errors by a factor of 283 when compared with no tuning, and by a factor of 4.77 when compared with OpenTuner.

Fig. 3.16 shows the tuning score variations over time for data2 and data10 that have the maximum and minimum improvement over OpenTuner, respectively. For

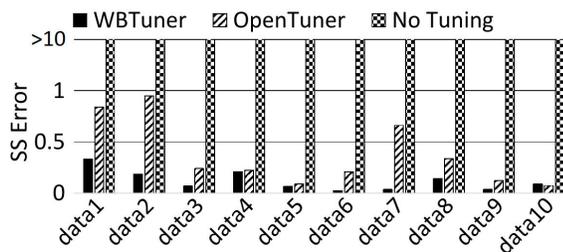


Fig. 3.15.: Phylip tree tuning scores on 10 datasets.

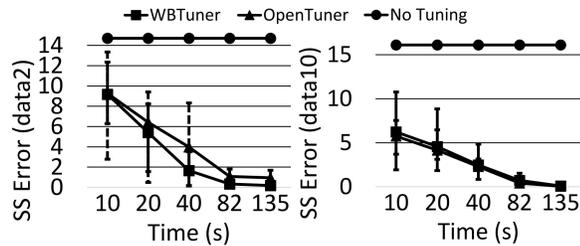


Fig. 3.16.: Phylyp tree tuning score variation

**data2**, 40 seconds of tuning in WBTUNER achieves a similar result as 135 seconds of tuning in OpenTuner. The improvement is achieved by the independent tuning/pruning in the three tuning regions. Although OpenTuner outperforms WBTUNER for **data10**, the difference between the two results is nearly invisible.

## Machine Learning

**Support Vector Machine (SVM).** SVM [47] is a popular machine learning algorithm for data classification. It is a supervised learning technique which takes the training data with feature class labels to build a model for classifying new data. We use the multi-class SVM [52] to classify data with multiple class labels. The algorithm has 8 tunable parameters, which lead to substantially different models if tuned differently. Furthermore, like most machine learning algorithms, certain parameter settings may lead to overfitting (Section 3.4.1). Thus, we leverage the  $k$ -fold cross-validation in WBTUNER to tune the parameters while preventing overfitting.

We compare the results tuned by WBTUNER with and without cross-validation for 10 datasets obtained from [53]. We divide each dataset into two equal sets and use the first half for training and tuning and the second half for testing. We then collect the results after both tuning converge. The results are depicted in Fig. 3.17. Observe that for the left two bars (without cross-validation), the training error (black bar) is close to zero while the testing error is very high, indicating overfitting. For the

right two bars (with cross-validation), the testing error is significantly lower than that without cross-validation, which strongly suggests that cross-validation substantially mitigates the overfitting problem. That is, the new model generalizes better from the training dataset, without being affected by its details and noise. The results strongly suggest that overfitting is a prominent challenge in tuning and WBTUNER effectively addresses this problem transparently.

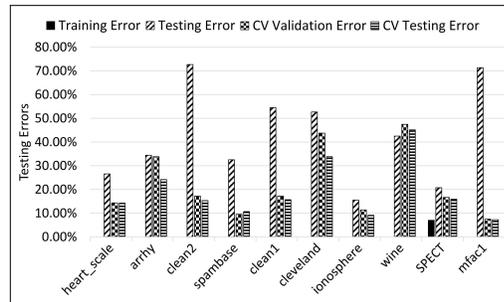


Fig. 3.17.: SVM tuning scores of 10 datasets w/wo validation

We also compare the result generated by WBTUNER and OpenTuner. As OpenTuner does not handle overfitting by default, we extended its implementation to provide cross-validation as well (using the same  $k$ ). Observe that WBTUNER consistently outperforms OpenTuner. The tuning improvement by OpenTuner over no-tuning is 35% whereas the improvement by WBTUNER is 47%. Fig. 3.19 shows the score variation for the best and the worst datasets. Observe that for Cleveland, even after 1500 seconds, OpenTuner cannot reach the result produced by WBTUNER within 80 seconds.

## Speech Recognition

**Sphinx.** Sphinx [54] is a popular speech recognition system. It takes a raw audio and a dictionary, and generates the script for the audio according to the dictionary. It has 16 tunable parameters, such as the upper and lower edges of filters, language

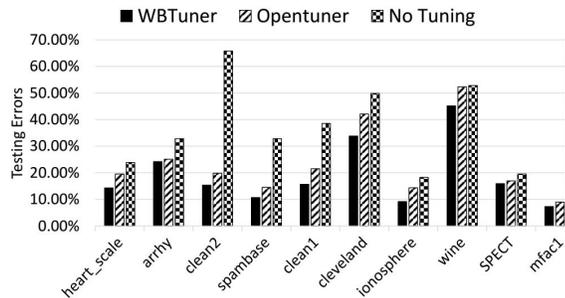


Fig. 3.18.: SVM tuning scores of 10 datasets

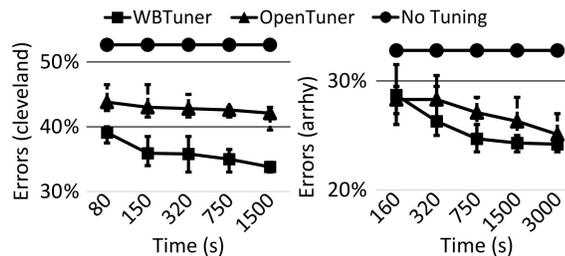


Fig. 3.19.: SVM tuning score variation

weight, and word insertion penalty. These parameters are critical to the recognition results. Different persons' audios may require different parameter sets. Since there does not exist a general scoring function, the tuning results are aggregated using majority vote. OpenTuner is also extended with the majority voting capability for fair comparison.

In the study, we took 10 sets of audios (for 10 persons) from the AN4 dataset [55], each set having 5 audios. We applied both WBTUNER and OpenTuner to all these 50 audios. Fig. 3.20 shows the recognition precision comparison when WBTUNER converges (i.e., the number of audios that are correctly recognized for each dataset). Observe that WBTUNER precisely recognizes all 5 audios for 6 out of 10 sets, and more than 4 audios for another 3 sets. To reduce non-determinism, we ran the experiment multiple times and took the average. Thus, there are some decimal numbers in the precision results. In contrast, **Sphinx** can only recognize 2.7 audios on average

without tuning, and 3.94 audios with OpenTuner. Fig. 3.21 shows the score variations for the best and worst data sets.

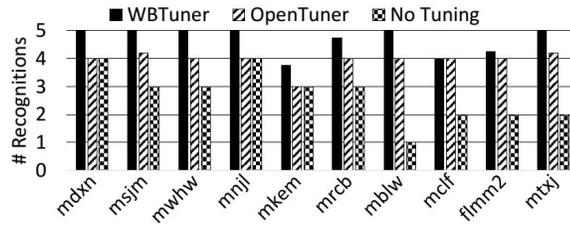


Fig. 3.20.: Sphinx tuning of 10 datasets

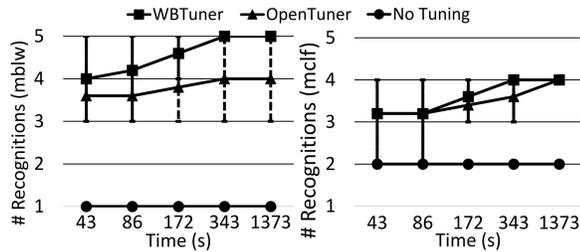


Fig. 3.21.: Sphinx tuning score variation

### Tuning Drone's Behavior.

Here we demonstrate how we can leverage WBTUNER to tune large and complex cyber-physical systems for behavior learning. Specifically, we aim to tune one drone's parameters so that it mimics the behavior of the other one.

We use two pieces of widely used drone control software: PX4 [56] and Ardupilot [48]. They are complex (385k and 278k LOC respectively), and have completely different features and implementations. Furthermore, PX4 has 426 configurable parameters and Ardupilot has 612 and the meanings of these parameters are quite different. Thus, High-end drones usually have parameter configurations enabling much better performance, as their engineers spent a lot of time in tuning. For example,

Ardupilot flies much slower than PX4 (25% slower) and has much higher battery consumption. PX4s controller is clearly out-performing Ardupilot. WBTuner allows Ardupilot to automatically learn from PX4, saving the substantial manual tuning efforts. Note that there is hardly any correspondence between parameters across the two systems so that one cannot simply copy parameter values. Moreover, only increasing the speed is suboptimal because there are other parameters to consider such as power consumption or way-point radius to prevent overshoot.

Although both PX4 and Ardupilot provide their own specific black-box parameter tuning tools [57,58], only a limited number of parameters can be tuned by these tools and thus cannot lead to optimal results. Furthermore, they cannot be applied to achieve more sophisticated tuning tasks such as behavior learning, which is a popular tendency for training autonomous vehicles with different purposes [59–61].

**Tuning Target.** We aim to tune the parameters of Ardupilot to make it learn the flying behavior of PX4. We identify 40 parameters that are most relevant to drone control in Ardupilot and mark them as the *tuning variables*. We use the motor speed variables as the *sample result variables* since the drone’s behavior is mostly determined by the speed of its four motors. We fly both Ardupilot and PX4 under the same mission, and then employ WBTUNER to tune the *tuning variables* in Ardupilot while learning from PX4’s flying behavior. Namely, we define the scoring function as the root-mean-square errors of the four motors speed between the two controllers. Furthermore, as a typical mission in Ardupilot often needs to execute under multiple flight modes (e.g., takeoff or land), we define the tuning regions as the individual mode control functions.

To tune Ardupilot according to PX4’s behavior, we first fly both Ardupilot and PX4 under 2 different missions. The first one consists of taking off, rising to 10 meters, and finally landing. The second mission makes the drone fly along a 45m route with 3 way points. Our experiments are conducted using the Gazebo simulator [62]. The first mission uses 2000 sample runs, while the second uses 6000 runs given its complexity, each taking 20-30 seconds. Overall, the tuning time is about 42 hours due to real-

time simulation. Then we test the subsequent performance of Ardupilot with the tuned parameters under a complex mission, where the drone zigzags and returns to the starting point with a flight distance of 165m.

Fig. 3.22 shows both the motors speed and visual results for the first tuning mission. Note that motor speeds represent the key states of a drone. The gray point in the visual results indicates the front of the drone. As illustrated, PX4 first accelerates the drone to a high speed (with the initial spikes of motor speeds close to time 0). It then maintains a stable high speed till until time-stamp 7 (sec). At this time, it decelerates as it reaches the targeted height. In contrast, the default Ardupilot rises very slowly and exhibits tilts and turns (due to some calibrations when taking off). After tuning, Ardupilot is more stable at take-off (i.e., the tilts and turns are avoided). If one looks into the motor speed chart, the spike and the dip (at 7th sec) appear, resembling the PX4's chart. While WBTuner is not able to achieve the same sharpness of the spike/dip as in PX4 due to other un-tuned parameters, the result is promising.

Fig. 3.23 shows the results of the second tuning mission (The three way points are indicated by A, B and C). When the default Ardupilot reaches the middle way point (i.e., B), it first tilts, turns at the same spot until its head points to the next way point C, and then flies towards C. Intuitively, changing the orientation at point B requires the drone to decrease its speed, and hence leads to a longer mission. Conversely, both PX4 and the tuned Ardupilot avoid turning as much as possible at point B, and rather change their orientation while flying towards C (as indicated by the white curved arrow), consequently finishing the mission in a much shorter time.

Fig. 3.24 shows the results of the testing mission. After tuning, the motors speed of Ardupilot is quite similar to PX4. Even more, its flight time is reduced from the original 105 seconds to 82 seconds (i.e., 22% fewer). The recorded videos for the test mission are available at [63–65].

OpenTuner cannot be applied here for the following reasons. (1) Several parameters that affect multiple flight modes in a single mission. They are tuned to different

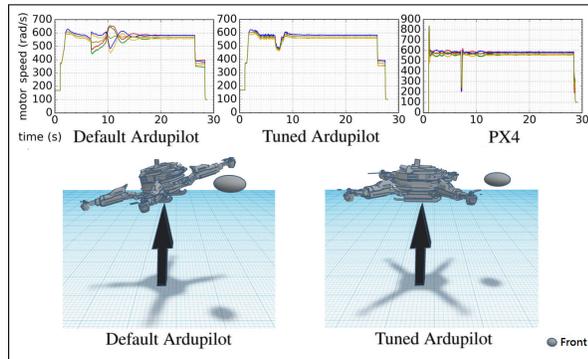


Fig. 3.22.: Tuning mission 1

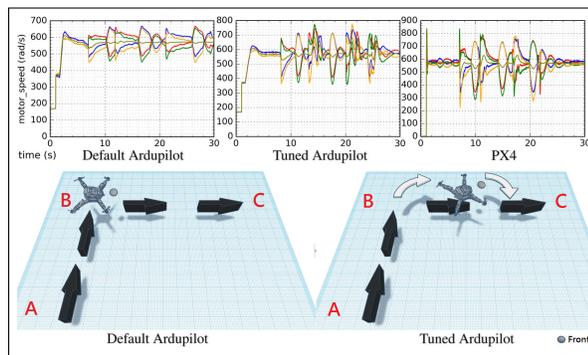


Fig. 3.23.: Tuning mission 2

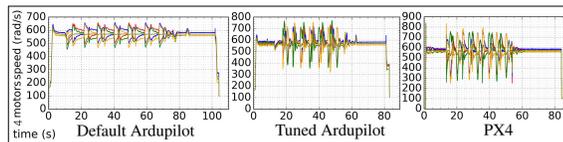


Fig. 3.24.: Testing mission

values for various modes. This cannot be supported by blackbox tuning; (2) Each sample run in OpenTuner is a whole execution that includes expensive simulator startup and drone preparation taking 3-4 minutes per sample. In contrast, WB-TUNER tunes small code regions and each sample run is just 20-30 seconds; (3) The simulator often fails to start (we suspect that it results from the locked resources of

previous closed execution). This is not a problem for WBTUNER as it can spawn all the sampling/tuning processes after a successful start.

### 3.6 Summary

In this chapter, we propose WBTUNER, a general white-box tuning engine. It provides primitives that allow users to easily compose complex tuning tasks as if they are writing extensions to the original programs. Our experiments show that WBTUNER substantially improves data processing results and outperforms the state-of-the-art black-box tuning engine. In the next chapter, we will discuss a novel approach for more efficient parameter selection by leveraging artificial intelligence.

## 4. PROGRAMMING SUPPORT FOR AUTONOMIZING SOFTWARE

In this chapter, we continue our focus on improving the parameter selection of data processing programs with better efficiency by leveraging artificial intelligence (AI). Most traditional software systems are not built with the artificial intelligence support in mind. Among them, some may require human interventions to operate, e.g., the manual specification of the parameters in the data processing programs, or otherwise, would behave poorly. We propose a novel framework called AUTONOMIZER to autonomize these systems by installing the AI into the traditional programs. AUTONOMIZER is general so it can be applied to many real-world applications. We provide the primitives and the runtime support, where the primitives abstract common tasks of autonomization and the runtime support realizes them transparently. With the support of AUTONOMIZER, the users can gain the AI support with little engineering efforts. Like many other AI applications, the challenge lies in the feature selection, which we address by proposing multiple automated strategies based on the program analysis. Our experiment results on nine real-world applications show that the autonomization only requires adding a few lines to the source code. Besides, for the data-processing programs, AUTONOMIZER improves the output quality by 161% on average over the default settings. For the interactive programs such as game/driving, AUTONOMIZER achieves higher success rate with lower training time than existing autonomized programs.

### 4.1 Introduction

Autonomous software systems have achieved incredible success in *specialized* domains. For example, the world is shocked when the self-learning AlphaGo program

defeated the human champions [66–69]. The **Waymo** self-driving car has driven flawlessly 25,000 miles each day on complex city streets [70], which is valued at around \$175B [71]. Inspired by the existing autonomous systems, we are intrigued by the question whether the *general* software engineering problems can benefit from the autonomization. In this work, we take the initiative to answer this question and share the results.

#### 4.1.1 Autonomization: Bringing the Intelligence to Traditional Programs

In the following, we present two sets of general software engineering problems, which are representative of the problems that the autonomization potentially applies to.

***Parameterized Programs.*** Many traditional software systems, especially data processing, machine learning and scientific computation programs, often carry the parameters that affect the quality of the results. However, different inputs require different configurations to achieve the ideal results, i.e., no parameter configuration universally applies. Therefore, the users need to manually configure the parameters, which is difficult for normal users due to the great domain expertise required and sometimes even difficult for the experts if the parameter value space is huge (consider the neural network hyperparameters [72]). To mitigate this problem, the users may use the autotuning tools [31] to tune the configuration. In either case, the users are faced with the dilemma that they either need to tune the parameters for each input, which prohibits the application to large volume of inputs, or have to tolerate the unsatisfactory results.

Artificial intelligence, specifically the supervised learning (SL), excels at learning the target values (i.e., the ideal parameter values) for different settings (i.e., inputs). According to the experiments (Section 5.4), the parameter values predicted by SL improve the quality of results over the default values by around 70% for **Canny**, a popular edge detection program. Besides, the prediction of SL is very fast, in contrast

with the manual specification or the autotuning, which means the programs equipped with SL can process large volume of inputs fast while offering good results. Therefore, we propose to install AI into the traditional parameterized Programs.

***Interactive Programs.*** Many software systems interact with the environments, e.g., the monkey testing software which interacts with the mobile UI environment [73], the Mario game agent that interacts with the simulated game environment [74], the software that controls the cooling of the data center [75] and many other cyberphysical system (CPS) software. However, these pieces of software usually follow the random behaviors [73] or the simple heuristics [75], which do not behave effectively in practice.

We propose to install the AI, in particular the reinforcement learning (RL) which is designed for the action selection, into the software systems such that they behave smart and achieve better results, while offering the full automation. Deepmind researchers pioneer the study: They reduced Google’s data centre cooling bill by 40% by leveraging the AI in place of the simple heuristic-based control [75]. Earlier, they built the control through deep reinforcement learning and demonstrated it can surpass the human-level performance on a set of Atari games [76]. Another work integrated RL with the monkey testing and shown the effectiveness [77].

#### 4.1.2 Problems and Challenges

We discuss potential problems and challenges of program autonomization in the following.

***Autonomization is Tedious and Not Portable.*** In existing autonomous systems [66, 74–83], autonomization is implemented manually, which requires a lot of engineering efforts. In general, it needs to construct the neural network model and select the feature variables, of which the runtime values are used as the inputs of the model. Besides, at runtime, it needs to (1) collect the feature values, (2) save them to database and load them back in batches, (3) feed them to the model for training and prediction, (4) integrate the predicted result into the execution, and (5) provide

program checkpointing/restore logic when the training (esp., reinforcement learning) enters the ending state. Even worse, when moving to a new application, the above process needs to be repeated.

***Challenge on Feature Extraction.*** Similar to many other neural network applications, the crucial challenge lies in the feature selection, i.e., selection of the **feature variables** for predicting the **target variables** specified by the users. The target variables are the variables of which the values are to be predicted and affect the quality of results, e.g., the parameters in the parameterized programs or the actions in the interactive programs.

Existing works usually use the raw program inputs (e.g., images) as the feature variables. For example, the autonomous game agent work [76] uses raw images to train AI models, which leads to very slow training process (e.g.,  $\sim 83$  hours of training to make an AI competitive with respect to human [84, 85]). The problem is that the neural network needs several preprocessing layers, i.e., the *convolutional neural network* (CNN) layers [86], to derive the high-level information from the raw program inputs (e.g., images).

Our key insight is the high-level information (e.g., the position of Mario or the image histogram information of Canny) is already derived through the code logic and stored as program variables. Therefore, we propose to use the program variables (which provide the rich information) as the feature variables, thereby obviating the preprocessing layers.

However, large number of program variables exist. Given the user-specified target variables, it is important yet challenging to select the variables that are most semantically relevant to the target variables as their feature variables. Manual extraction of the feature variables [80] would impose heavy burden on the programmers. We designed multiple automated strategies atop the program dependence graph and found through the extensive experiments that (1) the selected feature variable and the target variable should be correlated (share some common dependent), (2) the selected

feature variable closer to the common dependent leads to better prediction quality. The feature extraction algorithms are discussed in Section 4.4.

### 4.1.3 Our Design

In this paper, we propose a novel programming framework AUTONOMIZER which consists of the primitives <sup>1</sup> and the runtime support, where the primitives abstract the common components aforementioned while the runtime support does the heavy lifting and realizes them transparently.

As shown in the experiments (Section 5.4), autonomizing the programs, such as a widely used edge detection program Canny [29] and a speech recognition program Sphinx [41], only requires adding a few lines to the original source code. For SL programs, the autonomization output quality is improved by 161% on average over the baseline with execution overhead no more than 0.64X. For RL programs, the training procedures only need 3.5-20.36 hours for the AIs to be competitive with human players. Comparatively, prior art typically requires at least 83 hours of training [85] to be competitive with respect to human, which is not efficient.

**Contributions.** We made the following contributions.

- We proposed a novel idea of autonomizing traditional software programs, which applies to the parameter configuration of parameterized programs, the action selection of interactive programs and many other potential applications.
- We designed a novel programming framework which consists of the primitives and the runtime support. The primitives abstract common tasks of autonomization and the runtime support realizes them transparently.
- We designed the strategies for feature extraction.

---

<sup>1</sup>It is also possible to use the language constructs instead of the library API. However, it would require a specialized compiler and sacrifice the usability.

- We implemented our approach and evaluated it against nine real world programs. The results are promising as described above: the programs can be autonomized with little effort; the autonomized version leads to much better results while incurring the tolerable execution slowdown.

## 4.2 Autonomization Framework Overview

In this section, we show how to autonomize the `Mario` [87] game, which is a representative of a large set of interactive software applications that do not have autonomization in consideration during design. We will explain how to annotate and autonomize the game with reinforcement learning using our proposed primitives. The game is autonomized for two different purposes. We first autonomize the game to play by itself normally and compare the results with the model borrowed from `DeepMind` [76] which uses raw image screenshots as model input. Then we show how to autonomize the game to do coverage testing. Note that here we are not comparing our work with `DeepMind` but rather leveraging its model. `DeepMind`'s contributions are orthogonal to `AUTONOMIZER`. `DeepMind` demonstrates human-like learning by observing raw images. It does not focus on identifying an efficient way to train a model to play games.

***Primitives.*** The primitives are listed in Fig. 4.1, which are the library calls in the same programming language as the source program. While more details of the primitives will be discussed in Section 5.3, we will explain them when they are used in the example.

***Running Example.*** In this section, we show how to autonomize an interactive program, i.e., the `Mario` game, such that it achieves the decent score without human assistance. In Section 3.5.2, we further show that we can autonomize the parameterized programs to achieve ideal parameter configurations automatically on the fly.

In Fig. 4.2, we show how to autonomize the `Mario` game, where the primitives are highlighted. Lines 24-50 show the main game loop function of `Mario`. In each

<p><i>Library Calls :</i></p> <pre> @au_config(modelName, modelType, algo., layers, neuron1, ...)   @au_extract(extName, size, data)   @au_NN(modelName, extName1, ..., wbName1, ...)   @au_write_back(wbName, size, data)   @au_serialize(data1, ...)   @au_checkpoint()   @au_restore() </pre>
--

Fig. 4.1.: Primitives

iteration, multiple functions (lines 1-23) of the program are orchestrated to make the game work. For example, lines 5-13 handle minion collisions and lines 19-23 update Mario's position.

First, the user specifies with the primitive *au\_write\_back()* at line 44 the target variable (i.e., the output of the neural network model) , which is the variable `actionKey` that holds Mario's action (line 46). AUTONOMIZER then automatically extracts some program variables as the feature variables (i.e., the inputs of the model), which are also annotated with the primitive *au\_extract()* at lines 9-10, 17, and 21-22. The feature variables, e.g., the positions of Goombas at lines 9-10, contain the important and relevant information for predicting the target variable, e.g., Mario's action at line 46. We refer the readers to Section 4.4 for the feature extraction algorithms.

While executing the primitive *au\_extract()*, the AUTONOMIZER runtime automatically records the values of the feature variables into a database and assigns names to them for later reference. While executing the primitive *au\_write\_back()*, e.g., at line 44, the AUTONOMIZER runtime updates the target variable *actionKey* with the predicted value of the target variable. Note the value 5 means there are 5 possible actions.

During initialization, the neural network is configured with the primitive *au\_config()* (line 2). In our example, the model has two hidden layers with 256 and 64 neurons, respectively. The size of the input and output layers is automatically computed based

on the input fed to the network and the output to be predicted. We also provide a callback function in which the users can create arbitrary neural networks from scratch with Tensorflow, which is omitted due to space limit.

The program interacts with the neural network via the primitive `au_NN()` at line 40. It works in two modes: the training mode and the deployment mode. In the training mode, the program sends data to train the model, in addition to generating the predicted value. In the deployment mode, the program sends data solely to generate the predicted value. In practice, we produce two versions for the modes. AUTONOMIZER automatically writes the output value predicted by the model to the database and index it with the name `output` specified at line 43.

If Mario enters the end state (i.e., Mario dies), it is important to roll back to a previous checkpoint to avoid the expensive full restart. We provide two primitives to achieve this: The primitive `au_checkpoint()` checkpoints the program state at line 27. The primitive `au_restore()` would restore the program state at line 48. Note that the neural network states of AUTONOMIZER are not affected by this pair of primitives. More details are explained in Section 4.5.

**Execution Model.** The simplified execution model is shown in Fig. 5.4. Given the user-annotated target variable  $Y$ , AUTONOMIZER first extracts the feature variable  $X$  for predicting  $Y$  (Section 4.4). The variables  $X$  is then annotated in the program.

The runtime execution is as follows. The original main process executes normally until it reaches the `au_extract()` primitive (①). At this point, the value of the feature variable  $X$  is saved to the database (②).

The main process continues its execution until it reaches the primitive `au_NN()`. At this point, the main process transfers the control of the execution of the original program (③) to the execution of a piece of Python code (④), which is generated by AUTONOMIZER based on the annotations and performs the model training and testing using Tensorflow. Besides, AUTONOMIZER also feeds the input data  $X$  stored in the database to the model, uses the model to predict the output value (which corresponds to the target variable  $Y$ ) and writes the value to the database.

```

1 void initGame() {
2   // ... Init game ...
3   au_config("Mario", DNN, QLearn, 2, 256, 64);
4 }
5 void minionCollision() {
6   for (int i=0; i<Minion.size(); i++) {
7     for (int j=0; j<Minion[i].size(); j++) {
8       // ... Update minion collision ...
9       au_extract("MnX", 1, minion[i][j]->X);
10      au_extract("MnY", 1, minion[i][j]->Y);
11    }
12  }
13}
14 void checkObj() {
15   if (checkObj(player.front) == "PIPE")
16     ...
17   au_extract("OBJ", 1, player.front);
18}
19 void updatePlayer() {
20   // ... Update player.x and player.Y ...
21   au_extract("PX", 1, player->X);
22   au_extract("PY", 1, player->Y);
23}
24 void gameLoop() {
25   while (true) {
26     terminated = 0;
27     au_checkpoint();
28     // Reward calculation
29     if (moveForward(player)) reward = 2;
30     else reward = -1;
31
32     if (reachFlagPole(player)) {
33       reward = 10; terminated = 1;
34     } else if (dead(player)) {
35       reward = -10; terminated = 1;
36     }
37     // This line is only added for self-testing
38     if (checkNewCoverage()) reward = 30;
39
40     au_NN("Mario",
41         au_serialize("PX", "PY", "MnX", "MnY", "Obj"),
42         reward, term,
43         "output");
44     au_write_back("output", 5, actionKey);
45     // ... Act based on returned data ...
46     act(actionKey);
47
48     if (terminated) au_restore();
49   }
50}

```

Fig. 4.2.: Autonomizing Mario. The highlighted statements are added. AUTONOMIZER primitives start with au.

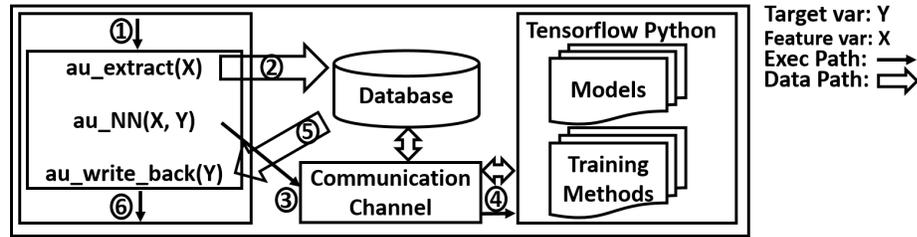


Fig. 4.3.: Execution model

Afterwards, the control of the execution is transferred back to the main process. Note the output value is now stored in the database. The primitive `au_write_back()` (⑤) loads it from the database to update the program variable  $Y$ .

The main process continues its normal execution (⑥) with the updated  $Y$  value. Note that AUTONOMIZER supports multiple model instances in one execution.

**Result and Comparison.** Videos of the training process and the autonomization result can be found at [88]. It took around 5.7 hours to train Mario to have reasonable behavior without using GPU. Furthermore, we compare the results between our model (i.e., the model using extracted program states) and the DeepMind model [76, 89] (i.e., a model using raw image screenshots).

*Low Engineering Efforts.* With the support of AUTONOMIZER, the users achieve the autonomization with very few annotations, as highlighted in Fig. 4.2. The automatic feature extraction of AUTONOMIZER further alleviates the burden of specification. Comparatively, existing work for Mario autonomization [74] spends great engineering efforts in the common tasks such as data collection, data saving/loading, integration of the model and the original program.

*Better Result and Faster Training.* We stop the training if the score of Mario is comparable with human (i.e., difference  $< 20\%$ ) or if the training time exceeds 24 hours. Here the score refers to the stage clearance rate of 10 runs.

According to the experiments, the DeepMind model is trained for 8000 epochs before exceeding the 24 hours limit. Comparatively, Our model is trained for only 2000 epochs. Note each epoch corresponds to 100 iterations of the loop. The results

show that the DeepMind model achieves the score 40% after 24 hours’ training, while our model achieves the score 80% after only 5.7 hours’ training.

The reason for the difference lies in the feature selection. The DeepMind model uses the raw images as the model inputs and applies multiple convolution layers to derive high-level information from the raw images. In particular, each image is an  $84 \times 84 \times 4$  input array (after preprocessing). The neural network has three convolution layers, each followed by a max pooling layer, and finally two hidden layers with 256 and 64 neurons. Due to the complexity of the network, the training requires very long time to achieve good result, or equivalently, achieves bad result within a short time.

Comparatively, our key insight is that the programmers derive the high-level information through the code and store them in some internal program variables. Therefore, our model directly uses such high-level information as the model inputs, thereby obviating the need for the three convolution layers in the DeepMind model. Our simpler model achieves better results while requiring shorter training time.

The screenshots illustrate the difference more intuitively. In Fig. 4.4, following our model, Mario jumps only when it’s necessary. In Figure 4.5, the Mario following the DeepMind model keeps jumping all the time, indicating the model is still at the early stage of the training. Intuitively, if Mario jumps too often, it is less likely to stay on the ground where control can be applied, i.e., the chance of controlling Mario becomes lower. Thus, it easily hits the Goombas and dies as shown in Fig. 4.5. The relevant videos can be found at [88].

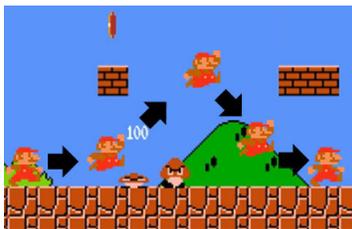


Fig. 4.4.: Using internal data

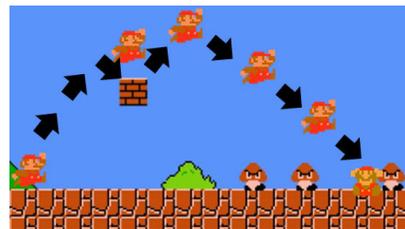


Fig. 4.5.: Using raw data

***Autonomization for Software Self-Testing.*** To further demonstrate the capability of software autonomization and the flexibility of AUTONOMIZER, we autonomize the Mario game in a way that it performs the testing. All we need to do is to update the reward so that it reflects the code coverage improvement, in addition to the original reward which reflects the stage clearing. Line 38 in Fig. 4.2 show the added reward for code coverage, where the code coverage is collected using `gcov` [90]. Intuitively, any improvement of code coverage results in large reward. Note that we also need the original reward to ensure Mario survives long enough to reach the complex game logic. After training for 10 hours, Mario is able to make many unexpected moves that lead to good code coverage quickly. In 30 seconds of game play,  $\sim 65\%$  code coverage can be achieved. In contrast, the previous AI model (which is not designed for testing) cannot reach a similar code coverage after 10 mins, not to mention the random testing in which Mario easily dies within seconds.

Fig. 4.6 shows the screenshot of the testing. Observe that Mario has more interesting behaviors, e.g., Mario jumps backward to eat the mushroom ((①)-(③)) and then jumps into the ditch ((④)). The AI even found two bugs during self-testing. The videos of both bugs can be found at [88].

Fig. 4.7 shows one of the bugs. Before falling to the ground of the dungeon, Mario moves in some unexpected ways such as jumping forward. As a result, Mario reaches the ceiling of the dungeon. Then it tries to further jump forward and goes out of the screen, which crashes the program. Code inspection discloses that the developer missed a boundary check. This case study illustrates the capabilities of AUTONOMIZER in enabling future research along this line.

### 4.3 Execution Model: Semantics

AUTONOMIZER features a novel set of primitives and a unique execution model that are particularly designed for software autonomization. After compilation and linking with AUTONOMIZER runtime, an executable with two execution modes is

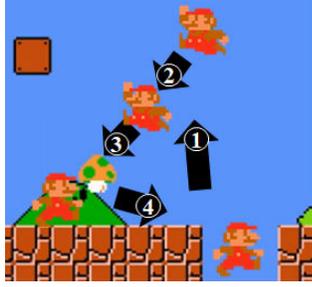


Fig. 4.6.: Coverage testing

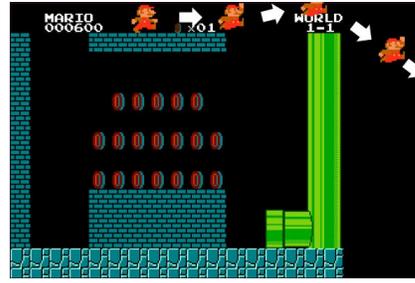


Fig. 4.7.: Bug

generated. One for training, and one for deployment (or production run). Training is piggybacking on normal software execution to derive an AI model (or multiple models). During production run, the model is used to replace human interaction/decisions. Supervised and reinforcement learning are supported by default. AUTONOMIZER is also extensible through its interface with Tensorflow to support other methods.

To support learning from software operation, AUTONOMIZER needs to monitor software execution, trace values of feature variables that would be used by the model to make decision, record the desirable decisions made by human users to serve as the objective of learning (i.e., desirable model output), and roll back software execution state but not the model state (during reinforcement learning). In supervised learning, model training is conducted offline after execution. In reinforcement learning, model training is conducted online and the training execution interleaves with software execution. Intuitively, AUTONOMIZER collects model inputs/outputs for a window of time (e.g., a few game loop iterations) and then invokes the training method to use the collected data.

During production runs, AUTONOMIZER intercepts values of feature variables and passes them to the model to make prediction. The predicted results are then copied to some program variables (such as the variable denoting the next move of Mario) to drive execution. Note that all the aforementioned complexities are transparent to

the users. In the following, we discuss the semantics of individual primitives, which is important to precisely understand how AUTONOMIZER works.

DEFINITIONS: $ProgStore \sigma ::= Var \rightarrow Value$ $DBStore \pi ::= String \rightarrow \widetilde{Value}$ $Model \theta ::= String \rightarrow \widetilde{Parm}$ $ModelType \delta ::= DNN \mid CNN$ $Algorithm \alpha ::= Q \mid AdamOpt$ $Mode \omega ::= TR \mid TS$ $String t, mdName, extName, wbName ::= [a-zA-Z0-9]^+$ $Int i ::= [0-9]^+$ $Stmt s ::= \dots \mid \mathbf{runModel}(\widetilde{Parm}, \widetilde{v}) \mid \mathbf{gradient}(\widetilde{Parm}, \widetilde{v}) \mid \mathbf{mkSnapshot}(\sigma, \pi) \mid \mathbf{rtSnapshot}() \mid$ $\mathbf{loadModel}(mdName) \mid \mathbf{buildModel}(mdName, \delta, \alpha, l, n1, \dots) \mid \mathbf{concat}(\widetilde{v}_1, \widetilde{v}_2)$		
STATEMENT RULES: $\boxed{\sigma, \pi, \theta, \omega : s \xrightarrow{s} \sigma', \pi', \theta', \omega, s'}$		
$\sigma, \pi, \theta, \omega : x := v$	$\xrightarrow{s} \sigma[x \mapsto v], \pi, \theta, \omega, \mathbf{skip}$	[ASSIGN]
$\sigma, \pi, \theta, TR : @\mathbf{au\_config}(mdName, \delta, \alpha, l, n1, \dots); s$	$\xrightarrow{s}$	
$\sigma, \pi, \theta', TR : s$	$, \text{ in which if } \theta(mdName) \equiv \perp \text{ then } \theta' = \theta[mdName \mapsto \mathbf{buildModel}(mdName, \delta, \alpha, l, n1, \dots)] \text{ else } \theta' = \theta$	[CONFIG – TRAIN]
$\sigma, \pi, \theta, TS : @\mathbf{au\_config}(mdName, \delta, \alpha, l, n1, \dots); s$	$\xrightarrow{s}$	
$\sigma, \pi, \theta', TS : s$	$, \text{ in which if } \theta(mdName) \equiv \perp \text{ then } \theta' = \theta[mdName \mapsto \mathbf{loadModel}(mdName)] \text{ else } \theta' = \theta$	[CONFIG – TEST]
$\sigma, \pi, \theta, \omega : @\mathbf{au\_extract}(extName, size, x); s$	$\xrightarrow{s}$	
$\sigma, \pi', \theta, \omega : s$	$, \text{ in which } \pi' = \pi[extName \mapsto \mathbf{concat}(\pi(extName), x[0], \dots, x[\sigma[size] - 1])]$	[EXTRACT]
$\sigma, \pi, \theta, \omega : @\mathbf{au\_write\_back}(wbName, size, x); s$	$\xrightarrow{s}$	
$\forall i \in [0, \sigma[size)), \sigma[x[i] \mapsto \pi(wbName)[i]], \pi, \theta, \omega : s$		[WRITE – BACK]
$\sigma, \pi, \theta, \omega, TR : @\mathbf{au\_NN}(mdName, extName, wbName); s$	$\xrightarrow{s}$	
$\sigma, \pi[wbName \mapsto \mathbf{runModel}(\theta'(mdName), \pi(extName)), extName \mapsto \perp], \theta', TR : s$	$, \text{ in which } \theta' = \theta[mdName \mapsto \theta(mdName) - \mathbf{gradient}(\theta(mdName), \pi(wbName))]$	[TRAIN]
$\sigma, \pi, \theta, \omega, TS : @\mathbf{au\_NN}(mdName, extName, wbName); s$	$\xrightarrow{s}$	
$\sigma, \pi[wbName \mapsto \mathbf{runModel}(\theta(mdName), \pi(extName)), extName \mapsto \perp], \theta, TS : s$		[TEST]
$\sigma, \pi, \theta, \omega : @\mathbf{au\_serialize}(t_1, t_2); s$	$\xrightarrow{s} \sigma, \pi[\mathbf{strcat}(t_1, t_2) \mapsto y], \theta, \omega : s$	$, \text{ in which } y = \mathbf{concat}(\pi(t_1), \pi(t_2))$ [SERIALIZE]
$\sigma, \pi, \theta, \omega : @\mathbf{au\_checkpoint}(); s$	$\xrightarrow{s} \sigma, \pi, \theta, \omega : \mathbf{mkSnapshot}(\langle \sigma, \pi \rangle); s$	[CHECKPOINT]
$\sigma, \pi, \theta, \omega : @\mathbf{au\_restore}(); s$	$\xrightarrow{s} \sigma', \pi', \theta, \omega : s$	$, \text{ in which } \langle \sigma', \pi' \rangle := \mathbf{rtSnapshot}()$ [RESTORE]

Fig. 4.8.: Operational Semantics

### 4.3.1 Definitions

Definitions related to the operational semantics are presented at the top of Fig. 4.8. AUTONOMIZER has two stores, the *Program Store*  $\sigma$  for original program states, which

can be intuitively considered as a hash map that projects a variable to its current value, and the *Database Store*  $\pi$  which stores the extracted feature variable values. It is a mapping from a string to a list of values. The value of a program variable extracted by the primitive `@au_extract()` will be appended to a list in the database store indexed by a string name. Furthermore, the outputs returned by the underlying model are also put in the database store before they are written back to target program variables with the primitive `@au_write_back()` (to affect the execution). The two stores are isolated. Transferring data between the two should be explicitly requested by the programmer through the primitives.

We abstract a neural network model  $\theta$  as a mapping from a model name to a list of parameter values. Intuitively, one can think of them as the weight values of matrices in individual model layers. By default, AUTONOMIZER supports two model types: fully connected neural network (DNN) as well as CNN and two popular algorithms: Q [91] for RL as well as AdamOpt [92] for SL. Execution mode  $\omega$  denotes the two modes supported: *TR* for training and *TS* for production runs (or testing). AUTONOMIZER runtime provides a list of API functions denoted as statement extensions (e.g., `buildModel()` for initializing a model inside Tensorflow). The semantics of many primitives are resolved to these API functions.

### 4.3.2 Rules

The semantics rules are in the lower part of Fig. 4.8. As indicated by the configuration (in the box), each rule is a transition of statement  $s$  to  $s'$  while updating the two stores and the model. We organize the rules to two groups: (i) model construction/training/testing and (ii) checkpointing/restore.

#### Model Construction/Training/Testing

**Model Construction.** In Rule [*CONFIG-TRAIN*], AUTONOMIZER in the training mode creates a new model if model exists in memory by executing the statement

**buildModel**, where the model name  $modelName$ , the model type  $\delta$ , the training algorithm  $\alpha$ , the number of layers  $l$  and the number of neurons  $n1$  are specified by the programmer. In Rule  $[CONFIG - TEST]$ , AUTONOMIZER in the testing mode simply loads an existing trained model with the specified model name  $modelName$  by executing the statement **loadModel**.

**Model Training and Testing.** In Rule  $[EXTRACT]$ , AUTONOMIZER appends the variable value(s) to a list in the database store indexed by a unique name  $extName$ . Note that if the primitive  $au\_extract()$  is inside a program loop that iterates multiple times before invoking the  $au\_NN()$  primitive, the list contains multiple values. In Rule  $[TRAIN]$ , AUTONOMIZER trains the model by gradient descent [93], i.e., updating model parameters  $\widetilde{Parm}$  along the largest gradient. Then, the model output generated by executing the statement **runModel** on the model input retrieved by  $\pi(extName)$  is put in the database store with index  $wbName$ . Afterwards, the model input is reset to an empty list (by mapping  $extName$  to  $\perp$  in  $\pi$ ). Rule  $[TEST]$  is similar to rule  $[TRAIN]$  except that it does not update the model. It simply uses the model.

In Rule  $[WRITE - BACK]$ , AUTONOMIZER writes the value with name  $wbName$  from the database store back to the program variable  $x$ . Rule  $[SERIALIZE]$  concatenates multiple lists of values into a single list through primitive **concat()**. The names of those lists are also concatenated through  $strcat()$ . This feature helps to combine multiple extracted values into one list and feed it to the underlying model. Note that neural network models only take vector inputs. For example, at line 41 of Fig. 4.2, six lists of extracted values are combined into one list of values and fed to the neural network as input. Note that AUTONOMIZER supports serializing multiple lists.

## Checkpointing/Restore

In Rule [*CHECKPOINT*], AUTONOMIZER checkpoints the states of current program store and database store by making the snapshot through the statement `mkSnapshot()`. Note that although model state in  $\theta$  is part of the software process, it is not checkpointed because we want the model to accumulatively learn. In Rule [*RESTORE*], AUTONOMIZER restores the states of program store and database store with the previously made snapshot through `rtSnapshot()`. Note that the states between both stores need to be consistent, so their states have to be checkpointed and restored together.

### 4.4 Feature Variables Extraction

In this section, we discuss how to extract program variables that correspond to important features. The values of these variables will be extracted as model inputs. The analyses are different for supervised learning and reinforcement learning. We adopt dynamic dependency analysis instead of static analysis which incurs too many false positives.

***Supervised Learning.*** In our settings, the supervised learning (SL) is used to predict the ideal value of the parameters (i.e., the *target variables*) that affect the quality of the result based on the relevant internal program states (i.e., the *feature variables*). While the target variables are specified by the users, which is an easy task according to our experiments, it is non-trivial (e.g., labor-intensive and error-prone) for the users to specify the feature variables. To lift the burden, we automatically extract the feature variables by combining heuristics and program analysis. We also conducted extensive experiments to validate the effectiveness of the heuristics (Section 5.4).

First, we observed the ideal values of the target variables (or parameters) vary for different program inputs, meaning that they are sensitive to the inputs. Therefore, we identify the input variables and those that transitively depend on them as the *candidate* feature variables. Furthermore, we conduct correlation analysis to determine the

---

**Algorithm 2** Automatic SL Feature Extraction
 

---

**Require:**  $In, Trg, G_{Dep}$ 
**Ensure:**  $Feature$ 

```

1:  $Candidate \leftarrow In \cup dep(In)$ 
2:  $Feature \leftarrow Map()$ 
3: for each  $v \in Trg$  do
4:   for each  $w \in Candidate$  do
5:     if  $dep(w) \cap dep(v) \neq \emptyset$  then
6:        $Feature[v] \leftarrow Feature[v] \cup \{w, \infty\}$ 
7:     for each  $w, dist \in Feature[v]$  do
8:        $dist \leftarrow BFS(G_{Dep}, w, first(dep(w) \cap dep(v)))$ 
9:        $Feature[v] \leftarrow \{w, dist\}$ 
10:     $Sort(Feature[v])$ 
11: return  $Feature$ 

```

---

subset of candidate feature variables correlated with each target variable. Intuitively, we say two variables are correlated if they are depended upon by the same variable. Lastly, to refine the subset of feature variables, we rank the feature variables heuristically according to their “distances” to the correlated target variable, and select the top-ranked variables for prediction. According to the experiments (Table. 4.3), the refinement leads to better prediction results.

In the following, we explain the automatic extraction and the involved terms in details.

Algorithm 2 takes three inputs:  $In$ ,  $Trg$ , and  $G_{Dep}$ .  $In$  is the set of input variable set,  $Trg$  is the set of target variable, and  $G_{Dep}$  is the pre-computed dynamic dependency graph. First, we construct the candidate set, which consists of the input variables and their transitive dependents.

$Feature$  is a map that maps a target variable to its feature variables, which is returned eventually. For each target variable  $v$  in  $Trg$ , if a candidate feature variable  $w$  shares some common dependent with  $v$  (line 5), then  $w$  is a feature variable

correlated with  $v$ . For prediction purpose,  $w$  is not considered as feature variable if it depends on  $v$ .

To rank a candidate feature variable  $w$ , we use its dependency graph distance, which is defined as the number of edges between  $w$  and the first common descendent of  $w$  and  $v$ . In lines 7-9, the shortest distance from each  $w$  to the common descendent is found by BFS on the dependency graph  $G_{Dep}$ . In line 10, the feature variables are sorted according to  $dist$ , which allows us to further select the top-ranked feature variables. Intuitively, the shorter the distance, the more abstract (and the more important) the feature variable.

Fig. 4.9 demonstrates the example of extracting feature variables in **Canny**. Variable  $lo$  is a target variable and the remaining are candidate feature variables. Variable  $hist$  is ranked first to predict  $lo$  because it has distance 1 to first common descendent  $result$ . Feature variable  $sImg$  has distance 2 so it is ranked lower than  $hist$ .

**Reinforcement Learning.** We propose a feature variable identification technique for the reinforcement learning (RL) applications. Unlike SL programs, RL programs are usually not for one-time data processing like **Canny**. Instead, they often have some main loop that continuously updates program states, such as the game loop in game applications and the control loop in autonomous vehicle control software. Intuitively, variables that represent these continuously updated states are *candidate* feature variables. Note that they may not be dependent on external inputs (e.g., user key strokes).

For a target variable, other program variables that share common descendent with it are considered correlated with it. Those program variables are candidate feature variables. According to our observation, variables that correlate with target variable contain program states that affect the prediction result of target variable in RL applications. Our experiment results in Section 5.4 also justify the observation. For simplicity, AUTONOMIZER only checks variables that are used in the same functions as variables that depend on the target variable. After finding all candidates, AUTONOMIZER prunes redundant or unchanging variables based on runtime values of each

variable’s trace according to two thresholds set by the user. The remaining variables are combined and returned as features. Note that ranking is not as effective as in SL because most feature variables would have loop-carry dependencies due to the iterative updates.

---

**Algorithm 3** Automatic RL Feature Extraction

---

**Require:**  $Trg, UseFunc, ProgVar, \epsilon_1, \epsilon_2$

**Ensure:**  $Feature$

```

1:  $Feature \leftarrow Map()$ 
2: for each  $v \in Trg$  do
3:    $Candidate \leftarrow Map()$ 
4:   for each  $w \in ProgVar$  and  $w \neq v$  and
        $UseFunc[dep(v)] \cap UseFunc[w] \neq \emptyset$  and
        $dep(v) \cap dep(w) \neq \emptyset$  do
5:      $Candidate[w] \leftarrow Scale_{0-1}(Tracing(w))$ 
6:     for each  $w, Trace_w \in Candidate$  do
7:       for each  $x, Trace_x \in Candidate$  and  $x \neq w$  do
8:         if  $EucDist(Trace_w, Trace_x) \leq \epsilon_1$  then
9:            $Delete(Candidate[x])$ 
10:        if  $Variance(Trace_w) \leq \epsilon_2$  then
11:          continue
12:         $Feature[v] \leftarrow Feature[v] \cup w$ 
13: return  $Feature$ 

```

---

Algorithm 3 takes five inputs:  $Trg$ ,  $ProgVar$ ,  $UseFunc$ ,  $\epsilon_1$  and  $\epsilon_2$ .  $Trg$  is the target variable set, and the  $ProgVar$  set contains all program variables. Map  $UseFunc$  maps a variable to its usage functions. Thresholds  $\epsilon_1$  and  $\epsilon_2$  are used to prune redundant and unchanging variables respectively.

At line 4, if program variable  $w$  is used in the same function as target variable  $v$ ’s dependent variable, and both  $v$  and  $w$  have common descendents, then  $w$  is considered correlated with  $v$  and is added to the map  $Candidate$  with its runtime trace  $Trace_w$  which contains  $w$ ’s runtime values in a profiled time sequence. The sequence of trace

values are scaled [94] between 0 and 1 (line 5). In lines 8-9, the similarity between  $w$  and  $x$  is computed according to the distance between  $Trace_w$  and  $Trace_x$  using the euclidean distance formula.<sup>2</sup> For example, assume  $Trace_w$  contains [0.1, 0.3, 0.4] and  $Trace_x$  contains [0.1, 0.2], the similarity is  $\sqrt{(0.1 - 0.1)^2 + (0.3 - 0.2)^2 + (0.4 - 0)^2} = \sqrt{0.17}$ . If the similarity is less than  $\epsilon_1$ ,  $x$  is considered redundant and pruned. In lines 10-12, if the variance of  $w$ 's trace values is smaller than  $\epsilon_2$ ,  $w$  is considered unchanging and pruned. Intuitively, a rarely changing variable is not a good feature. Real pruning examples are shown in the TORCS autonomization case study.

Fig.4.10 shows an example of extracting feature variables in Mario to predict the target variable *right*, which makes Mario move right. Variable  $Player\_jX$  is a feature variable because it depends on itself and shares the same descendent with *speed* (and transitively with *right*). Another feature variable is  $Minion\_jX$  as it shares the descendent *collide* with  $pX$  (and transitively with *right*). Variable  $mX$  is pruned by  $\epsilon_1$  because it is a duplicate of  $Minion\_jX$ .

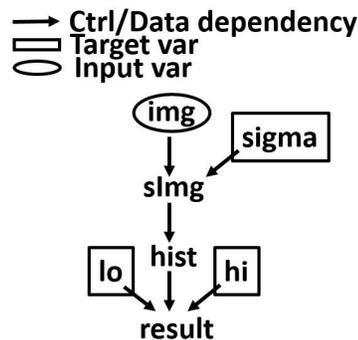


Fig. 4.9.: Alg.1 on Canny

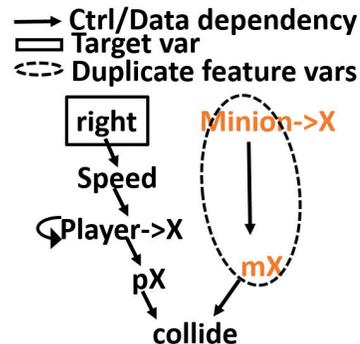


Fig. 4.10.: Alg.2 on Mario

## 4.5 Implementation

We leverage Tensorflow [95] to support model training and execution. Tensorflow is an open source software library with strong support for machine learning

<sup>2</sup>If the sequences' lengths are different, we append zeros to the shorter one.

applications. AUTONOMIZER essentially stitches the execution of Tensorflow with the execution of the original software. In other words, both executions occur in the same process space. Since Tensorflow has a comprehensive Python interface, at compile time, AUTONOMIZER generates a Python template for each injected model. The template essentially provides the API functions used in the semantics (e.g., `run-Model()`). These functions further invoke Tensorflow functions to realize their functionalities. Templates have to be generated based on primitive annotations because different model structures, model input sizes, and learning algorithms (in `au_config()` and `au_NN()`) lead to different templates. To make a source program interact with its Python template, a communication channel (Fig. 5.4) is implemented with the Python C/C++ extensions [96]. Details are elided.

**Checkpointing and Restore.** In our context, we need to checkpoint very complex software states in addition to memory states (e.g., thread and socket), and memory mapped I/O states. Simple software checkpointing by forking does not work. Even existing process level virtualization techniques (e.g., criu [97] or docker [98]) do not provide the guarantees of arbitrary-scale program states checkpointing/restore. We hence leverage KVM [99] to create checkpoints.

As mentioned earlier, we only checkpoint software states and database states but not model states. However, this is difficult to achieve as all these states are in the process space and indistinguishable for KVM. As such, before restoring to a checkpoint, AUTONOMIZER saves the current model states to persistent storage. After restoring, it overwrites the model states with those saved in storage.

## 4.6 Evaluation

In AUTONOMIZER, the feature variable extraction component is implemented using Valgrind-3.14.0 [100, 101]. The language and runtime are implemented in C++/Python. It is publicly available at [88]. Experiments were run on a machine with Intel i7-2640M 2.80GHz processor, 16GB RAM. We use a GPU of NVIDIA

Table 4.1.: Program analysis statistics

Program	LOC	Added	Trg	Candidate	Feature
		LOC	Vars	Vars	Vars
[SL] Canny	1.1K	8	3	26	1/23/23
[SL] Rothwell	1.3K	6	3	8	1/8/8
[SL] Phylip	12K	7	3	42	1/1/28
[SL] Sphinx	28K	37	2	107	13/14
[RL] Flappybird	0.8K	40	2	19	4
[RL] Mario	21K	73	5	345	25
[RL] Arkanoid	1K	39	-	-	-
[RL] TORCS	150K	89	2	370	20
[RL] Breakout	153K	65	-	-	-

Arkanoid and Breakout are emulator games so we annotate the emulator and use the exported game information directly.

Table 4.2.: Model statistics

Program	Raw		Med		Min		Raw/Min		Checkpoint Time(s)	Restore Time(s)
	Trace	Model	Trace	Model	Trace	Model	Trace	Model		
	Size(MB)	Size(MB)	Size	Size	Size	Size	Size	Size		
[SL] Canny	48	215	48	215	14	113	3.43	1.9	-	-
[SL] Rothwell	49	215	143	215	143	215	0.34	1	-	-
[SL] Phylip	4	21	1	1.8	1.2	1.9	3.33	14	-	-
[SL] Sphinx	157	172	36	15	36	15	4.36	11.66	-	-
	Raw				All		Raw/All			
[RL] Flappybird	781042	0.58	-	-	13.93	0.25	56069.06	2.32	27.38	6.23
[RL] Mario	68359	0.58	-	-	100.41	0.51	680.79	1.13	25.11	7.51
[RL] Arkanoid	351562	0.59	-	-	57.13	0.23	6153.72	2.57	26.82	6.21
[RL] TORCS	25313	1.9	-	-	714.8	0.47	35.41	4.04	25.33	6.91
[RL] Breakout	304018	0.62	-	-	11.44	0.21	26575	2.95	26.12	6.71

GeForce 1060 with 6GB RAM for supervised learning tasks. We use a wide variety of C/C++ benchmarks in our experiments. All programs have multiple datasets that can be found online or come with the program. We only selected those that have the ground truth (for SL).

In Section 4.6.1, we present the statistics that expose details about our approach. In Section 4.6.2, we conduct a comparative study of the effectiveness of our approach. In Section 3.5.2, we conduct in-depth case studies for more insights.

#### 4.6.1 Statistics

The statistics of our approach are presented in Table 4.1.

***Lines of Code Added.*** Column 2 shows the lines of code (LOC) of the programs and Column 3 shows the lines of code added for autonomization. According to Column 3, only a few lines are required for autonomization. In other words, with the help of our language support, the users introduce the advanced AI capability to the programs with little effort.

***Feature Variables.*** Column 4 shows the number of target variables, column 5 shows the number of candidate feature variables, and column 6 shows the feature variables available for selection.

***Supervised Learning.*** Any feature variables in column 6 are available for use. To assist the selection, they are ranked as discussed in Section 4.4. For example, only around 15 out of the 100+ candidates are available in the **Sphinx** application. The results suggest that a lot of candidates are pruned.

***Reinforcement Learning.*** For RL applications, we made similar observation that a large number of candidates are pruned during the selection. By pruning the redundant and unchanging candidate feature variables, our approach keeps the prediction focused, thereby making it more effective (as confirmed in Section 4.6.2). All feature variables are combined to predict multiple target variables due to the large overlap of the feature variable sets.

**Model Construction.** Table 4.2 shows the statistics related to the model, including (1) the size of the trace collected which consists of the input to the model, and (2) the size of the trained Tensorflow model.

*Supervised Learning.* To study the effect of distance (Algorithm 2) upon the model statistics, we compare three versions: *Raw* in Columns 2-3 which selects feature variables with the maximum distance (i.e., input variables), *Med* in Columns 4-5 which selects the feature variables with the median distance, and *Min* in Columns 6-7 which selects the feature variables with the minimum distances. For fairness, all versions use the same neural network architecture except for the input layer which accounts for different input size.

In columns 8-9, we show the ratio of *Raw* and *Min* in terms of the trace size and the model size, respectively. Both the trace size and the model size of *Raw* is much larger than *Min*. The reason is that the model of *Raw* usually has a larger number of input neurons than *Min* because the model inputs of *Raw* are typically raw data which are usually larger. Specially, for Rothwell, *Min* has larger trace size. This is because *Raw* and *Min* have a similar number of inputs but *Raw* represents them with the char type while *Min* represents them with the float type.

*Reinforcement Learning.* Results are compared between two settings: *Raw* and *All*. *Raw* uses the DeepMind model [76], which takes the scaled images as the inputs. *All* uses a four-layer fully connected neural networks. It combines and uses all feature variables identified by Algorithm 3 as inputs. For the RL programs, since game executions do not terminate as SL programs, we collected the statistics for a time window of a fixed length. In columns 8-9, we show the ratio ( $Raw/All$ ) in terms of the trace size and the model size, respectively. According to column 8, the size of the trace collected by *Raw* is 35-56069 times the size of *All*. This is because *Raw* collects raw images which are typically much larger than the (extracted) internal states collected by *All*. For instance, in Flappybird, a raw images collected by *Raw* is 700x800x4 bytes while a vector of internal data collected by *All* is only 32 bytes. The fact that RL usually requires many iterations further amplifies the difference.

Table 4.3.: Benchmark experimental results.

Program	Baseline		Raw			Med			Min			Train Time Raw/Min
	Exec. Time	Score	Train Time	Score	Score	Train Time	Exec. Time	Score	Train Time	Exec. Time	Score	
[SL] $\uparrow^1$ Canny	1.32	0.45	1791.52	1.58	0.543	1719.34	1.39	0.69	817.62	1.47	0.763	2.4
[SL] $\uparrow$ Rothwell	1.14	0.49	1978.1	1.63	0.64	1540.68	1.62	0.68	1616.37	1.53	0.705	1.22
[SL] $\downarrow$ Phylip	1.49	1.013	46.91	2.23	0.96	6.079	2.01	0.63	5.56	2.15	0.54	8.44
[SL] $\uparrow$ Sphinx	0.86	0.108	4001.52	1.52	0.57	3898.22	1.61	0.581	141.73	1.41	0.6323	28.23
	Players		Raw			All			All			Raw/All
[RL] $\uparrow$ Flappybird	0.0101	91.4%	t/o <sup>2</sup>	0.071	1.3%	-	-	-	12781.32	0.028	95.7%	-
[RL] $\uparrow$ Mario	0.0243	92%/90%	t/o	0.101	63%/40%	-	-	-	18465.42	0.046	84%/80%	-
[RL] $\uparrow$ Arkanoid	0.0041	77.2%/60%	t/o	0.049	1.5%/0%	-	-	-	41328.13	0.015	88%/60%	-
[RL] $\uparrow$ TORCS	0.003	100%	t/o	0.072	7.8%	-	-	-	73323.59	0.018	100%	-
[RL] $\uparrow$ Breakout	0.0071	29.8	68902.14	0.058	25.3	-	-	-	34452.74	0.012	28.5	1.99

1.  $\uparrow$ : Higher scores are better;  $\downarrow$ : lower scores are better.

2. "t/o" means using raw data cannot achieve the similar score (i.e., difference < 20%) of 10 human players.

Besides, according to column 9, the model size of *Raw* is larger than *All*. The reason is that the model of *Raw* has more layers where the first few layers extract the visual features from the raw images. Comparatively, *All* does not need such layers because it takes the program states of the feature variables as the input, which already hold the important feature information.

**Checkpointing/Restore.** Column 10 and 11 represent the time for creating and restoring a checkpoint. Only RL programs need checkpointing/restore. Creating a checkpoint takes around 26 seconds. Although it takes non-trivial time, it only needs to be done once at the beginning. After that, AUTONOMIZER can restore the checkpointed state when ending states (e.g., Mario dies) are encountered during training. Restoring takes around 7 seconds.

#### 4.6.2 Effectiveness

In this section, we present the study of the effectiveness of our approach. In particular, we are interested in how our autonomization affects the quality of the results. The results are shown in Table 4.3.

#### Experiment Settings

We first discuss the settings for gathering the results.

**Comparisons.** For the SL applications, we compare four versions: the baseline, *Raw*, *Med*, and *Min*, where the baseline refers to the execution with the default parameter configurations and the others three versions are explained in Section 4.6.1. For the RL applications, we compare three versions: the players version, *Raw*, and *All*, where the players version accounts for the average of 10 human players and the remaining two versions are explained in Section 4.6.1. Each experiment is repeated 10 times to compute the average.

**Scoring.** The quality of the results is measured with the score assigned to the results. We will explain how the score is assigned soon. Note that higher quality does not necessarily correspond to higher score. We put a mark in Column 1 to specify whether higher quality corresponds to a higher score or a lower score. For the SL programs, we use the built-in score functions shipped with the programs. For the RL programs, the score functions are not available. Instead, we define the score for each program that accounts for the progress or the success rate. For Flappybird, the score stands for the progress (i.e., how far the bird flies in terms of the percentage of the whole distance). For Mario, the score is a pair in the  $X/Y$  form, where  $X$  stands for the progress (i.e., how far Mario goes) and  $Y$  is the success rate (i.e., the rate of taking down the flag). For Arkanoid, the score is also a pair  $X/Y$ , which respectively stands for the progress (i.e., the percentage of cleared bricks) and the success rate (i.e., the rate of clearing all bricks). For Torcs, the score represents the driving progress (i.e., how far the car drives) without bumping the wall before finishing. For Breakout, the score represents the number of hit bricks before missing the ball. Note all scores are averaged over 10 runs.

**Training.** For SL programs, we train each version (except the baseline version which does not need training) until convergence (i.e., the score stops changing). For RL applications, training RL applications is normally considered non-stationary and hard to converge. Thus, we force the training to time out after 24 hours.

For each setting, we show the training time and the execution time, which correspond to the time taken by the training run and the testing run, respectively.

Specifically, for the RL applications, the execution time stands for the time taken by each iteration of the game loop.

## Experiment Results

Here we discuss the efficiency and effectiveness of using program internal features extracted by AUTONOMIZER through execution time, training time, and evaluation score.

***Comparing to Baseline and Human Players.*** We compare a baseline with the corresponding best setting. (*Min* for SL programs and *ALL* for RL programs).

*Supervised Learning.* The *Min* version (Columns 11-12) improves the baseline results by 161% on average. Besides, the overhead is less than 0.64X. It shows that autonomization improves the data processing results with small overhead.

*Reinforcement Learning.* The extracted feature variables help the *All* version (Columns 11-12) to achieve scores close-to/better-than the results of human players. The execution overhead ranges from 0.89X to 6.14X. Although 6.14X looks substantial, the incurred overhead does not cause any noticeable delay<sup>3</sup> because the execution time is computed for each time frame and the overhead is not human perceptible if the number of frames that the program can handle in a time unit exceeds a certain threshold.

***Comparison among Different Settings.*** We compare the results between different autonomization settings for SL and RL programs.

*Supervised Learning.* Although all settings (Columns 5-6, 8-9, and 11-12) outperform the baseline, the quality of improvements are different. Specifically, *Min*, *Med*, and *Raw* versions improve the baseline results by 161%, 141%, and 120% on average respectively. It shows that feature variables that close to the target variable are more important.

---

<sup>3</sup>The execution videos can be found at [88]

*Reinforcement Learning.* The *All* version (Columns 11-12) achieves good performance. On the other hand, the *Raw* (Columns 5-6) version cannot achieve similar score (difference  $< 20\%$ ) of human players and times out after 24 hours of training for most benchmarks. Furthermore, the execution overhead of *Raw* is higher (3.16X-23X) than *All* because Algorithm 3 helps *All* to prune many redundant feature variables and only retain the most representative ones. We further compare the *All* version with the *Raw* version using the most representative Breakout benchmark from DeepMind [76]. The *Raw* version uses the model in DeepMind. Observe that both the *Raw* and the *All* versions can compete with the human players and the *All* version has higher score. Note that DeepMind aims to demonstrate feasibility of human-like learning (from raw images). It does not focus on efficient training or automating the procedure.

**Training Time.** For SL programs, training *Min* only take  $\frac{1}{28} \sim \frac{1}{1.22}$  of the time taken by *Raw*. For RL programs, the training time for *All* version ranges from 3.5 to 20.36 hours while *Raw* times out for most RL benchmarks except the Breakout benchmark. The reason that the *Raw* version can be trained within the time limit for this benchmark is that the playing field for this game is not as complex as other benchmarks (e.g., Mario). Besides, after following the preprocessing steps (e.g., greyscale conversion and image cropping) in DeepMind [76], the input images become much less noisy. These factors make the *Raw* version model training easier compared to other RL benchmarks. For the *All* version, its training overhead is 1.99X less than the *Raw* version, which demonstrates the advantage of using internal program states.

### 4.6.3 Case Studies

In this section, we study the details of autonomizing two representative programs with SL and RL.

**Canny.** Canny [29], a popular edge detection tool, carries the parameters that affect the quality of the edge detection. To achieve the ideal result, each input image requires

a specific parameter configuration, i.e., no universal optimal parameter configuration generally applies. Thus, users either have to manually tune [102] or auto-tune [12] the configuration for each image, which prohibits the application from handling a large volume of images with satisfactory results. Comparatively, AUTONOMIZER automatically predicts the proper parameter values on the fly without human intervention. With the support of AUTONOMIZER, Canny can process a large volume of diverse images and provide satisfactory results.

*Ease of Use.* Fig. 4.11 shows the user specification for autonomization, specifically for the *Min* version. Initially, the user only needs to annotate the three important parameters of Canny (i.e., the target variables): `low`, `high` and `sigma`, where the former two are for edge traversal (lines 6-7) and the last one is for Gaussian smoothing (line 18). Then our extraction algorithm automatically recommends `image` (line 19) as the feature variable for predicting `sigma` and `hist` (line 9) as the variables for predicting `low` and `high`, which are annotated at lines 16 and line 4, respectively. In total, we need only 9 lines of extra code (i.e., the highlighted ones) as shown in Fig. 4.11.

*Running Example of Algorithm 2.* We also created the *Raw*, *Med*, and *Min* versions, which use different feature variables to predict the target variable. Consider Fig. 4.9, given the target variables `low` and `high`, Algorithm 2 determines `hist`, `mag`, `sImg`, and `image` as the candidate feature variables because they share the common dependent `result` with the target variables. Algorithm 2 further sorts them based on their distance (i.e., 1, 2, 3, 4 respectively) to the dependent. Accordingly, *Min* uses the variable `hist` with the minimum distance as the feature variable, *Med* uses the variable `sImg` with the medium distance and *Raw* uses `image` with the maximum distance.

*Usefulness.* To demonstrate how autonomization helps improve data processing results, we show the results of baseline, *Raw*, *Med*, and *Min*. For fair comparison, all versions use the same neural network structure, i.e., a six-layer fully connected neural network inspired by [103], except for the input layer. In particular, for input layer,

```

1 char *hysteresis(mag, lo, hi)
2 {
3   hist = computeHist(mag);
4   au_extract("HIST", 32767, hist);
5   au_NN("MinNN", "HIST", "LO", "HI");
6   au_write_back("LO", 1, &lo);
7   au_write_back("HI", 1, &hi);
8
9   return do_hysteresis(hist, lo, hi);
10 }
11
12 void canny(image, sigma, lo, hi) {
13   // 1. Gaussian smooth
14   au_config("SigmaNN", DNN, AdamOpt, 6, ...);
15   au_config("MinNN", DNN, AdamOpt, 6, ...);
16   au_extract("IMG", 62500, image);
17   au_NN("SigmaNN", "IMG", "SIGMA");
18   au_write_back("SIGMA", 1, &sigma);
19   sImg = smooth(image, sigma);
20
21   // 2. Magnitude computation
22   mag = magnitude(sImg);
23
24   result = hysteresis(mag);
25 }

```

Fig. 4.11.: Canny. Automonizing with the *Min* version. The highlighted statements are added.

both *Raw* and *Med* have 62500 neurons whereas *Min* has 32768 neurons. They differ because *Min* uses `hist` for prediction, which is of a smaller size than `image` and `sImg` used by *Raw* and *Med*.

We use the images from [104] to train the neural networks with SL. We use 10 images from [102] for testing which are associated with the ground truth specified by experts. Better result has a higher score (the SSIM score [33]).

Fig. 4.12 shows the test scores of baseline, *Raw*, *Med*, and *Min*. Each model is trained around 30 epochs. On average, the improvement of *Min* over baseline is 70%, which clearly shows that AUTONOMIZER significantly improves the quality of the result. Meanwhile, the improvement of *Raw* and *Med* over the baseline is around

20% and 53%. It shows that Algorithm 2 (in particular, the ranking) is useful for extracting the most relevant feature variables.

Fig. 4.13 shows the change of the score along with the increase of the number of the training epochs. *Min* consistently has higher scores than all the rest versions. Furthermore, as shown in Table 4.3, the training time of *Min* is about half of *Raw* and *Med*, which is because the feature variables it adopts have a smaller size.

Fig. 4.14 shows a list of sample images denoting the edge detection results. Clearly, *Min* provides the outcome most similar to the ground truth.

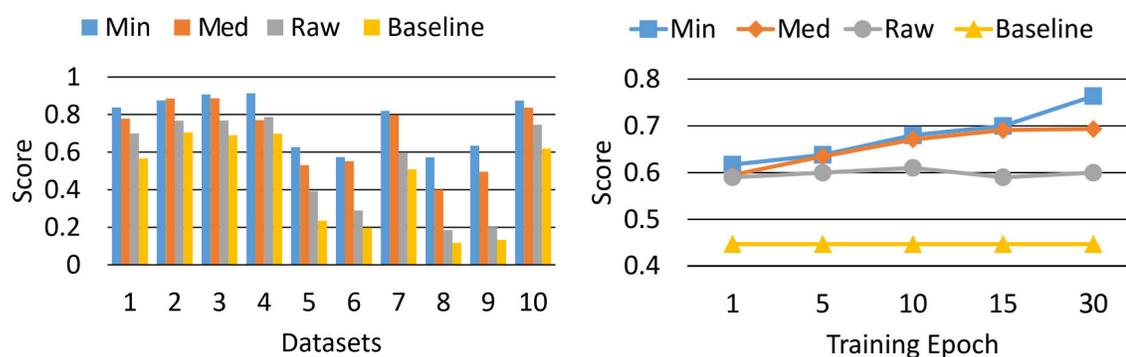


Fig. 4.12.: Canny predictions of 10 datasets

Fig. 4.13.: Canny prediction score variation

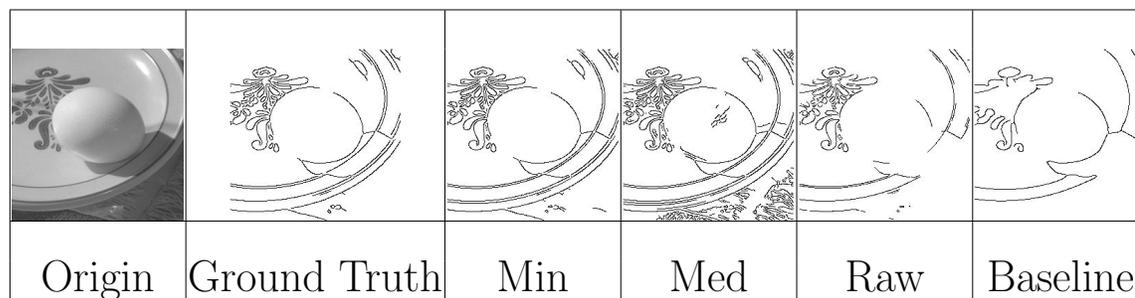


Fig. 4.14.: Canny results

**TORCS.** TORCS [105] is an open source C++ 3D car racing simulator. Many works [106–108] use it to study the application of reinforcement learning to self-driving cars. In this study, we autonomize TORCS by using the program internal

states AUTONOMIZER automatically extracts as the feature variables. Comparatively, existing works either use the manually extracted features [80], or the raw image features [79].

*Ease of Use.* To autonomize Torcs for the *All* version, we annotate the variable `steer` for steering control as the target variable, which determines the turning of wheel. We run Algorithm 3 by setting  $\epsilon_1$  to zero and  $\epsilon_2$  to 0.01. If the traced values of two variables are similar (i.e., the euclidean distance  $\leq \epsilon_1$ ), we can prune one of them. As shown in Fig. 4.15, the traced values of the candidate feature variables `posX` and `roll` are almost the same ( $EucDict(posX, roll) \approx 0$ ), so `roll` is pruned. Besides, we prune the candidate feature variables whose values rarely change. For example, as shown in Fig. 4.16, the variable `accX` is pruned because the variance of its values is  $\sim 0.007$ , which is less than  $\epsilon_2$  (0.01). In total, twenty feature variables are automatically extracted. The annotation is similar to Mario (Section 5.2) and hence elided.

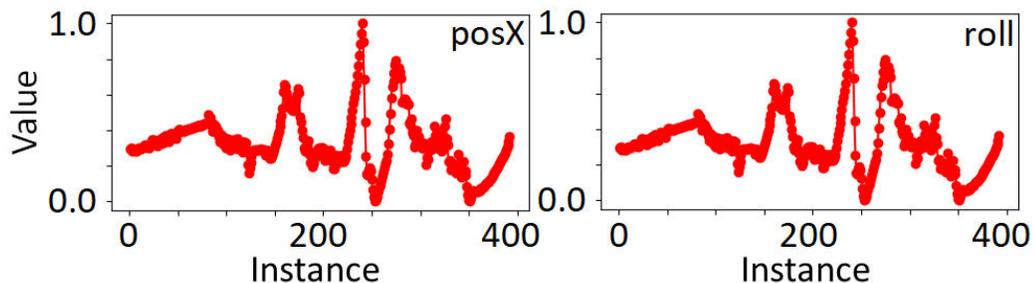
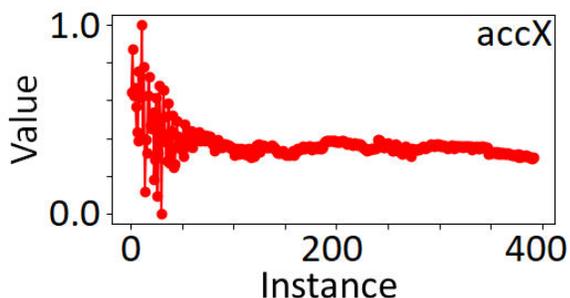


Fig. 4.15.:  $EucDict \approx 0$

*Usefulness.* To demonstrate how autonomization helps self-driving without human intervention, we compare four settings: *Players*, which represents the average score from 10 human players, *Raw*, which uses an existing model [79] that takes screenshots as the input data, *All*, which is our version, and *Manual*, which uses an expert model [80] with manually extracted/preprocessed program variables as the input data. All models have the same output consisting of three actions: left turn,

Fig. 4.16.: Variance  $\approx 0.007$ 

right turn, and no turn. We compare the results using the following criterion: how far the car drives without bumping to the wall before finishing.

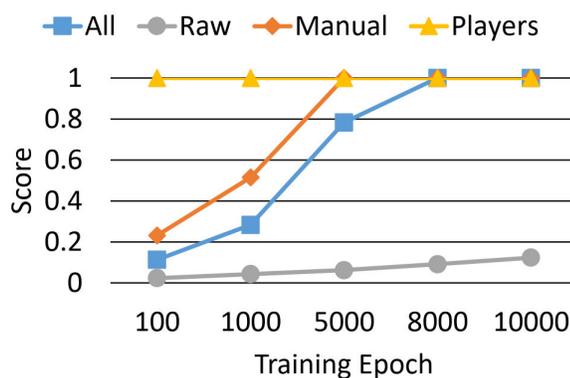


Fig. 4.17.: Driving score

Fig. 4.17 presents the scores of all settings after training for the same number of epochs. It also includes the average of 10 human players as a reference. Observe that *All* consumes around 8000 epochs (20.3 hours) and *Manual* consumes 5000 epochs (14.5 hours) to have close-to human performance. Although *Manual* learns quicker than *All*, it requires non-trivial human effort ( $\sim 2000$  lines of code) to extract and preprocess input data. On the other hand, AUTONOMIZER only uses 89 lines of code as shown in Table 4.1.

For *Raw*, after training for 10000 epochs ( $\sim 40$  hours), it still performs bad. Furthermore, the improvement is really slow. According to the author [79], it takes

around 200000 epochs for *Raw* to learn reasonable behavior. Each epoch consists of 100 neural network updates.

## 4.7 Summary

In this chapter, we propose AUTONOMIZER, a novel framework for much more efficient parameter selection. AUTONOMIZER leverages artificial intelligence (AI) to autonomize existing software systems that require human interactions/interventions. It features a novel execution model facilitated by several programming primitives. The developer can instruct an AI model to learn from the normal operations of the software by adding few invocations to these primitives in the source code. AUTONOMIZER then transparently weaves the model training and deployment into program execution, hiding all the complexities such as collecting data, extracting features, and replacing user interactions/interventions with the AI model. Our experiments on nine real world programs show that very little human effort is required to autonomize these programs with AUTONOMIZER. The autonomized versions produce results with higher quality. Autonomized games can play by themselves and have performance competitive with human players with less training time. In the next chapter, we will discuss how we further improve the software autonomization result by leveraging both program analysis and statistical analysis.

## 5. SPSA: STATISTICAL AND PROGRAM ANALYSIS AIDED SOFTWARE AUTONOMIZATION

In the previous chapter, we have discussed our software autonomization framework `AUTONOMIZER` that improves the efficiency of data processing programs parameter selection. However, `AUTONOMIZER` only leverages program analysis and simple heuristic for feature variables selection, leading to sub-optimal output quality and long model training time. In this chapter, we propose a novel approach `SPSA`, that is implemented as an extension to the library of `AUTONOMIZER`. It combines both program analysis and statistical analysis for much better feature variables identification. Our experiments show that we are able to improve ten real world data processing applications including edge detection and speech recognition. On average, `SPSA` improves the output quality by 99.04% over the baseline while the improvement of `AUTONOMIZER` is only 80.24%. Furthermore, the training time of `SPSA` is 27.44X lower than `AUTONOMIZER` on average.

### 5.1 Introduction

Many traditional software systems require human interventions. For example, scientific data processing programs are often parameterized. Human experts have to configure a set of parameters for given inputs [109] because different inputs require different configurations to achieve the optimal results. Complex systems such as compilers, symbolic execution engines, browsers, streaming servers [110], unmanned vehicles (UxV) often require substantial human efforts to identify the configuration to deliver the optimal performance or to properly address cross-cutting concerns such as security [111] and usability. For example, a symbolic execution infrastructure `KLEE` [112] has different path exploration strategies suitable for different kinds of applications;

a commercial-off-the-shelf (COTS) UxV control software system Ardupilot [48] has more than 600 configurable parameters. Different parameter settings are desired for various environmental conditions (e.g., high altitude and wind gusts) and mission objectives (e.g., optimizing energy consumption or mission completion time). Many end-user applications (e.g., third party mobile apps and computer games) are UI driven. User interactions are needed to perform normal functionalities.

In many cases, it is particularly desirable to autonomize these software systems. First, it enables full automation. Human efforts (to interact with these software) are no longer needed or substantially reduced such that optimal performance can become easily achievable and human mistakes are largely avoided. Note that compared to human decisions, machine decisions can be much faster, more accurate, and more rigorous as they can be inferred from much more comprehensive information that is often beyond human capabilities to collect and analyze. For example, it would be highly desirable for scientists to have a bio-data processing program that can automatically select the optimal parameters for processing a given input. But such setting is often dependent on features of the input data that are difficult for humans to apprehend. Second, the operation of an interactive program is driven by human and hence limited by human capacities. For example, testing a UI software often requires human testers to interact with the software in a way that follows the work flow. The pace of testing is hence limited by human speed. Autonomizing a system to allow it to self-test could substantially accelerate the development cycle. The example of AlphaGo [113] illustrates that without human intervention, autonomous computation systems can train themselves in a much faster pace and eventually supersede humans. Third, autonomized software can still interact with human users by providing assistance/instructions to users on how to properly and most efficiently use the software under various conditions.

In most existing autonomous systems [66, 82, 83], autonomization is achieved by design. The AI/ML component is a key component to begin with. In such a development procedure, training the AI/ML component is a stand-alone step that requires

a lot of (human) efforts. For example, in order to construct a model by supervised learning, inputs need to be collected and the corresponding expected outputs need to be crafted. Model structure (e.g., number of layers and number of neurons for each layer for a neural network model) needs to be carefully chosen to extract important features from raw inputs (e.g., images and videos).

Such a heavy-weight procedure is difficult for autonomizing existing software that does not have autonomization in consideration during design. For example, in order to autonomize the Mario game, a traditional method would require taking screenshots of the game, using neural networks to extract critical features such as the position/speeds of Mario, and Goombas, and the height of brick walls. This task itself is highly challenging. Even if the model could be successfully trained, injecting the model into the software requires substantial additional efforts.

### 5.1.1 Existing Software Autonomization

AUTONOMIZER [88] is the first framework proposed to achieve software autonomization. It is mainly achieved by adding instrumentations to the original software. AUTONOMIZER provides a number of primitives that make a lot of entailed tasks such as feature variable extraction, input data collection, model training, and model integration transparent and substantially simplified. After autonomization, the software can automatically operate without human inputs.

It is a general framework for autonomizing existing software systems including a widely used edge detection program Canny [29] and a speech recognition program Sphinx [41]. It only requires adding a few lines in the original source code. The instrumented program automatically extracts a number of program variables as model input feature variables, which is used to train the model for target variable (parameter) prediction. The key observation is that the regular operation/execution of the original software provides very rich information in the program variables from which AI/ML models can learn. For example, to autonomize the speech recognition pro-

gram Sphinx (Section 5.4.3), instead of selecting the raw speech as a feature variable, the user may select the program variable that stores the data produced by Fourier transform as the feature variable. Intuitively, the audio is being transferred into an alternate representation by FFT, i.e., from time domain to frequency domain. Different audios have different frequency information and hence these program variables serve as important features.

While AUTONOMIZER have achieved a certain level of success, it suffers greatly from several problems:

- Only program analysis and simple heuristics are leveraged to identify program feature variables, leading to sub-optimal feature variables selection and output quality.
- The identified program feature variables are usually huge and contain redundant values. Both factors make neural network unreasonably large and increase the training time because the neural network requires a large number of input neurons as shown in Fig. 5.1.

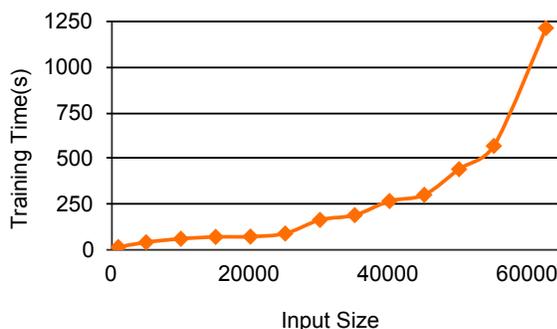


Fig. 5.1.: Time cost – Input Size

### 5.1.2 Our Work

In this paper, we propose a novel feature variable selection approach called SPSA. It not only leverages program analysis but also statistical analysis to find important

program variables as feature variables to predict target variables (parameters). Specifically, we first identify program variables as candidate feature variables with program dependency analysis. We then apply data reduction to eliminate redundant values from the candidate feature variables. Intuitively, many program variables are multi-dimensional vectors so data reduction reduce the input vector by either selecting the min/max value or taking the average or variance value along the specified dimension. Furthermore, we leverage statistical correlation analysis to find statistical dependence between each reduced feature variable and target variable (parameter). Afterwards, a set of more relevant and important feature variables is selected through data mining algorithm for each target variable.

Although AUTONOMIZER supports programs using supervised learning and reinforcement learning, we are currently focusing on improving the results of data processing programs suitable for supervised learning. The reason is that reinforcement learning programs usually have no input data like data processing programs while our approach requires the program analysis information from input data. Furthermore, the program internal feature variables used in supervised learning programs are normally vectors which contain more noise or redundant values compared to reinforcement learning programs.

### 5.1.3 Contributions

The following shows the contributions we made:

- We propose a novel approach *SPSA* to identify important program variables as neural network input feature variables. As discussed above, it leverages both program analysis and statistical analysis.
- We develop *SPSA* by providing several primitives to extend current *Autonomizer* library. It allows the users to take advantage of both program and statistical analysis by adding a few lines of code.

- We substantially improve data processing results by improving the feature variables selection on ten widely used parameterized programs. The output quality is improved by 99.04% on average over the baseline with execution overhead almost the same as AUTONOMIZER. Comparatively, AUTONOMIZER only improved the output quality by 80.24% on average over the baseline.
- As shown in the experiments (Section 5.4), *SPSA* largely reduces the training time and size of the neural network models compared to *Autonomizer*. For *SPSA*, its model training overhead is 27.44X lower than AUTONOMIZER and its model size is 603.5X smaller than *Autonomizer* on average.

## 5.2 Motivation

In this section, we show how to further improve the autonomization result of *Canny* [29], a popular edge detection algorithm. *Canny* carries the parameters that affect the quality of the edge detection result. Each input image for *Canny* requires a specific parameter configuration in order to achieve ideal edge detection result. It is a representative of a large set of data-processing software applications that require parameter configurations for different inputs. We will explain how to improve the autonomization result by improving the parameters prediction result with supervised learning using our newly proposed primitives.

***Extension Primitives.*** The primitives are listed in Fig. 5.2. They are essentially extensions of the AUTONOMIZER original library. We will explain those primitives when they are used in the motivating example. More details of our design will be discussed in Section 5.3.

***Running Example.*** In this section, we show how to improve the autonomization result of the *Canny* edge detection algorithm, such that it is able to achieve ideal parameter configurations automatically on the fly. In Fig. 5.3, we first demonstrate how to autonomize *Canny* with original primitives, then we show how to improve the

<p><i>Library Calls :</i></p> <p><code>@au_reduce(reductionName, extName)  </code></p> <p><code>@au_stats(wbName1, wbName2, ...)  </code></p> <p><code>@au_opt(optName, wbName)</code></p>
--

Fig. 5.2.: Primitives

autonomization result using our proposed primitives. All primitives start with *au...* The new primitives proposed by *SPSA* are highlighted.

**Canny** has four main computation stages. The first is *Gaussian smoothing* stage (line 32 in Fig. 5.3). This stage removes noise from the input image. The second is *image transformation* stage (line 39). It is responsible for image gradient computation and performing non-maximal suppression. Moreover, the *edge traversal* stage (line 46) leverages the hysteresis analysis to track all potential edges in the image. Finally, the *visualization* stage writes final edge detection result to a file.

As a typical data processing program, **Canny** takes three input parameters: **sigma**, **low**, and **high**. Specifically, the parameter **sigma** is used during the Gaussian smoothing stage. The higher the value is, the more the noise will be filtered. The **low** and **high** parameters are two important thresholds used during the edge traversal stage.

To autonomize **Canny**, AUTONOMIZER can help to predict the these input parameters. For example, to predict the target variable (i.e., the output of the neural network model) **sigma**, the user first specifies it with the primitive *au\_write\_back()* at line 29. Then some program variables are extracted automatically by AUTONOMIZER as the feature variables (i.e., the inputs of the model), which are also annotated with the primitive *au\_extract()* (e.g., *image* at line 23). The feature variable, i.e., the original image contains the relevant information for predicting the target variable **sigma**.

To use *SPSA* for further improving the prediction result (e.g., for better edge detection result) of **sigma**, the user specifies to reduce the feature variable along the

```

1 char* hysteresis(gradient, low, high){
2   /* Compute histogram */
3   hist = computeHist(gradient);
4
5   au_extract("HIST", hist.row * hist.col, hist);
6   au_reduce("VEC_HIST", "HIST");
7
8   au_stats("LO", "HI");
9   au_opt("OPT_LO", "LO");
10  au_opt("OPT_HI", "HI");
11  au_NN("LOW_HIGH", "OPT_LO", "OPT_HI",
12       "LO", "HI");
13  au_writeback("LO", &low);
14  au_writeback("HI", &high);
15
16  return do_hysteresis(hist, low, high);
17}
18
19 void canny() {
20   /* Initialization */
21   image = readInput(imageName);
22
23   au_extract("IMAGE", image.row * image.col, image);
24   au_reduce("VEC_IMAGE", "IMAGE");
25
26   au_stats("SIGMA");
27   au_opt("OPT_SIGMA", "SIGMA");
28   au_NN("NN_SIGMA", "OPT_SIGMA", "SIGMA");
29   au_writeback("SIGMA", &sigma);
30
31   /* Smoothing the image */
32   smoothImg = gaussianSmooth(rawImg, sigma);
33
34   au_extract("SIMG", smoothImg.row * smoothImg.col,
35             smoothImg);
36   au_reduce("VEC_SIMG", "SIMG");
37
38   /* Compute the magnitude of gradient */
39   gradient = magnitude(smoothImg);
40
41   au_extract("GD", gradient.row * gradient.col,
42             gradient);
43   au_reduce("VEC_GD", "GD");
44
45   /* Track edge by hysteresis */
46   finalImg = hysteresis(gradient, low, high);
47   writeOutput(finalImg);
48}

```

Fig. 5.3.: Autonomizing Canny. The highlighted statements are provided by SPSA.

The primitives start with au.

dimensions given in `axis` with the primitive `au_reduce()` at line 24. We provide several reduce functions such as `avg` and `sum`. After reduction, the reduced feature variable (i.e., `VEC_IMAGE`) is stored in the database. The user then specifies to compute the statistical score between `sigma` and currently stored feature variables (e.g., `VEC_IMAGE`) with the primitive `au_stats()` at line 26 using the training samples. At line 27, the primitive `au_opt()` selects and combines a set of the most important and relevant variables as the final feature variable named `OPT_SIGMA` to predict `sigma`. Note that `OPT_SIGMA` is a name used to index the final feature variable `img_min`. We refer the readers to Section 5.3 for the feature variables selection approach.

When the program reaches the primitive `au_extract()`, e.g., at line 23, the values of the feature variables will be extracted and stored into a database maintained by the AUTONOMIZER runtime. Different names are assigned to different feature variables for future reference.

When reaching the primitive `au_NN()` at line 28, the program interacts with the neural network. The primitive has two modes: the training mode and the testing mode. In the training mode, the extracted values of the feature variables are used to train the model. In the testing mode, the extracted values are used to generate the predicted value. Note that the neural network uses the feature variable `OPT_SIGMA` generated through lines 24-26 to predict `sigma`. The predicted value is stored in AUTONOMIZER's database and can be retrieved with the name `SIGMA`.

When the program reaches the primitive `au_write_back()`, e.g., at line 29, the AUTONOMIZER runtime updates the target variable `sigma` with the predicted value of the target variable. The predicted `sigma` is used during Gaussian smoothing stage at line 32.

To predict `lo` and `hi`, the user then specify them with the primitive `au_write_back()` at lines 14-15. Similarly, AUTONOMIZER will automatically extract feature variables like `smoothImg` and `gradient` annotated at lines 34 and 41. The remaining is similar to predicting `sigma`, so details are elided. Note `smoothImg` and `gradient` can only

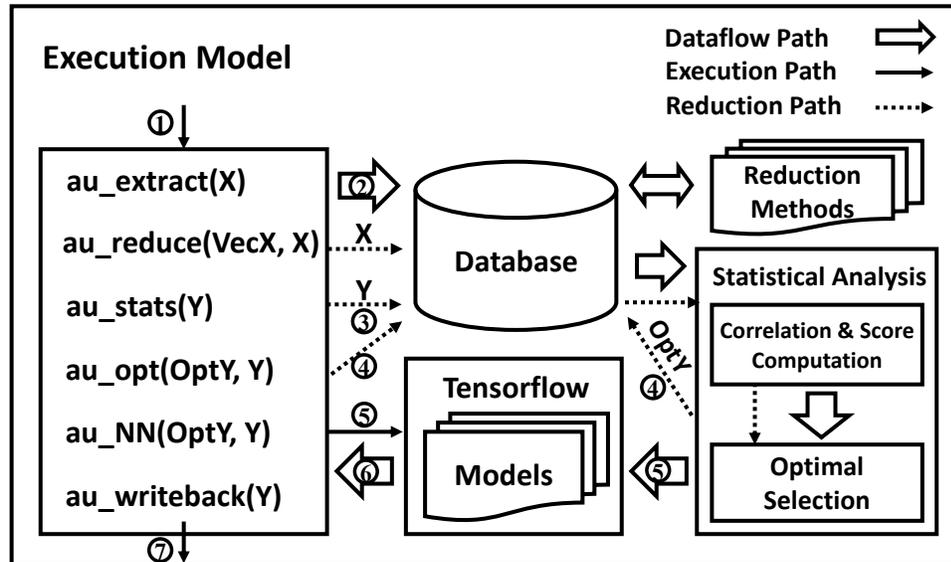


Fig. 5.4.: Execution Model

be used to predict target variables `lo` and `hi` instead of `sigma`. The reason is that they transitively depend on `sigma`.

**Execution Model.** Fig. 5.4 shows the simplified execution model. First, the user specifies to predict target variable `Y` with an annotation, and SPSA selects the program variable `X` as the neural network input feature variable. The user then annotates `X` in the program. In addition, three more annotations (i.e., `au_reduce()`, `au_stats()`, and `au_opt()`) are added to find the most relevant feature variables `OptY` as neural network input for predicting `Y`.

The following shows the runtime execution. In the beginning, the main process executes normally until reaching the `au_extract()` primitive (①). At this point, the value of the feature variable `X` is stored in the database (②) maintained by SPSA. When the program reaches the `au_reduce()` primitive, the feature variable `X` is down-sampled with different types of reductions and the result `VecX` is stored in the database. When the program reaches the primitive `au_stats()` (③), the statistical correlations w.r.t `Y` of different down-sampled feature variables indexed by `VecX` are then computed. The final feature variable indexed by `OptY` is then selected by the

optimal selection algorithm through the primitive `au_opt()` (④) and stored in the database for later reference. When the main process reaches the primitive `au_NN()`, SPSA transfers the execution to the Tensorflow Python code (⑤) to interact with the model. The feature variable `OptY` is fed to the model to predict and store the value of target variable `Y` to the database. Finally, the execution transfers back to the source program and the primitive `au_writeback` loads the value from database and updates the source program variable `Y`.

**Result and Comparison.** Fig. 5.5 shows the execution results of the running example in Fig. 5.3. Furthermore, we compare our edge detection result with AUTONOMIZER.

Initially, the original 2-dimensional image in Fig. 5.5.a is extracted as the feature variable at line 23. At line 24, different 1-dimensional reduced variables are produced by the reduce functions in Fig. 5.5.b. Afterwards, the statistical score of reduced feature variables w.r.t. `SIGMA` are calculated and stored in a score table as shown in Fig. 5.5.c. Note that the scores of all currently stored feature variables will be computed. There are four groups (① - ④) of reduced feature variables in Fig. 5.5.c.

Finally, in Fig. 5.5.d, a set of feature variables are selected and combined by our variable selection algorithm to predict target variables. The higher the score, the better the feature variables for target variable prediction. For example, variables `mag_min`, `hist_avg`, and `hist_sum` are selected to predict target variables `high` and `low` in the model `NN_LOW_HIGH`. Observe that the three selected variables are in the same tiger orange color as shown in Fig. 5.5.c.

To achieve fair comparison, we use the same neural network structure as AUTONOMIZER except the input layer. In particular, we use a fully connected neural network with 6 inner layers for each model. Furthermore, for input layer, we use 250 neurons for model `NN_SIGMA` and 2 neurons for model `NN_LO_HIGH`. We then compare our edge detection results with results produced by the best model of AUTONOMIZER.

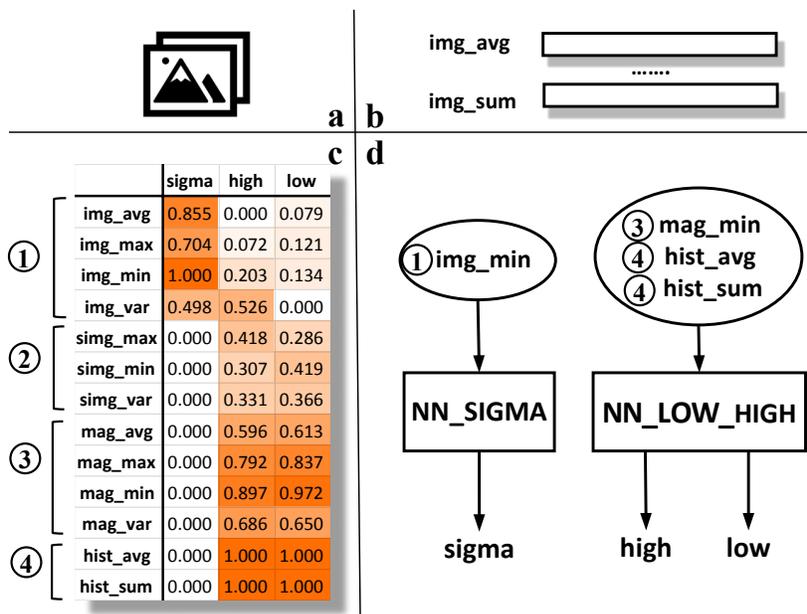


Fig. 5.5.: Execution Result

The results are shown in Fig. 5.6. We show the visual results and the SSIM [33] scores for comparison. Observe that both results demonstrate that SPSA substantially improves the edge detection result of AUTONOMIZER.

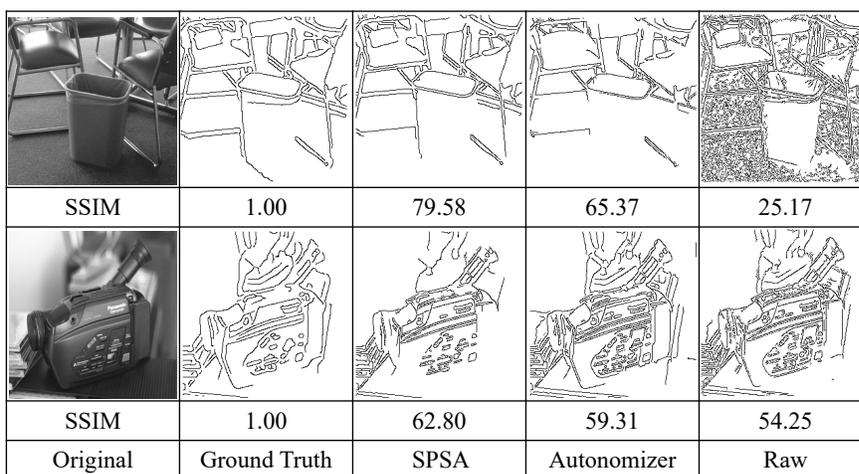


Fig. 5.6.: Canny Results

### 5.3 Design

In this section, we propose an automatic way to find the most relevant feature variables as the inputs of neural network models to predict target variables. Our method combines program analysis as well as statistical analysis and takes advantage of both worlds.

#### 5.3.1 Overview

Generally speaking, our purpose is to substantially improve the prediction results of the program parameters values (i.e. *target variables*) with better program internal states (i.e. *feature variables*) than AUTONOMIZER.

Our feature variables selection method consists of four phases:

- *Candidate Variables Reduction*: Apply data reduction on each candidate feature variable in order to remove redundant values.
- *Distance calculation*: Calculate the distance between a candidate feature variable and a target variable.
- *Score Computation*: Compute score of each reduced candidate feature variable w.r.t target variables based on program dependency and statistical analysis.
- *Final Feature Variables Selection*: Select a set of most relevant feature variables to predict each target variable using data mining algorithm.

In the next section, we will discuss each phase in details about identifying more correlated feature variables for better prediction results.

#### 5.3.2 Automatic Feature Variables Selection

In the following, we give the overall workflow of our feature variables selection method. First, like AUTONOMIZER, we identify the input variables and program

variables that transitively depend on them as the candidate feature variables. Then for each candidate variable, we apply data reduction on it. Intuitively, data reduction helps to eliminate redundant values by down-sampling a multi-dimensional vector. Then for each candidate feature variable, we calculate the distance between a candidate feature variable and a target variable. The shorter the distance, the more important the feature variable. Note that a reduced feature variable shares the same distance with the original feature variable. Then for each reduced feature variable, we compute a score w.r.t each target variable according to their statistical correlation and distance. Finally, a set of feature variables with similar scores are clustered in one group. *SPSA* then selects the set of feature variables with highest average score as the ultimate set of feature variables to predict target variable. According to our experiments in Section 5.4, the refinement substantially improves the prediction results compared to *AUTONOMIZER*.

Next, we explain the automatic feature variables selection and discuss each phase of feature variables selection in details.

Algorithm 1 has three inputs:  $In$ ,  $Trg$ , and  $G\_Dep$ .  $In$  is the set of input variables,  $Trg$  is the set of target variables, and  $G\_Dep$  is the dynamic dependency graph computed previously. First, we construct the initial program variables set, which consists of the input variables and their transitive dependents. Then we construct the Feature map that maps a target variable to its selected feature variables. The Feature map is returned as the algorithm output.  $Candidate$  is a set of candidate feature variables, which is initialized to empty.  $Matrix$  is a scoring table that stores the computed score between each candidate feature variable and each target variable as shown in Fig. 5.5.c.

***Candidate Variables Reduction.*** For each multi-dimensional initial program variable  $v$ , we first apply data reduction on it for redundant values elimination (line 6). The variable  $v$  is reduced by one dimension with five reduction methods including  $min$ ,  $max$ ,  $sum$ ,  $avg$ , and  $var$ . The reduced variables are stored in a set  $Reduced$  for later reference.

---

**Algorithm 4** Automatic Feature Variables Extraction
 

---

**Require:**  $In, Trg, G_{Dep}$ 
**Ensure:**  $Feature$ 

```

1:  $ProgVars \leftarrow In \cup dep(In)$ 
2:  $Feature \leftarrow Map()$ 
3:  $Candidate \leftarrow \emptyset$ 
4:  $Matrix \leftarrow Zero$ 
5: for each  $v \in ProgVars$  do
6:    $Reduced \leftarrow reduce(v)$ 
7:   for each  $p \in Trg$  do
8:     if  $dep(p) \cap dep(v) = \emptyset$  or  $v \in dep(p)$  then
9:       continue
10:     $dist \leftarrow BFS(G_{Dep}, v, first(dep(p) \cap dep(v)))$ 
11:    for each  $v' \in Reduced$  do
12:       $Candidate \leftarrow Candidate \cup \{v'\}$ 
13:       $dCorr \leftarrow DCorr(Traces(p), Traces(v'))$ 
14:       $Matrix(p, v') \leftarrow exp(-sqrt(dist)) \times dCorr$ 
15: for each  $p \in Trg$  do
16:    $Feature[p] \leftarrow Clustering(Matrix, p, Candidate)$ 
17: return  $Feature$ 

```

---

**Distance Calculation.** Next, we are going to calculate the shortest "distance" between the feature variable  $v$  and each target variable  $p$  by leveraging AUTONOMIZER [88]. The distance is defined as the number of edges on data dependency graph between two nodes that represent variables (e.g., program feature variable and target variable). In line 10, the shortest distance  $dist$  from feature variable  $v$  to the common descendent of  $v$  and  $p$  is found by BFS on the dependency graph  $G_{Dep}$ . However, for each candidate program variable  $v$ , if it does not share some common

dependent with a target variable  $p$ , then  $v$  and  $p$  are not considered correlated according to AUTONOMIZER. For such case, we just skip current target variable  $p$  and check next one (lines 8-9). For prediction purpose,  $v$  is also not considered as feature variable if it depends on  $p$ .

**Score Computation.** In this phase, a statistical score of each reduced feature variable  $v'$  and target variable  $p$  is calculated (lines 11-14). We first use *distance correlation* [114] (*dCorr* for short) to measure the statistical dependence between the target variable and the candidate feature variable. Instead of using Pearson correlation [115], we use distance correlation because it can find the statistical dependence between two vectors. For distance correlation, zero means no dependence. In line 13, we first collect runtime traces  $Traces(v')$  and  $Traces(p)$  of  $v'$  and  $p$  respectively. We collect the traces by running the program with different training samples to train the neural network model such that we can get two vectors for  $p$  and  $v'$  respectively for each training sample. Then we calculate the distance correlation *dCorr* between the two.

After calculating the distance correlation, we compute the score of reduced feature variable  $v'$  w.r.t target variable  $p$  and update the scoring table *Matrix* accordingly (line 14). The score is computed with the previously calculated distance *dist* and correlation *dCorr* information using the following formula where  $\Gamma(x) = e^{-\sqrt{x}}$ .

$$Score(p, v') = \Gamma(dist(p, v')) \times dCorr(p, v') \quad (5.1)$$

Intuitively,  $\Gamma(dist(p, v'))$  is a decay factor that prefers feature variables  $v'$  with shorter distance to the target variable  $p$ . Thus, the score is produced by multiplying the distance correlation of a feature variable  $v'$  with the factor. For each target variable  $p$ , it helps to identify those feature variables  $v'$  with shorter distance and higher statistical dependence.

**Final Feature Variables Selection.** After updating the scoring table *Matrix*, we select a set of feature variables for each target variable. In lines 15-16, feature variables with similar computed scores are clustered together. Group with the highest

average score are selected as the final set of feature variables to predict target variable  $p$ . For example, in Fig. 5.5.c, feature variables *mag\_min*, *hist\_avg*, and *hist\_sum* are clustered together. Furthermore, they are selected as the final set of feature variables to predict *high* and *low*.

## 5.4 Evaluation

Our technique is built on top of AUTONOMIZER. The feature extraction algorithm is implemented with Valgrind-3.15.0 [100] and Python. Experiments are conducted on a machine with Intel i7-6700HQ 2.60GHz processor, 32G RAM. Model training tasks are run on GPU NVIDIA Geforce GTX 970 with 3G RAM. To show the validity and advantages of our proposed approach, we select a variety of programs from different fields for evaluation. The datasets for evaluation are found on the Internet and/or come with the program.

In Section 5.4.1, we show the statistical results of our approach, such as input size and neural network model size. In Section 5.4.2, we present the effectiveness of SPSA by comparing SPSA, *Autonomizer* and baseline. In Section 5.4.3, we conduct three case studies to provide more insights.

### 5.4.1 Statistics

Table 5.1 shows the model statistics including (1) the size of the model input (i.e., the number of input neurons), (2) the collected trace, and (3) the size of the model. Note that all models are trained using Tensorflow.

To show the effect of our proposed algorithm, we compare three settings: *Raw* (Columns 2-4), *Autonomizer* [88] (Columns 5-7), and *SPSA* (Columns 8-10). *Raw* uses original input of program as the input of neural network models. *Autonomizer* selects internal variables as the input of models using the minimum distances on the program dependency graph. *SPSA* selects the feature variables based on program and

Table 5.1.: Statistics of models and feature variables

Program	Raw			Automomizer			SPSA			Automomizer/SPSA		
	Input	Trace	Model	Input	Trace	Model	Input	Trace	Model	Input	Trace	Model
	Size	Size(MB)	Size(MB)	Size	Size	Size	Size	Size	Size	Size	Size	Size
[29] Canny	62,500	48.0	215.0	95,268	14.1	328.0	252	2.000	2.100	378	7.1	156.2
[116] Rothwell	62,500	49.0	215.0	125,000	143.0	429.0	750	1.200	3.900	166.7	119.2	110.0
[117] Sobel	250,000	105.0	859.0	250,000	105.0	859.0	1	0.004	0.480	250,000.0	26,923.1	1,789.6
[118] Watershed	250,000	87.0	859.0	250,001	87.0	859.0	1,502	0.639	0.670	166.4	136.1	1,282.1
[119] Colorseg	62,500	24.0	215.0	125,000	40.0	430.0	500	0.363	2.600	250.0	110.1	165.4
[120] Boruvka	62,500	24.1	215.0	125,000	40.0	430.0	250	0.246	1.300	500.0	162.5	330.8
[42] Phylip	6,000	4.0	21.4	6,420	1.2	22.9	21	0.107	0.742	305.7	11.2	30.9
[45] Metis	6,000	6.7	21.0	3,000	3.4	11.0	2	0.003	0.873	1,500.0	1,062.5	12.6
[121] Facerec	372,000	317.0	1278.0	372,001	317.0	1278.0	62	0.066	0.598	6,000.0	4,774.1	2,138.2
[41] Sphinx	262,144	157.0	901.0	4,096	36.0	15.0	2	0.016	0.770	2,048.0	2250.0	19.5

statistical analysis. For fair comparison, we use the same neural network architecture and hyper-parameter settings for all settings except for the input layer which accounts for different input size.

In Columns 11-13, we show the ratio of SPSA and AUTONOMIZER regarding input size, trace size, and model size. For programs using multiple models to predict different parameters (i.e., target variables), the input size is the sum of the input size from all models. The trace size and model size are similar.

Observe that AUTONOMIZER consistently has larger input size, trace size and model size compared to SPSA. According to column 11, the model input size of AUTONOMIZER is 166X-250000X larger than SPSA. Column 12 shows that the collected trace size of AUTONOMIZER is 7X-26923X times the trace size of SPSA. Since the input layer of AUTONOMIZER contains a large number of neurons, the model size of it is significantly larger than the model size of SPSA (12.6X-1789.6X larger).

#### 5.4.2 Effectiveness

In the following, we study the effectiveness of our proposed approach. Particularly, we want to demonstrate how our approach further improves the quality of the autonomization results compared to AUTONOMIZER. The results are shown in Table 5.2.

### Experiment Settings

First, we discuss the experimental settings of different results.

**Comparisons.** For the experiments, we compare four settings: *Baseline*, *Raw*, *Autonomizer*, and SPSA. *Baseline* stands for program execution with default parameter configurations and the other three settings are introduced in Section 5.4.1.

**Scoring.** We use scores to measure the quality of results. In Column 1, higher scores are better for programs marked with  $\uparrow$ , while lower scores are better for programs marked with  $\downarrow$ . Benchmarks either have ground truth or come with their own scor-

ing functions to compute the scores from the autonomization results with different settings.

**Time Cost.** For each setting, we present training time and execution time of all programs. For training time, we train the models until convergence. The execution time represents the time required for one execution.

## Experiment Results

Next, we discuss the efficiency and effectiveness of reducing and selecting program internal features extracted by SPSA with respect to training time, execution time and evaluation score.

**Training Time.** In Columns 13-14 of Table 5.2, training speed overhead ratio of *Raw* and *SPSA* ranges from 3.90X to 942.46X while the ratio of *Autonomizer* and *SPSA* ranges from 2.22X to 177.21X. In particular, for benchmark *Facerec*, both *Raw* and *Autonomizer* have significantly higher training overhead than *SPSA* because both settings selected feature variables with huge size. It shows that *SPSA* can substantially improve the model training time using much fewer input features.

**Execution Time.** In Column 15-16 of Table 5.2, execution overhead ratio of *Raw* and *SPSA* ranges from 0.90X to 1.21X while the ratio of *Autonomizer* and *SPSA* ranges from 0.91X to 1.40X. Observe that *SPSA* can reduce the execution time with fewer features. For some benchmarks, *SPSA* requires longer execution because it uses more models for predicting different target variables, which further requires more interaction time with the Tensorflow Python framework. However, the slowdown of *SPSA* is negligible.

**Evaluation Score.** The evaluation scores are shown in Columns 3, 6, 9, and 12. Observe that all settings outperform *Baseline*. It demonstrates the advantage of software autonomization. Among different settings, *SPSA* can substantially improve the quality of the results compared to the others settings. Specifically, *Raw*, *Autonomizer*, and *SPSA* improves the evaluation scores of *Baseline* by 56.33%,

80.24% and 99.04% on average respectively. This shows that our approach can not only improve the model training time but also the quality of the results by eliminating the model input data redundancy and noise to some extent.

### 5.4.3 Case Study

In this section, we discuss the details of autonomizing three representative programs in fields of edge detection, color segmentation and audio processing.

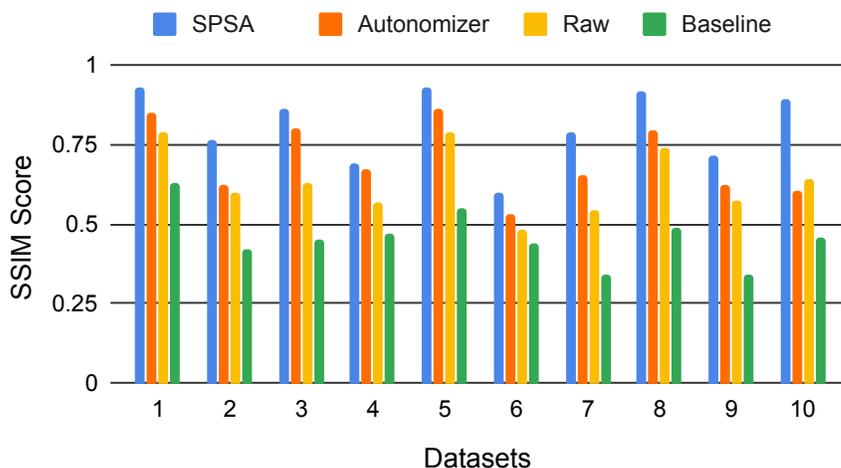


Fig. 5.7.: Canny prediction of 10 datasets

## Canny

As we have introduced in Section 5.2, **Canny** has three configurable parameters: `sigma`, `low` and `high`. `sigma` is used by Gaussian smoothing stage while `low` and `high` are used during edge traversal stage.

**Feature variables.** We create three different settings: *Raw*, *Autonomizer*, and *SPSA* to predict the target variables with different feature variables. As shown in Fig. 5.6.c, *SPSA* identifies `mag_min`, `hist_avg`, and `hist_sum` to predict target

Table 5.2.: Benchmark experimental results.

Program	<i>Baseline</i>		<i>Raw</i>		AUTONOMIZER		SPSA		Train O/H		Exec O/H			
	Exec. Time(s)	Score	Train Time(s)	Exec. Time	Score	Train Time	Exec. Time	Score	<i>Raw</i> /SPSA	AUTONOMIZER/SPSA	<i>Raw</i> /SPSA	AUTONOMIZER/SPSA		
[29] ↑ <sup>1</sup> Canny	1.32	44.69	1790.98	1.56	54.28	182.00	1.47	76.34	15.32	1.53	116.90	11.88	1.02	0.96
[116] ↑ Rothwell	1.15	49.31	1976.88	1.63	63.75	1564.36	1.53	70.49	63.39	1.65	31.19	25.15	0.99	0.93
[117] ↑ Sobel	1.12	64.69	230.00	1.81	80.54	178.53	1.80	83.03	6.39	1.49	35.99	27.94	1.21	1.21
[118] ↑ Watershed	1.03	59.85	140.59	1.34	63.17	127.60	1.38	69.66	11.80	1.23	16.15	10.81	1.09	1.12
[119] ↑ Colorseg	1.07	64.98	96.67	1.55	66.09	50.88	1.87	67.08	17.11	1.34	5.65	2.97	1.16	1.40
[120] ↑ Boruvka	0.64	71.29	97.70	1.90	80.54	57.80	2.26	85.03	9.69	1.86	10.08	5.96	1.02	1.22
[42] ↑ Phylip	1.48	1.01	47.01	2.21	0.96	5.56	2.15	0.54	2.18	2.20	21.56	2.55	1.00	0.98
[45] ↓ Metis	1.47	103.44	171.32	1.60	132.88	61.95	1.62	139.64	14.90	1.78	11.50	4.16	0.90	0.91
[121] ↓ Facerec	2.50	22.20	1847.23	3.28	20.30	347.33	3.75	18.60	1.96	3.25	942.46	177.21	1.01	1.15
[41] ↑ Sphinx	0.89	10.90	4047.80	1.53	57.20	142.21	1.41	63.22	24.69	1.32	163.94	5.76	1.16	1.07

1. ↑: Higher scores are better; ↓: lower scores are better.

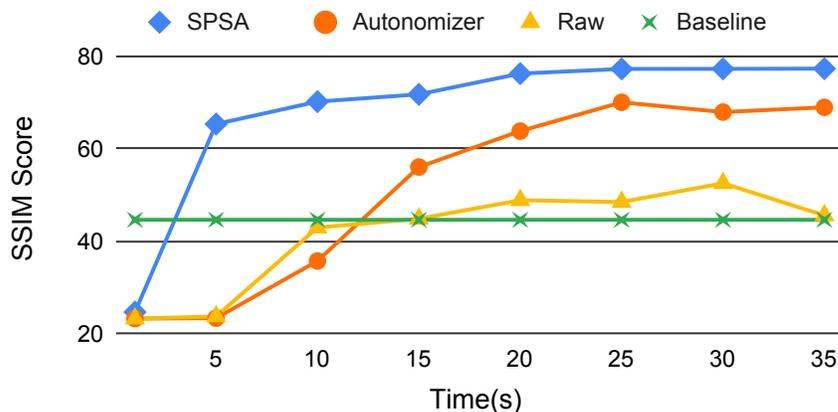


Fig. 5.8.: Canny training score variation

variable `low` and `high`. For other settings, *Raw* uses `image` while *Autonomizer* uses `hist` as model input feature variable respectively.

**Results.** To demonstrate how our approach further improves the data processing results, we show the results of baseline, *Raw*, *Autonomizer*, and *SPSA*. For fair comparison, all versions use the same neural network structure, i.e., a 6 layer fully connected neural network, except for the input layer. In particular, for input layer, *Raw* has 62,500 neurons, *Autonomizer* has 95,268 neurons, and *SPSA* has 252 neurons. The input neurons are different because different settings use different set of program variables as model input features.

For the experiment, we use the images from [104] to run *Canny* to train neural network models under each setting using supervised learning. For each setting, we apply the trained model to test 10 images from [102] for parameters prediction. We use the SSIM [33] score to compare the edge detection result with the ground truth hand-picked by experts. The higher score, the better quality of the result.

Fig. 5.7 shows the 10 edge detection scores of baseline, *Raw*, *Autonomizer*, and *SPSA*. Each model is trained until it converges. On average, the improvement of *SPSA* over baseline is 79%, which demonstrates that the quality of the result improvement is substantial. Observe that the improvement of *Raw* and *Autonomizer*

```

1 void watershed() {
2  /* Initialization */
3  pixs = pixRead(imageName);
4
5  au_extract("PIXS", pixs.row * pixs.col, pixs);
6  au_reduce("VEC_PIXELS", "PIXS");
7
8  au_stats("MAXMIN");
9  au_opt("OPT_MAXMIN", "MAXMIN");
10 au_NN("EXTREMA", "OPT_MINMAX", "MAXMIN");
11 au_writeback("MAXMIN", &maxmin);
12
13 /* Find local extrema */
14 mark = localExtrema(pixs, minmax, maxmin);
15
16 au_extract("MARK", mark.size, mark);
17 au_reduce("VEC_MARK", "MARK");
18
19 au_stats("MINDEPTH");
20 au_opt("OPT_MINDEPTH", "MINDEPTH");
21 au_NN("NN_MINDEPTH", "OPT_MINDEPTH", "MINDEPTH");
22 au_writeback("MINDEPTH", &mindepth);
23
24 /* Generate watershed */
25 wshed = wshedCreate(mark, mindepth);
26 pixWrite(wshed);
27 }

```

Fig. 5.9.: Autonomizing Watershed. The highlighted statements are provided by SPSA. The primitives start with au.

over baseline is around 21% and 71%. It shows that the combination of program and statistical analysis is useful for not only extracting the most relevant feature variables but also eliminating noise in the data.

In Figure 5.8, we illustrate the testing score variation with the training time in seconds. Observe that *SPSA* consistently has higher scores than the other settings. Furthermore, even 10-second training in *SPSA* yields similar results for 35 seconds training in *Autonomizer*. Observe that the score of *Raw* is quite unstable. The reason is that its model requires a large number of input features which makes it hard for the model to converge quickly compared to *SPSA*.

## Watershed

Watershed [118] is a widely used algorithm for image segmentation. It is used to separate different objects in an image. Watershed treats the input image as a topographic map, with the grayscale of each pixel representing its height, and seeks out the lines that run along the ridges. At first, we mark local minima in the image and these minima are regarded as "seeds" where the flooding will start rising. When the flooding fills in two adjacent basins, the algorithm decides whether a watershed should exist between these two. As the flooding stop rising, the algorithm stops and output a watershed picture of the original image.

In our case, we focus on two tunable parameters in Watershed: `maxmin` and `mindepth`. The parameter `maxmin` is the upper bound for local minima (i.e. the seeds mentioned above). It means that local minima with values greater than `maxmin` are not recognized as a real local minima. As the flooding rises and two adjacent basins become closer, the second parameter `mindepth` decides whether these two should be merged. For pixels between two basins, if the relative height of a pixel is higher than `mindepth`, then a watershed will be built on this pixel. Otherwise it will be submerged by the flooding. The function of these two parameters is to minimize the negative effect of noise or other factors that may lead to over-segmentation.

**Feature Variables.** We created three different model settings: *Raw*, *Autonomizer*, and *SPSA* to predict the target variables with different feature variables. Specifically, *Raw* is trained with raw input images. *Autonomizer* selects the raw image and mark information for local minima that stored in program variable `mark` as the feature variables to predict `maxmin` and `mindepth` respectively. For *SPSA*, it selects the reduced feature variables `pixs_sum` and `mark_max` to predict target variable `maxmin` and `mindepth` respectively. (Fig. 5.9, line 10 and line 21).

**Results.** To demonstrate how our feature extraction method improves the data processing results, we show the results of *Baseline*, *Raw*, *Autonomizer*, and *SPSA*. For fairness, all settings adopt the same neural network structure, i.e., a 6 layer fully

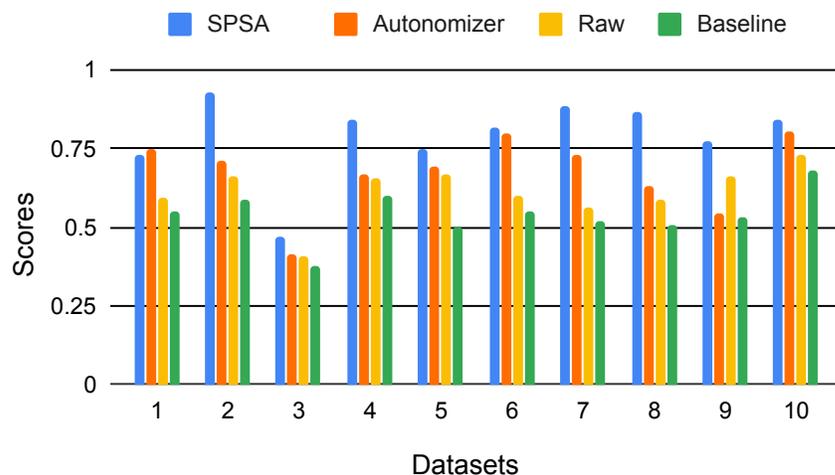


Fig. 5.10.: Watershed prediction of 10 datasets

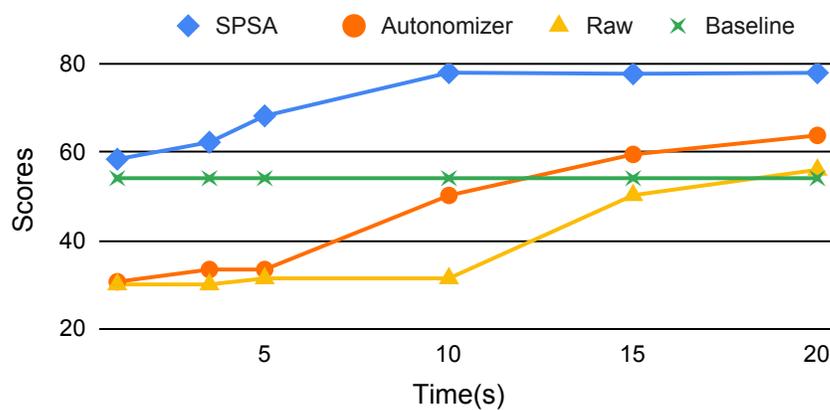


Fig. 5.11.: Watershed tuning score variation

connected neural network, except for the input layer. In particular, for input layer, Raw has 125,000 neurons, *Autonomizer* has 125,001 neurons, and *SPSA* has 1,502 neurons, as shown in Table 5.1. The input neurons vary in size because different settings select different set of program variables as model input features.

In the experiment, we take 10 datasets and apply three settings, *Raw*, *Autonomizer* and *SPSA* to these datasets. We use SSIM score to measure the quality of output segmentation. Higher scores indicate better segmentation results.

Figure 5.10 demonstrates SSIM scores of three methods after convergence. In order to decrease potential influence of randomness, we ran each dataset for 10 times. On average, *SPSA* enhances the score by 25% compared with *Baseline* while *Autonomizer* and *Raw* improved the *Baseline* by 16% and 6% respectively.

Figure 5.11 illustrates the score variation of the three settings within the first 35 seconds of training time. Observe that *SPSA* consistently achieves the highest score among all three settings and it converges within the first 10 seconds. Although *Autonomizer* has a sharp increase after 5 seconds training, it still needs more time to converge and reach the optimal score. For *Raw*, it has to be trained for longer time to converge. The reason is that the input neurons for *Raw* is really huge and the neural network needs more time to extract important features from the raw input.

## Sphinx

**Sphinx** [41] is a very popular speech recognition application. It takes a raw audio and a dictionary, and generates the script for the audio according to its dictionary. Sphinx has several tunable parameters, such as the upper and lower edges of filters, language weight, and word insertion penalty. These parameters are critical to the recognition results. Different audios of different people may require different parameter configurations. Fig. 5.14 gives the pseudo code of Sphinx. The decoder is first initialized with parameters at line 3, then the decoding starts at line 22 using the initialized parameters. Finally, the decoded script can be retrieved at line 25. In this study, we focus on predicting parameters **upperf** and **lowerf** and we leave other parameters with their default values. As described on the official website, **upperf** and **lowerf** are critical to the speech recognition accuracy. They represent the cutoff

frequency of the audio, which means that any frequency that does not fall within that range will not be processed.

**Feature Variables.** We created three different model settings: *Raw*, *Autonomizer*, and *SPSA* to predict the target variables with different feature variables. Specifically, *Raw* is trained with raw audio and *Autonomizer* identifies Fourier transform information that stored in program variable `fft` as the feature variable to train its model. Intuitively, the audio is being transferred into an alternate representation by FFT, i.e., from time domain to frequency domain. Like histogram in image processing, different audios have different frequency information and hence serve as important features. Finally, for model training, *SPSA* selects the reduced feature variable `fft_avg` and `fft_var` (i.e., data reduction by variance) to predict target variable `lowerf` and `upperf` respectively. (Fig. 5.14, lines 14-15).

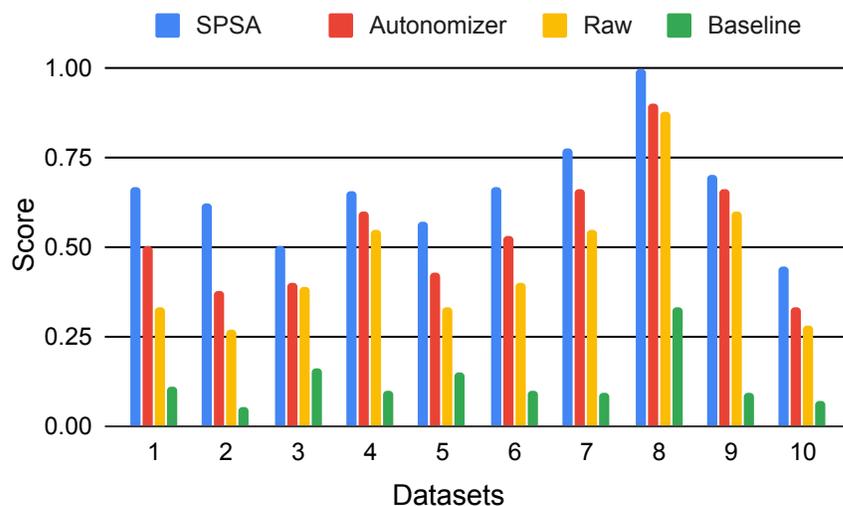


Fig. 5.12.: Sphinx tuning score variation

**Results.** We show the results of baseline, *Raw*, *Autonomizer*, and *SPSA* for comparison. The structure of all models are the same except their input layers. In particular, we use a fully connected neural network with three inner layers for each model. All neurons use the RELU function for activation. The structure is inspired

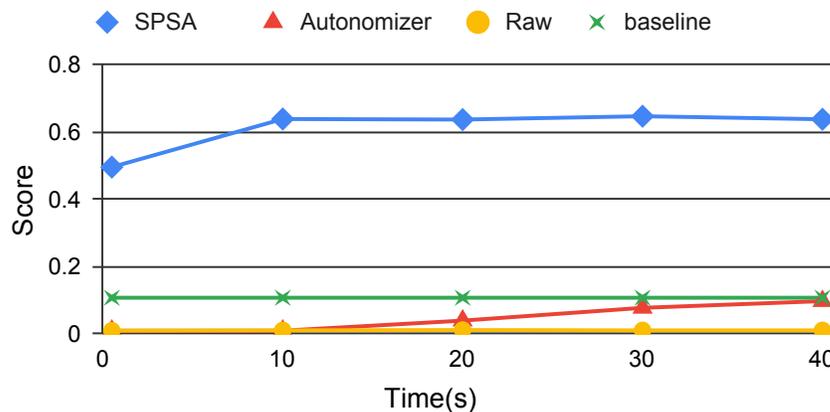


Fig. 5.13.: Sphinx tuning score variation

by [122]. *Raw* and *Autonomizer* has 262,144 neurons and 4,096 neurons in the input layer respectively whereas *SPSA* has 2 neurons.

We use the default training and testing audios from the AN4 dataset [55, 123]. The testing set is split into 10 datasets. Each testing audio comes with a ground truth script from the dataset. For each audio, we use the recognition accuracy to compare the prediction result with the ground truth. The higher the score the better the result.

Fig. 5.12 shows the prediction score after convergence. Observe that *SPSA* almost always produces the best results. On average, *Raw*, *Autonomizer*, and *SPSA* have 425%, 480%, and 497% improvement over the baseline respectively.

We also show the score variation with the training time in Fig. 5.13 for the 10 datasets. Observe that *SPSA* consistently has much better prediction result. Furthermore, even 1-2 seconds training in *SPSA* yields much better results for 40 seconds training in *Raw* and *Autonomizer*. The reason is that both models of *Raw* and *Autonomizer* require a large number of input neurons, which substantially increase the training time.

```

1 void sphinx(audio, upperf, lowerf, ...) {
2 // 1. Initialize decoder with parameters
3 ... initialization ...
4
5 // 2. Smooth audio
6 sAudio = smooth(audio);
7
8 // 3. FFT
9 fft = doFFT(audio);
10 au_extract("FFT", 4096, fft);
11
12 au_reduce("VEC_FFT", "FFT");
13 au_stats("UPPER", "LOWER");
14 au_opt("OPT_UPPER", "UPPER");
15 au_opt("OPT_LOWER", "LOWER");
16
17 au_NN("NN", "FFT", "UPF, LOF");
18 au_write_back("UPF", 1, &upperf);
19 au_write_back("LOF", 1, &lowerf);
20
21 // 2. Decode
22 script = decode(audio);
23
24 // 3. Output result
25 output(script);
26 }

```

Fig. 5.14.: Improving Sphinx autonomization. The highlighted statements are added. Autonomizing primitives start with au.

## 5.5 Summary

In this chapter, we propose SPSA, a novel approach to further improve the software autonomization for even better data processing programs output quality. It provides a set of extension primitives that allow users to find better or more relevant program feature variables for target variable prediction by taking advantage of program analysis and statistical analysis. Our experiments show that SPSA substantially improves data processing results and outperforms the state-of-the-art autonomization framework, AUTONOMIZER.

## 6. RELATED WORK

### 6.1 Floating Point Instability

The floating point instability problem of data processing programs has been studied for a long time. RAIVE is related to dynamic instability detection techniques such as interval analysis [4, 124], high precision computation [10], and error tagging [11]. Compared to these techniques, RAIVE is much more efficient and can reason about output variations.

A dynamic technique was proposed in [20] to detect bit cancellations. It does not distinguish benign and problematic cancellations and thus reports many false alarms. Researchers have proposed techniques to generate tighter bounds for interval arithmetic [22, 125] and affine arithmetic tools [7, 8]. Affine arithmetic handles variable correlations using affine forms. These techniques are very expensive (3-4 orders of magnitude slowdown [8]) and may have difficulty scaling to complex programs. They mostly focus on numerical cores and can hardly reason about discrete differences caused by errors, which are common in real world programs and usually induce substantial output variations.

There are also a large body of work on abstract interpretation, SMT solving, model checking and code perturbation to tackle the internal error problem [1, 3, 126, 127]. Robustness analysis [128] tries to statically prove that a floating point program is free from instability problems. While it is quite successful in handling simple programs, the mathematical complexity and the iterative nature of many real world programs are difficult to address by the technique. Moreover, as instability problems are input dependent and rarely happen, dynamic analysis may be more preferable when completely fixing instability is difficult.

RAIVE is also related to uncertain data processing. In [129], a static analysis is proposed to analyze probabilistic programs that operate on uncertain data. In [130], an abstraction was proposed to help developers operate on and reason about uncertain data. A sampling technique was proposed in [19] to expose discontinuity in output functions, in the presence of input uncertainty. Different from RAIVE, they explicitly model and sample external errors.

In [131], a technique is proposed to search for error-causing inputs that can maximize result errors due to internal errors. In [132], researchers propose to reason about the portability of numerical programs by using symbolic analysis to find inputs that cause different branch decisions, when the program is executed with the same input on different platforms. Recently, [133, 134] propose techniques to reason about the required precision to compile a given program with given output requirements.

## 6.2 Program Parameter Configuration

**Program Tuning.** Many works have been proposed to solve the program tuning problem. WBTuner is related to existing input selection or fuzzing works [135–142]. They use different search techniques such as MCMC or genetic algorithms to address software engineering or cyber-security problems.

Several autotuning frameworks are proposed for domain-specific programs. For example, [143, 144] tune data-mining algorithms; [145] aimed to generate an optimized matrix multiply routine by empirical autotuning; [146] is specialized for tuning stencil computation; and [147] is a stochastic approach for parameter tuning of SVM. In [148], a compiler autotuning framework is proposed to speed up application performance using bayesian networks. Several dynamic autotuning frameworks [149–156] were proposed to monitor program execution to guide the program to perform self-adaptation for achieving specific optimization goal. For example, PowerDial [149] transforms static configuration parameters into dynamic controllable variables to make programs power-aware.

PetaBricks [157, 158] proposes a language- and compiler-based solution for tunable algorithm construction. Different algorithms and parameter configurations are being tuned to achieve better performance and accuracy. Different algorithms are selected for execution by the Petabricks runtime. It advocates the concept of tuning by construction, targeting on stream data processing. The individual streaming components only interact through their interfaces and do not have any other inter-dependences. However, it cannot tune pre-existing non-streaming programs where inter-dependences across phases are substantial like in Ardupilot. Furthermore, users need to use the proposed language.

**Program Autonomization.** Many works were proposed for solving software engineering problems with machine/deep learning techniques, such as test generation [159–162], fuzzing [163–165], and bug repair [166–168]. AUTONOMIZER has the potential of allowing some learning tasks to piggyback on software operation.

Machine learning and deep learning techniques have a wide range of applications such as computer vision [169–171], speech recognition [172, 173], and bioinformatics [174, 175].

For individual application, the developers have to compose the learning procedures from scratch and use raw data. In fact, many of these problems have been extensively studied and they have existing solutions based on programs. The problem is that these programs are often heavily parameterized and require human interventions. AUTONOMIZER is the first work to provide a general technique to autonomize such programs.

Some frameworks have been proposed to train models for playing games such as OpenAI Gym [81], Arcade Learning Environment [176], and Mario AI competition [177]. However, these frameworks are limited to specific platforms and the training is a stand-alone process isolated from the original system operation. In contrast, AUTONOMIZER is general and the training is piggybacking on software operation.

In [178], researchers propose a technique to play NES games automatically. Unlike AUTONOMIZER that works on source, it works on executable. It first identifies the

locations of simulated NES memory that stores the progress (i.e., score or game level) of the NES game. Then the objective function is derived and learned accordingly in order to make progress. Unlike AUTONOMIZER that aims to support various software systems, the work is specific to playing NES games.

While there are some works that leverage internal data as model input features (e.g., in training AIs to play Mario [179] and a first-person shooting game [180]), they are application specific. AUTONOMIZER proposes two general heuristic approaches for programs using supervised and reinforcement learning in order to extract program variables that correspond to important features for model training.

Furthermore, SPSA proposes to combine both program analysis and statistical analysis. Similarly, many works also leverage statistical analysis to solve software engineering problems, such as bug finding [181] and error ranking [182, 183]. Particularly, [183] employs a statistical model to rank error messages in order to lower false positive rates. These works witness how statistical analysis enhances the power of program analysis.

Additionally, deep learning/machine learning techniques are incorporated into traditional software and system fields, such as test generation [162, 184–190] and fuzzing [191–194]. SPSA may be potentially capable of generating optimal test cases and seeking out program patterns based on observations of internal states. [195] replaces traditional index structures with deep-learning models and achieves better indexing performance.

## 7. CONCLUSION

In this dissertation, we propose several techniques, enabled by runtime program analysis and parameter exploration, to provide better data processing results.

Runtime program analysis provides a simple and effective means of studying the uncertainty caused by internal and external errors. We propose RAIVE, a technique to leverage dynamic program analysis through program vectorization. Every floating point value is transformed to a vector of multiple values—the values added to create the vector are obtained by introducing artificial errors that are upper bounds of actual errors. The propagation of artificial errors models the propagation of actual errors. When values in vectors result in discrete execution differences (e.g., following different paths), the execution is forked to capture the resulting output variations. Our evaluation shows that RAIVE can precisely capture output variations. Its overhead (340%) is 2.43 times lower than the state of the art.

In addition to the techniques that handle the program instability problem, we also present several techniques to improve the quality of the results of data processing programs. We first propose WBTUNER, a white-box tuning technique that is implemented as a library. It allows user to compose complex program tuning tasks by adding a small number of library calls to the original program and providing a few callback functions. The comparison with the state of the art OPENTUNER shows that OPENTUNER takes 3.08X time to achieve the same results under a single core environment and 4.67X when multiple cores are used. To prevent per-input tuning, we propose a general framework AUTONOMIZER to autonomize software systems by installing the AI into the traditional programs. With the support of AUTONOMIZER, the users can gain the AI support with little engineering efforts. Our experiment results on nine real-world applications show that the autonomization only requires adding a few lines to the source code. Besides, for the data-processing programs,

AUTONOMIZER improves the output quality by 161% on average over the default settings. For the interactive programs such as game/driving, AUTONOMIZER achieves higher success rate with lower training time than existing autonomized programs. Finally, we propose SPSA, a novel approach that combines both program analysis and statistical analysis for much better feature variables identification. The evaluation shows that SPSA substantially improves data processing results by improving the feature variables selection on ten widely used parameterized programs. The output quality is improved by 99.04% on average over the baseline with execution overhead almost the same as AUTONOMIZER. Comparatively, AUTONOMIZER only improved the output quality by 80.24% on average over the baseline. Furthermore, the model training overhead of SPSA is 27.44X lower than AUTONOMIZER and its model size is 603.5X smaller than AUTONOMIZER on average.

## REFERENCES

## REFERENCES

- [1] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The ASTRÉE Analyzer,” in *ESOP '05*. Springer, 2005.
- [2] E. Goubault and S. Putot, “Static analysis of finite precision computations,” in *VMCAI '11*. Springer-Verlag, 2011.
- [3] M. Martel, “Propagation of roundoff errors in finite precision computations: A semantics approach,” in *ESOP '02*. Springer-Verlag, 2002.
- [4] R. E. Moore, *Interval analysis*, ser. Prentice-Hall series in automatic computation. Prentice-Hall, 1966.
- [5] G. Melquiond and C. Munoz, “Guaranteed proofs using interval arithmetic,” in *ARITH '05*. IEEE Computer Society, 2005.
- [6] L. H. de Figueiredo and J. Stolfi, “Affine Arithmetic: Concepts and Applications,” *Numerical Algorithms*, vol. 37, 2004.
- [7] C. F. Fang, T. Chen, and R. A. Rutenbar, “Floating-point error analysis based on affine arithmetic,” in *ICASSP '03*, 2003.
- [8] E. Darulova and V. Kuncak, “Trustworthy numerical computation in scala,” in *OOPSLA '11*, 2011.
- [9] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker, “Fpinst: Floating point error analysis using dyninst,” 2008.
- [10] F. Benz, A. Hildebrandt, and S. Hack, “A dynamic program analysis to find floating-point accuracy problems,” in *PLDI '12*, 2012.
- [11] T. Bao and X. Zhang, “On-the-fly detection of instability problems in floating-point program execution,” in *OOPSLA '13*, 2013.
- [12] J. Ansel, S. Kamil, K. Veeramachaneni, U.-M. O'Reilly, and S. Amarasinghe, “Opentuner: An extensible framework for program autotuning,” in *PACT '14*, 2014.
- [13] J. Kiefer and J. Wolfowitz, “Stochastic estimation of the maximum of a regression function,” *The Annals of Mathematical Statistics*, 1952.
- [14] J. C. Spall, “Multivariate stochastic approximation using a simultaneous perturbation gradient approximation,” *IEEE Transactions on Automatic Control*, 1992.
- [15] P. Merz and B. Freisleben, “A genetic local search approach to the quadratic assignment problem,” in *ICGA '97*, 1997.

- [16] M. de Hoon, “Cluster 3.0,” <http://bonsai.hgc.jp/~mdehoon/software/cluster/software.htm>.
- [17] R. M. L. Page, S. Brin and T. Winograd., “The pagerank citation ranking: Bringing order to the web.” 1999.
- [18] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, 2008.
- [19] T. Bao, Y. Zheng, and X. Zhang, “White box sampling in uncertain data processing enabled by program analysis,” in *OOPSLA '12*, 2012.
- [20] M. O. Lam, J. K. Hollingsworth, and G. Stewart, “Dynamic floating-point cancellation detection,” *Parallel Computing*, vol. 39, 2013.
- [21] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2013.
- [22] F. D. Dinechin and L. P. Arnaire, “Assisted verification of elementary functions using gappa,” in *SAC '06*, 2006.
- [23] F. Yuan, Z.-H. Meng, H.-X. Zhangz, and C.-R. Dong, “A new algorithm to get the initial centroids,” in *Proceedings of the 3rd International Conference on Machine Learning and Cybernetics*, 2004.
- [24] M. Yedla, S. Pathakota, and T. M. Srinivasa, “Enhancing k-means clustering algorithm with improved initial center,” *IJCIT '10*, 2010.
- [25] K. A. A. Nazeer, S. D. M. Kumar, and M. P. Sebastian, “Enhancing the k-means clustering algorithm by using a  $o(n \log n)$  heuristic method for finding better initial centroids,” in *EAIT '13*, 2013.
- [26] K. A. A. Nazeer and M. P. Sebastian, “Improving the accuracy and efficiency of the k-means clustering algorithm,” in *WCE '09*, 2009.
- [27] A. M. Fahim, A. M. Salem, F. A. Torkey, and M. A. Ramadan, “An efficient enhanced k-means clustering algorithm,” *Journal of Zhejiang University SCIENCE A*, 2006.
- [28] T. Blaschke, “Object based image analysis for remote sensing,” *ISPRS Journal of Photogrammetry and Remote Sensing*, 2010.
- [29] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.
- [30] M. D. Heath, S. Sarkar, T. Sanocki, and K. W. Bowyer, “Robust visual method for assessing the relative performance of edge-detection algorithms,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.
- [31] WBTuner, “Wbtuner source and supplementary material,” ”<https://github.com/cgo2019/WBTuner>”, 2018.
- [32] F. Kerouh, “A no-reference blur image quality measure based on wavelet transform,” *IJDIWC '12*, 2012.

- [33] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Transactions on Image Processing*, 2004.
- [34] A. Moore, "Cross-validation for detecting and preventing overfitting," 2001.
- [35] F. Kane, "Hands-on data science and python machine learning," 2017.
- [36] M. Stone, "Cross-validators: choice and assessment of statistical predictions," *Journal of the Royal Statistical Society. Series B (Methodological)*, 1974.
- [37] E. backoff, IEEE Standard 802.3-2008, 2008.
- [38] D. Bloomberg, "Leptonica image processing and analysis library," "<http://www.leptonica.com/>", 2001.
- [39] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD '96*, 1996.
- [40] D. S. Bolme, J. R. Beveridge, M. Teixeira, and B. A. Draper, "The csu face identification evaluation system: its purpose, features, and structure," in *Computer Vision Systems*, 2003.
- [41] P. Lamere, P. Kwok, E. Gouvea, B. Raj, R. Singh, W. Walker, M. Warmuth, and P. Wolf, "The CMU sphinx-4 speech recognition system," in *ICASSP '03*, 2003.
- [42] D. Plotree and D. Plotgram, "Phylip-phylogeny inference package (version 3.2)," *Cladistics*, 1989.
- [43] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *PNAS '98*, 1998.
- [44] X. Ning and G. Karypis, "Slim: Sparse linear methods for top-n recommender systems," in *ICDM 2011*, 2011.
- [45] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.
- [46] J. R. Quinlan, *C4. 5: programs for machine learning*, 1993.
- [47] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, 1995.
- [48] R. Mackay, "Ardupilot," "<http://ardupilot.org/>", 2007.
- [49] Á. Fialho, L. Da Costa, M. Schoenauer, and M. Sebag, "Analyzing bandit-based adaptive operator selection mechanisms," *Annals of Mathematics and Artificial Intelligence*, 2010.
- [50] N. Friedman, M. Ninio, I. Pe'er, and T. Pupko, "A structural em algorithm for phylogenetic inference," *Journal of Computational Biology*, 2002.
- [51] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *IISWC '06*, 2006.

- [52] T. Joachims, “Making large-scale SVM learning practical,” *Advances in Kernel Methods - Support Vector Learning*, 1999.
- [53] A. Asuncion and D. Newman, “Uci machine learning repository,” 2007.
- [54] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, “The alpbench benchmark suite for complex multimedia applications,” in *IISWC 2016*.
- [55] R. Reddy, “An4 database,” ”<http://www.speech.cs.cmu.edu/databases/an4/>”, 1991.
- [56] L. Meier, “Px4 autopilot,” ”<http://px4.io/>”, 2009.
- [57] P. Tuning, ”[https://docs.px4.io/en/advanced\\_config/pid\\_tuning\\_guide\\_multicopter.html](https://docs.px4.io/en/advanced_config/pid_tuning_guide_multicopter.html)”, 2017.
- [58] A. Tuning, ”<http://ardupilot.org/copter/docs/tuning.html>”, 2017.
- [59] O. Andersson, M. Wzorek, and P. Doherty, “Deep learning quadcopter control via risk-aware active learning,” in *AAAI ’17*, 2017.
- [60] I. Mordatch, K. Lowrey, G. Andrew, Z. Popovic, and E. V. Todorov, “Interactive control of diverse complex characters with neural networks,” *Advances in Neural Information Processing Systems*, 2015.
- [61] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, “Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search,” in *ICRA ’16*, 2016.
- [62] N. Koenig and A. Howard, “Gazebo,” ”<http://gazebo.org/>”, 2009.
- [63] Adrupilot-default, ”<https://drive.google.com/open?id=0BxgPTM7nEUyCcTFKdjM4RVNrMk0>”, 2018.
- [64] Adrupilot-tuned, ”<https://drive.google.com/open?id=0BxgPTM7nEUyCYmVPdzBtMnoyT1U>”, 2018.
- [65] PX4, ”<https://drive.google.com/open?id=0BxgPTM7nEUyCYnRxS2FSN2JRbEE>”, 2018.
- [66] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [67] (2018) Alphago versus lee sedol. [Online]. Available: [https://en.wikipedia.org/wiki/AlphaGo\\_versus\\_Lee\\_Sedol](https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol)
- [68] (2018) Googles alphago defeats chinese go master in win for a.i. [Online]. Available: <https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html>
- [69] (2018) Alphago at the future of go summit,. [Online]. Available: <https://deepmind.com/research/alphago/alphago-china/>
- [70] (2018) Waymo. [Online]. Available: <https://waymo.com/>

- [71] (2018) Waymo. [Online]. Available: <https://www.cnbc.com/2018/08/29/waymo-alphabets-self-driving-unit-morgan-stanley-forecast-to-highs.html>
- [72] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” 2017. [Online]. Available: <https://arxiv.org/abs/1611.01578>
- [73] (2018) Ui/application exerciser monkey. [Online]. Available: <https://developer.android.com/studio/test/monkey>
- [74] S. Bling. (2018) Mario bizhawk emulator. [Online]. Available: <https://pastebin.com/u/SethBling>
- [75] (2018) Deepmind ai reduces google data centre cooling bill by 40. [Online]. Available: <https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/>
- [76] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, 2015.
- [77] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, “Testar: Tool support for test automation at the user interface level,” *Int. J. Inf. Syst. Model. Des.*, vol. 6, no. 3, pp. 46–83, Jul. 2015. [Online]. Available: <http://dx.doi.org/10.4018/IJISMD.2015070103>
- [78] A. Jung. (2018) mario-ai: Playing mario with deep reinforcement learning. [Online]. Available: <https://github.com/aleju/mario-ai>
- [79] (2017) Torcs for reinforcement learning. [Online]. Available: <https://github.com/YurongYou/rlTORCS>
- [80] (2016) Using keras and deep deterministic policy gradient to play torcs. [Online]. Available: <https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html>
- [81] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” 2016.
- [82] (2018) Skydio. [Online]. Available: <https://www.skydio.com/>
- [83] O. M. Parkhi, A. Vedaldi, and A. Zisserman, “Deep face recognition,” in *BMVC '15*, 2015.
- [84] A. Irpan. (2018) Deep reinforcement learning doesn’t work yet. [Online]. Available: <https://www.alexirpan.com/2018/02/14/rl-hard.html>
- [85] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” *CoRR*, 2017.
- [86] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

- [87] L. Jakowski. (2018) jakowskidev/umario\_jakowski: umario c++/sdl2 game by lukasz jakowski. [Online]. Available: [https://github.com/jakowskidev/uMario\\_Jakowski](https://github.com/jakowskidev/uMario_Jakowski)
- [88] Autnomizer. (2018) Autnomizer. [Online]. Available: <https://github.com/ProjectDemooo/Autnomizer>
- [89] (2018) Using deep q-network to learn how to play flappy bird. [Online]. Available: <https://github.com/yenchenlin/DeepLearningFlappyBird>
- [90] d. t. GNU. (2018) Using and porting the gnu compiler collection (gcc): Gcov. [Online]. Available: [http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc\\_8.html](http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html)
- [91] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.
- [92] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, 2014.
- [93] R. Hecht-Nielsen, “Neural networks for perception,” 1992, ch. Theory of the Backpropagation Neural Network.
- [94] (2018) sklearn scale. [Online]. Available: [https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.minmax\\_scale.html](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.minmax_scale.html)
- [95] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *OSDI '16*, 2016.
- [96] Python2.7. (2018) Extending python with c or c++. [Online]. Available: <https://docs.python.org/2/extending/extending.html>
- [97] d. t. CRIU. (2018) Criu. [Online]. Available: [https://criu.org/Main\\_Page](https://criu.org/Main_Page)
- [98] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.
- [99] I. Habib, “Virtualization with kvm,” *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.
- [100] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *PLDI 2007*, 2007.
- [101] W. M. Khoo. (2013) wmkhoo/taintgrind - github. [Online]. Available: <https://github.com/wmkhoo/taintgrind>
- [102] M. Heath, S. Sarkar, T. Sanocki, and K. Bowyer, “Comparison of edge detectors: a methodology and initial study,” *Computer vision and image understanding*, vol. 69, no. 1, pp. 38–54, 1998.
- [103] D. stutz, “Introduction to neural networks,” *Selected Topics in Human Language Technology and Pattern Recognition*, 2014.

- [104] P. Arbelaez, M. Maire, C. Fowlkes, and J. Malik, "Contour detection and hierarchical image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 5, pp. 898–916, 2011.
- [105] (2016) Torcs, the open racing car simulator. [Online]. Available: <http://torcs.sourceforge.net/>
- [106] V. Mnih, A. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, 2016.
- [107] Y. You, X. Pan, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," *CoRR*, 2017.
- [108] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *ICCV '15*, 2015.
- [109] M. C. Frith, M. Hamada, and P. Horton, "Parameters for accurate genome alignment," *BMC bioinformatics*, vol. 11, no. 1, p. 80, 2010.
- [110] A. Biernacki and K. Tutschku, "Performance of http video streaming under different network conditions," *Multimedia Tools and Applications*, vol. 72, no. 2, pp. 1143–1166, 2014.
- [111] A. Outchakoucht, E.-S. Hamza, and J. P. Leroy, "Dynamic access control policy based on blockchain and machine learning for the internet of things," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 7, pp. 417–424, 2017.
- [112] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI '08*, 2008.
- [113] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.
- [114] G. J. Szekely, M. L. Rizzo, and N. K. Bakirov, "Measuring and testing dependence by correlation of distances," *The Annals of Statistics*, 2007.
- [115] "Thirteen ways to look at the correlation coefficient," *The American Statistician*, 1988.
- [116] C. A. Rothwell, J. L. Mundy, W. Hoffman, and V. D. Nguyen, "Driving vision by topology," in *ISCV '95*, 1995.
- [117] I. Sobel, "An isotropic 3x3 image gradient operator," *Presentation at Stanford A.I. Project 1968*, 02 2014.
- [118] R. Barnes, C. Lehman, and D. Mulla, "Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models," 2015.
- [119] (2001) Dan bloomberg. leptonica image processing and analysis library. [Online]. Available: <http://www.leptonica.org/color-segmentation.html>

- [120] M. Tepper, M. Mejail, P. Muse, and A. Almansa, “Boruvka meets nearest neighbors,” in *CIARP '13*, 2011.
- [121] D. Bolme, J. Beveridge, M. Teixeira, and B. Draper, “The csu face identification evaluation system: Its purpose, features, and structure,” in *Machine Vision and Applications '03*, 2003.
- [122] Z. Kons and O. Toledo-Ronen, “Audio event classification using deep neural networks,” in *INTERSPEECH '13*, 2013.
- [123] A. Acero, “Acoustical and environmental robustness in automatic speech recognition,” in *ICASSP '90*, 1990.
- [124] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, *Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag New York Incorporated, 2012.
- [125] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védryne, “Towards an industrial use of fluctuat on safety-critical avionics software,” in *FMICS '09*, 2009.
- [126] E. Tang, E. Barr, X. Li, and Z. Su, “Perturbing numerical calculations for statistical analysis of floating-point program (in)stability,” in *ISSTA '10*, 2010.
- [127] D. Monniaux, “The pitfalls of verifying floating-point computations,” *TOPLAS '08*, vol. 30, 2008.
- [128] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, “Proving programs robust,” in *ESEC/FSE '11*, 2011.
- [129] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, “Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths,” in *PLDI '13*, 2013.
- [130] J. Bornholt, T. Mytkowicz, and K. S. McKinley, “Uncertain t: A first-order type for uncertain data,” in *ASPLOS '14*, 2014.
- [131] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, “Efficient search for inputs causing high floating-point errors,” in *PPoPP '14*, 2014.
- [132] Y. Gu, T. Wahl, M. Bayati, and M. Leeser, “Behavioral non-portability in scientific numeric computing,” in *Euro-Par '15*, 2015.
- [133] E. Darulova and V. Kuncak, “Sound compilation of reals,” in *POPL '14*, 2014.
- [134] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, “Precimonious: Tuning assistant for floating-point precision,” in *SC '13*, 2013.
- [135] M. Carbin and M. C. Rinard, “Automatically identifying critical input regions and code in applications,” in *ISSTA '10*, 2010.
- [136] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, “Automatic input rectification,” in *ICSE '12*, 2012.

- [137] M. Carbin, S. Misailovic, and M. C. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *OOPSLA '13*, 2013.
- [138] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, 1990.
- [139] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *ICSE '07*, 2007.
- [140] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *ICSE '09*, 2009.
- [141] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O’Reilly, and S. Amarasinghe, “Autotuning algorithmic choice for input sensitivity,” in *PLDI '15*, 2015.
- [142] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, “Generating test cases for specification mining,” in *ISSTA '10*, 2010.
- [143] O. Burmeister, M. Reischl, G. Bretthauer, and R. Mikut, “Data mining analyses with the matlab toolbox gait-cad,” *Automatisierungstechnik*, 2008.
- [144] J. Besson, C. Rigotti, I. Mitasiunaite, and J.-F. Boulicaut, “Parameter tuning for differential mining of string patterns,” in *ICDMW '08*, 2008.
- [145] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *SC '98*, 1998.
- [146] M. Christen, O. Schenk, and H. Burkhart, “Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures,” in *IPDPS '11*, 2011.
- [147] F. Imbault and K. Lebart, “A stochastic optimization approach for parameter tuning of support vector machines,” in *ICPR 2004*, 2004.
- [148] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, “Cobayn: Compiler autotuning framework using bayesian networks,” *ACM Transactions on Architecture and Code Optimization*, 2016.
- [149] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, “Dynamic knobs for responsive power-aware computing,” *ACM SIGPLAN Notices*, 2011.
- [150] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann, “Using code perforation to improve performance, reduce energy consumption, and respond to failures,” MIT, Tech. Rep., 2009.
- [151] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Transactions on Autonomous and Adaptive Systems*, 2009.
- [152] V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed, “Enabling self-managing applications using model-based online control strategies,” in *ICAC 2006*, 2006.
- [153] F. Chang and V. Karamcheti, “A framework for automatic adaptation of tunable distributed applications,” *Cluster Computing, 2011*, 2011.

- [154] W. Baek and T. M. Chilimbi, “Green: A framework for supporting energy-conscious programming using controlled approximation,” *SIGPLAN Not.*, 2010.
- [155] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O’Reilly, and S. Amarasinghe, “Siblingrivalry: Online autotuning through local competitions,” in *CASES 2012*, 2012.
- [156] M. F. Ringenburt, A. Sampson, L. Ceze, and D. Grossman, “Profiling and autotuning for energy-aware approximate programming,” in *WACAS 2014*, 2014.
- [157] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, “Petabricks: A language and compiler for algorithmic choice,” in *PLDI 2009*, 2009.
- [158] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, “Language and compiler support for auto-tuning variable-accuracy algorithms,” in *CGO 2011*, 2011.
- [159] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, “Automatic text input generation for mobile testing,” in *ICSE ’17*, 2017, pp. 643–653.
- [160] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” in *ACM Sigplan Notices*, vol. 48, no. 10, 2013, pp. 623–640.
- [161] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, “Learning combinatorial interaction test generation strategies using hyperheuristic search,” in *ICSE ’15*, 2015.
- [162] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing advanced driver assistance systems using multi-objective search and neural networks,” in *ASE ’16*, 2016, pp. 63–74.
- [163] P. Godefroid, H. Peleg, and R. Singh, “Learn&fuzz: Machine learning for input fuzzing,” in *ASE ’17*, 2017, pp. 50–59.
- [164] O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” in *PLDI ’17*, 2017, pp. 95–110.
- [165] K. Böttinger, P. Godefroid, and R. Singh, “Deep reinforcement fuzzing,” *CoRR*, 2018.
- [166] R. Gupta, S. Pal, A. Kanade, and S. Shevade, “Deepfix: Fixing common c language errors by deep learning.” in *AAAI ’17*, 2017, pp. 1345–1351.
- [167] K. Wang, R. Singh, and Z. Su, “Dynamic neural program embedding for program repair,” *CoRR*, 2017.
- [168] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Combining deep learning with information retrieval to localize buggy files for bug reports,” in *ASE’15*, 2015, pp. 476–481.
- [169] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS ’12*, 2012.

- [170] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. Lecun, “Pedestrian detection with unsupervised multi-stage feature learning,” in *CVPR '13*, 2013.
- [171] W. Shen, X. Wang, Y. Wang, X. Bai, and Z. Zhang, “Deepcontour: A deep convolutional feature learned by positive-sharing loss for contour detection,” in *CVPR '15*, 2015.
- [172] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Transactions on , Speech, and Language Processing*, vol. 20, no. 1, pp. 30–42, 2012.
- [173] A. Graves, A. r. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *ICASSP '13*, 2013.
- [174] Y. Wang, H. Mao, and Z. Yi, “Protein secondary structure prediction by using deep learning method,” *Knowledge-Based Systems*, vol. 118, pp. 115 – 123, 2017.
- [175] Y. Chen, Y. Li, R. Narayan, A. Subramanian, and X. Xie, “Gene expression inference with deep learning,” *Bioinformatics*, vol. 32, no. 12, pp. 1832–1839, 2016.
- [176] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.
- [177] (2012) Mario ai championship 2012. [Online]. Available: <http://www.marioai.org/>
- [178] D. Tom and M. V. P. D, “The first level of super mario bros. is easy with lexicographic orderings and time travel... after that it gets a little tricky.” 2013.
- [179] Y. Liao, K. Yi, and Y. Zhe, “Cs229 final report reinforcement learning to play mario,” 2012.
- [180] G. Lample and D. S. Chaplot, “Playing FPS games with deep reinforcement learning,” *CoRR*, 2016.
- [181] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, “From uncertainty to belief: Inferring the specification within.” in *OSDI '07*, 2006.
- [182] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, “Correlation exploitation in error ranking,” in *SIGSOFT '04*, 2004.
- [183] T. Kremenek and D. Engler, “Z-ranking: Using statistical analysis to counter the impact of static analysis approximations,” in *ISAS '03*, 2003.
- [184] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, “Learning combinatorial interaction test generation strategies using hyperheuristic search,” in *ICSE '15*, 2015.
- [185] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” in *OOPSLA '13*, 2013.
- [186] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, “Automatic text input generation for mobile testing,” in *ICSE '17*, 2017.

- [187] M. Carbin and M. Rinard, “Automatically identifying critical input regions and code in applications,” 2010.
- [188] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, “Generating test cases for specification mining,” in *ISSTA '10*, 2010.
- [189] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, “Automatic input rectification,” in *ICSE '12*, 2012.
- [190] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *ICSE '07*, 2007.
- [191] P. Godefroid, H. Peleg, and R. Singh, “Learn&fuzz: Machine learning for input fuzzing,” in *ASE '17*, 2017.
- [192] O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” in *PLDI '17*, 2017.
- [193] K. Böttinger, P. Godefroid, and R. Singh, “Deep reinforcement fuzzing,” *CoRR*, 2018.
- [194] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *ICSE '09*, 2009.
- [195] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, “The case for learned index structures,” in *CoRR*, 2018.