Compiler Auto-Tuning for Code Optimization

by

Sunghyun Park

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor Scott Mahlke, Chair
Associate Professor Ronald Dreslinski Jr
Professor Mingyan Liu
Associate Professor Yongjun Park, Hanyang University
Associate Professor Lingjia Tang

© Sunghyun Park

sunggg@umich.edu

ORCID iD: 0000-0003-4793-9069

2021

To my family

ACKNOWLEDGEMENTS

Overall, it was such a journey. However, this journey would have not been possible without the support of many people.

First of all, I would like to thank Professor Scott Mahlke for giving me this wonderful opportunity to grow and advising me how to become an independent researcher. By using his insights and critical thinking, I could learn important questions that have to be asked and shape my own way of attacking the problem. Also, I truly appreciate how he helped me making important decisions in/outside the research.

It was fortunate to have the privilege of working with amazing people. Professor Yongjun Park has provided invaluable advice and support for my studies. The mentorship from Dr. Youfeng Wu had a great influence on my direction of thesis by assisting me to explore my research interests during my internship at Intel. The collaboration work with folks at Carnegie Mellon University under the guidance of Professor Tianqi Chen and Professor Zhihao Jia has been inspiring and their insights have been helpful to move forward to the right direction. I also enjoyed working as a teaching assistant for Professor Lingjia Tang by having certain degree of freedom in leading every Friday's discussion session. The support and advice from Professor Jae W. Lee was very helpful during the graduate school admission process. Also, Professor Mingyan Liu and Ronald Dreslinski Jr gave me great feedback to improve my dissertation.

I am grateful to have great comrades as a part of the CCCP research group. Ankit Sethia, Mehrzad Samadi, Anoushe Jamshidi, Andrew Lukefahr, Shruti Padmanabha, Hyoun Kyu Cho, John Kloosterman, Jason Jong Kyu Park, and Janghaeng Lee have been

always approachable and willing to help me whenever I need. Also, it has been fun to suffer together and grow together with Babak Zamirai, Jiecao Yu, Shikai Li, Jonathan Bailey, Salar Latifi, Ze Zhang, Hossein Golestani, Pedram Zamirai, Armand Behroozi, Brandon Nguyen, Yunjie Pan and Sanjay Sri Vallabh.

When life gets rough on me, I could use my friends to laugh and get recharged. Hyochan Ahn, Sungwoon Jang, Sangmin Yoo, Heewoo Kim, Gwen Ahn, Sangtaek Oh, Walter Shin, Chihyo Ahn, Kyumin Kwon, Byungjoon Lee, Taeju Park, Junki Cho, Seungjong Lee, Changyoung Jeong, Jonghoon Shin, Hun Kim, Hyorim Han and many others made Ann Arbor feel like home. Jongjin Lee, Chungha Sung and Sungbo Park are the ones who have been going through the graduate school together at different locations in North America. Although he is half a globe away, my old friend, Daseul Shin, has been giving me great encouragement by having our own silly conversations over the phone. Also, whenever I visit Korea once a year, Inhyuk Seo, Yonghan Kim, Junoh Park, Heesung Lee, Hyunkwon Shin, Ji-Su Lee, Seejun Choi, Sookyung Park, Jae-Yun Kim, Sedong Yeo, Minseo Kim, Jaewook Mirco Jung, Dojeon Lee, Junsu Kim, Jaehwan Seol, Sungmin Kim, and Jimo Gu always welcome me and make me feel like old times. During my last semester, I had a unique opportunity to collaborate with my old friend, Byungsoo Jeon. I have been enjoying this special experience very much. Also, I am happy to have my close friend, Yoon Jung, nearby for my new start in Seattle.

I would like to give my special thanks to my dear friend, Youngdae Kim, who always has been funny and inspiring to me. Without him, this journey would not have started. I miss him so much lately.

Last but not least, my utmost gratitude goes to my beloved family. Their infinite love and support made me possible to come this far. Unconditional love from my grandmother always makes me feel warm inside, steadfast belief from my parents gives me confidence to challenge myself to the next step, and my little bother always provides me with an immense amount of gratitude for everything he does for our family. I cannot wait to celebrate the

end of this chapter in my life with them.

TABLE OF CONTENTS

DEDICATION . . . . **ii**

ACKNOWLEDGEMENTS . . . . **iii**

LIST OF FIGURES . . . . **ix**

LIST OF TABLES . . . . **xiii**

ABSTRACT . . . . **xiv**

CHAPTER

1 Introduction **1**

1.1 Introduction 1
1.2 Challenges and Contributions 3
1.2.1 Feasibility Study of Learning-Based Auto-Tuning Methods . . . 3
1.2.2 Customization for a Group of Compiler Optimizations 3
1.2.3 Search for the Best Use of Diverse Backends 4
1.2.4 Provision of Low-Cost Fault Protection 4
1.3 Background 5
1.3.1 Compilers From Diverse Domains and Their Optimizations . . . 5
1.3.2 Compiler Auto-Tuning 6

2 Multi-objective Exploration for Practical Optimization Decisions in Binary Translation **10**

2.1 Introduction 10
2.2 Background 12
2.2.1 Our Infrastructure with Binary Translation 12
2.2.2 Supervised Multi-class Classification 14
2.3 Motivation 16
2.4 Challenges and Opportunities 17
2.5 Data Generation 18
2.5.1 Feature Extraction 19
2.5.2 Optimal Factor Exploration 20
2.6 Evaluation 20
2.6.1 Experimental Setup 20
2.6.2 Prediction Accuracy 22

2.6.3 Performance Improvement In Translated Code 24
2.6.4 Prediction Overhead Analysis 25
2.6.5 Choice of Classification Algorithm 26
2.7 Redundant Feature Pruning 28
2.8 Related Works 30
2.9 Conclusion 31

3 SRTuner: Effective Compiler Optimization Customization By Exposing Synergistic Relations 33

3.1 Introduction 33
3.2 Related Works 35
3.3 Design Motivation 37
3.3.1 Multistage Structure 38
3.3.2 Optimization Impact Estimation 40
3.3.3 Handling Local Optima 42
3.4 SRTuner 43
3.4.1 Reward Policy 46
3.4.2 Stage-Optimization Mapping 47
3.4.3 Extract Important Synergistic Relations 48
3.5 Performance Evaluation 48
3.5.1 Experimental Setup 48
3.5.2 GCC 49
3.5.3 TVM 51
3.5.4 OpenCL Compilers 52
3.6 Discussion 52
3.6.1 Identification Of Important Misuses 52
3.6.2 Generalization Of Tuning Experiences 55
3.7 Conclusion 57

4 Collage: Auto-tuning Deep Learning Execution Plans With Diverse Backends 58

4.1 Introduction 58
4.2 Related Work 62
4.3 Overview 63
4.4 Backend Pattern Abstraction 64
4.5 Auto-tuning Execution Plan 66
4.5.1 Backend Placement Problem 66
4.5.2 Op-level Auto-tuning 66
4.5.3 Graph-level Auto-tuning 69
4.6 Evaluation 70
4.6.1 Experimental Setup 70
4.6.2 End-to-end Evaluation 71
4.6.3 Analysis of Backend Operator Placement 73
4.6.4 Tuning Time 73
4.7 Conclusion 74

5 Low-Cost Prediction-Based Fault Protection Strategy 76

5.1 Introduction 76
5.2 Motivation and Idea 78
5.3 System Overview 82
5.4 Prediction Techniques 83
5.4.1 Dynamic Interpolation 84
5.4.2 Approximate Memoization 85
5.5 Run-time Management 86
5.6 Training 88
5.7 Evaluation 88
5.7.1 Performance Overhead 89
5.7.2 Reliability 93
5.7.3 The Rationality of Acceptable Range 94
5.8 Related Works 95
5.9 Conclusion 96

6 Conclusion and Future Works 97

6.1 Conclusion 97
6.2 Future Works 99

BIBLIOGRAPHY 101

LIST OF FIGURES

1.1 Growth of computer performance using SPECintCPU benchmark suite [78]. The end of Dennard Scaling and Moore's Law slows down the performance improvement with hardware innovations. Annotation (star) indicates where we stand today in 2020. 2
1.2 Representative tuning problems 7
1.3 Representative tuning approaches 8

2.1 Big picture of our HW/SW co-designed CPU with a DBT system. It constains two different microarchitectures that each supports its own ISA. The highlighted box illustrates the translation process by a binary translator and a dynamic optimizer. 13
2.2 Distribution of optimal unroll factors. 16
2.3 Evaluation of current heuristic design. Prediction accuracy for optimal loop unrolling decision and its impact on the dynamic instruction count in the resulting optimized code are measured. 17
2.4 Prediction accuracy with various classes. 23
2.5 The breakdown of wrong predictions. 23
2.6 Averaged instruction count reduction and the ratio of optimized loops compared to *Oracle* for each decision model. 25
2.7 Inference time for each model with given budget. 26
2.8 The change in prediction accuracy and inference time during greedy feature selection process. With Top 15 features (marked with stars), the decision tree model can show an almost identical level of prediction accuracy to the model with all 34 features while satisfying the time constraint. 28

3.1 Motivational experiment with two representative compilers. A reward (+1) is assigned whenever a combination outperforms the baseline setting. By sampling 2,000 settings, averaged reward for configuring each flag is examined. . . 38
3.2 Averaged reward given extra condition: (a)-(c) for GCC, (d) for TVM. Starting from Figure 3.1, the high-impact flag is configured for each benchmark. (a) shows the average reward after a single stage. When more than one flags are given, the relation between optimizations can be unveiled. (b) and (c) present the positive/negative interaction between two given optimizations, respectively. (d) demonstrates the same approach can be applied to TVM as well. 39
3.3 Comparison between prior impact estimation method and our distribution-based method. To approximate an optimization impact, prior approaches ignore inter-optimization relations and measure the performance difference on the single fixed configurations of the rest. However, our method estimates based on the performance distribution to consider interactions with diverse configurations of others. 41
3.4 Performance distributions and their KL divergence (D_{KL}) in *consumer_jpeg_c* given a certain GCC optimization. Relative frequency presents an occurrence normalized by the entire sample counts. (a) suggests to turn off the flag by showing a higher relative frequency of outperforming the baseline (i.e., speedup above 1.0) when disabled. (c) illustrates how a decision of earlier optimization affects latter optimization decision. Left graph presents performance distribution when there is no other optimizations configured and right two graphs display how the performance distribution changes when (a) or (b) are configured, respectively. 42
3.5 SRTuner overview 43
3.6 An example of multistage structure used in SRTuner. Each stage configures an optimization and each node represents a combination forged at earlier stages. For example, the annotated node ($\star$) will configure X_3 when *config1* and *config2* are chosen for X_1 and X_2, respectively. Each node counts the number of visits, n, and accumulates rewards, r. 45
3.7 An example of SRTuner strategy. White nodes represent expanded nodes while black nodes imply topmost nodes in the unknown areas. +1 reward is applied when a trial outperforms the baseline. As tuning proceeds, SRTuner naturally exposes popular paths on the structure, which show the constructive relations between optimization decisions. 45
3.8 Visualization of different reward policies 46
3.9 Analysis of reordering frequencies. Each performance is normalized by the one without remapping. 47
3.10 Tuning C/C++ applications with GCC on Intel Xeon E5-2430. 50
3.11 Tuning deep learning workloads with TVM on NVIDIA RTX 2080Ti. 51
3.12 Tuning deep learning workloads with TVM on NVIDIA Quadro K620. 52
3.13 Tuning *gemm* with OpenCL on Intel i7-9700K. 53
3.14 Tuning *gemm* with OpenCL on NVIDIA RTX 2080Ti. 53

3.15 Examine representative subset of synergistic optimizations revealed by 54
3.16 Hot code region in *telecom_adpcm_d* where if-conversion and phi-node optimizations are seriously misapplied by GCC. 54
3.17 Diverse applications with *-fno-if-conversion* and *-fno-ssa-phiopt* in their synergistic optimizations. Breakdown shows their importance in each customized setting. 55
3.18 Experiment with eight different kernels (*k1-8*) and diverse SM activation ratios. (a) Performance ratio between high-end and low-end settings. Green and red cells imply cases where high-end and low-end settings should be applied respectively. Blue line represents a possible decision boundary. (b) Speed-up when choosing one of high-end and low-end settings in the right places. Speed-up is computed against the performance of default setting with 100% SMs. 56

4.1 Performance of various convolution operators in ResNext-50 on NVIDIA RTX 2070; Note that there is no single backend that always delivers the fastest implementation. 59
4.2 An example of possible execution strategies for a simple workload with four different backends. This illustrates the explosive size of the search space and the difficulty in the efficient exploration. To assign backend, both graph topology and the supported backend pattern should be matched. In case of lowering to fused operators or offloading partial graph to inference engines, matched operators should not have any cyclic data dependency to avoid deadlock. . . . 60
4.3 System overview of *Collage* 63
4.4 Example illustrating how the backend pattern generator would automatically generate valid patterns with the pattern rule assumed in the listing 1. 65
4.5 Example of Dynamic Programming (DP) procedures for pattern matching and optimal cost update 67
4.6 Example of Evolutionary Search (ES) procedure 69
4.7 End-to-end performance of diverse strategies in five different workloads on NVIDIA RTX 2070. Each performance is normalized by the performance of *Collage*. Note that following state-of-the-arts backends are employed for each framework according to their capabilities: PyTorch (cuDNN, cuBLAS), TVM (cuDNN, cuBLAS, AutoTVM) and (cuDNN, cuBLAS, AutoTVM, and TensorRT). 71
4.8 Backend placement discovered by *Collage* on ResNeXt50 and BERT. 72
4.9 Op-level tuning (DP) time breakdown for five different workloads. On average, profiling overhead for operator cost measurements takes up 82% of the entire tuning time. Note that profiling is only necessary for new operators. Once the cost of a new operator is measured, its information will be saved in the logging database in *Collage* to avoid the repetitive profiling. 74
4.10 Performance improvement of graph-level tuning over time for five different workloads. The y-axis presents the speedup relative to op-level tuning (DP). We observe that one hour of graph-level tuning empirically provides good trade-off between tuning time and speedup for most workloads. 74

5.1 Idea of each protection strategy. **(a)** Re-compute and validate. **(b)** Estimate and fuzzy-validate. 79
5.2 Proportion of dynamic instructions whose computation outputs can be estimated. Measured in Rodinia [54] benchmark suite. 80
5.3 System Overview 82
5.4 Sketch of dynamic interpolation. Whenever slope change is above tuning parameter (TP), a phase is defined. During the validation process at (c), a data element is considered as a possible fault if the difference between original value and prediction value is greater than acceptable range (AR). 83
5.5 An example of run-time management for dynamic interpolation. Run-time management periodically generates context signatures by summarizing the current run-time context to adjust TP based on the QoS model. 87
5.6 Test result with test inputs of each benchmarks (AR : Acceptable Range) . . . 89
5.7 Test result with test inputs of each benchmarks (AR : Acceptable Range) . . . 90
5.8 The detailed analysis for two selected benchmarks 91
5.9 The result of the fault injection experiment. Each protection scheme is tagged under every application. 92

LIST OF TABLES

2.1 A subset of features for loop unroll decision under different categories. The ratio of each static operation is computed by dividing the number of each operation by the number of static instructions. Note, unavailable features (e.g., live range size) in our infrastructure are crossed out. The total of 34 features are extracted and used for the experiment. 19
2.2 Configurations for the classifiers. Annotated (*) configurations are generated without any restriction on both the maximum depth and the maximum number of leaf nodes. 22
2.3 Memory requirement for each model. The requirements for decision tree and random forest configurations are represented by their largest configurations. . . 27
2.4 Top 15 features for decision tree. Check marks (✓) indicate that the corresponding features are considered in the current heuristic design: 8 out of the top 15 features are employed by current heuristics. 29

3.1 Experiment setup 49
3.2 GPU specification. We consider RTX 2080 Ti and Quadro K620 as high-end and low-end devices respectively. 55

5.1 Selected benchmarks. The impact of skipping re-computation can be imagined by provided computation type and location. Both training input and test input are randomly generated or selected without any intersection. 86

ABSTRACT

To deliver the best performance to users, modern compilers apply hundreds of optimizations that transform a program into a more efficient form. Since a program execution is a complicated process of the delicate interplay between software and hardware, each compiler optimization should be carefully determined with consideration for its trade-offs to benefit from the corresponding code transformation. Today, most of the important optimization decisions are made by hand-crafted heuristics which often largely depend on the developers' expertise. However, as the complexity of both software and hardware continues to increase, performance analysis often overly simplifies the interactions between diverse system components and cannot capture system characteristics accurately. As a result, current human-made heuristics generally fail to achieve maximum performance. In the era of limited processor performance gains, this loss is becoming more significant. Furthermore, a huge amount of time and cost need to be repeatedly invested for this manual tuning process whenever one of the system components is updated.

To attack these challenges, this thesis proposes a suite of automatic methods that can successfully improve optimization decisions inside state-of-art compilers by effectively exploiting the knowledge acquired during previous executions. Various representative compilers from diverse domains are investigated with different backends.

First, the decision for a single optimization is investigated. By focusing on *loop unrolling*, which is one of the most representative compiler optimizations, the first part of this thesis suggests a methodology that automatically constructs the best affordable decision model for the dynamic binary translator in mobile system. By effectively learning the

patterns between optimal decisions and workload features from the training dataset, this method significantly outperforms the best heuristics handwritten by industry experts by saving $1.5\times$ dynamic instructions.

Next, a group of optimizations is considered. To identify the best use of existing optimizations, the second work of this thesis proposes an intelligent pure search method, called *SRTuner*, which customizes effective optimization settings for each workload by exposing important inter-optimization relations. Results show that SRTuner accelerates target executions upto $34.4\times$ compared to the highest level of optimization provided by each compiler while outperforming prior state-of-the-art tuning methods. As a byproduct of its unique tuning strategy, SRTuner offers synergistic optimizations which can benefit future tuning strategies.

The third work of this thesis proposes *Collage* which is an auto-tuning system that attacks the practical problem of identifying the best mixed use of diverse backends to efficiently run deep learning workloads on their target devices. The experimental results demonstrate that this system efficiently customizes fast execution strategy that outperforms the hand-written strategies in the existing deep learning frameworks by $1.3\times$ by investing a short amount of tuning time.

Finally, the last work investigates the field of reliability and suggests *RSkip* which provides a cost-efficient protection scheme for a transient fault. By leveraging the trade-off between its overhead and protection rate, RSkip optimizes the protection scheme by creating an approximate copy of the program that will be used to detect a potential fault. With a control for the loss in the protection rate, RSkip could reduce protection cost by $1.83\times$.

The contributions in this thesis improve future optimization decisions in state-of-the-arts compilers by searching for the right use of existing optimizations for each workload and hardware devices and identifying the pattern between good optimization settings and workload characteristics. By delivering better optimizations settings, users can enjoy better performance by better utilizing their underlying hardware.

CHAPTER 1

Introduction

1.1 Introduction

The main mission of a compiler is to transform a program into its best form for the target system by applying hundreds of various optimizations. Since a compiled program will be running on the top of multiple system layers, various interactions between components should be considered to make optimal optimization decisions. For example, *loop unrolling* is a representative compiler optimization that can increase both instruction-level parallelism and optimization window at the cost of certain degree of code bloat. As its actual impact is the combined effect of both advantage and disadvantage from the optimization, loop unrolling may harm performance when it is too aggressively used in the system with restricted memory like embedded system. However, when the increased optimization window can significantly benefit later optimizations or underlying hardware supports high level of instruction parallelism, even embedded system can achieve significant improvement with aggressive use of loop unrolling. Therefore, optimization should be carefully configured depending on the situation for its right use.

Today, such optimization decisions in modern compilers mostly rely on heuristic decision models hand-crafted by system developers [149, 70]. To construct an effective model, developers often conduct performance analysis to capture system characteristics and estimate the impact of an optimization. Although such the manual approach may be feasible, it inevitably requires an expert level of understanding towards the overall target system and a huge amount of person-hours. Given that these non-trivial manual efforts need to be repeatedly invested to reflect any update among the system components, heuristic-based approach can be highly resource-intensive and time-consuming. This may become a serious problem for the fast evolving domain like deep learning. Furthermore, due to the increasing complexity and diversity of the both hardware and software, it gets much harder to model system characteristics accurately. An optimization may have different impact depending on

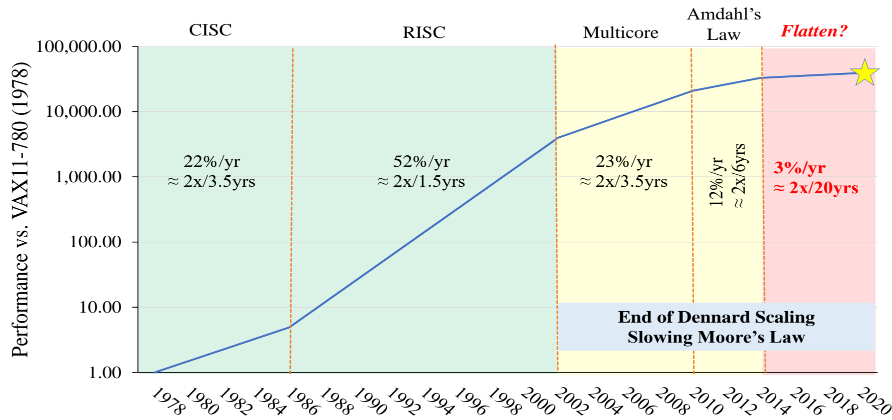


Figure 1.1: Growth of computer performance using SPECintCPU benchmark suite [78]. The end of Dennard Scaling and Moore's Law slows down the performance improvement with hardware innovations. Annotation (star) indicates where we stand today in 2020.

the target hardware, application, or input. On the top of that, various interactions with other optimizations add severe difficulty in performance analysis. As a result, even system experts often overly simplify system characteristics, which leads to the sub-optimal optimization decisions. Stephenson and Amarasinghe reported that a heuristic decision model in Open Research Compiler (ORC) makes optimal decisions only 16% of the time [149]. Also, even the most naive tuner easily outperforms the highest optimization setting in GCC by 11% [70]. These performance losses can be critical since squeezing the last drop of performance actually matters. At the data center scale, 1% of performance improvement can save significant amount of operation cost and there are explosive demands for faster deep learning execution from various domains. Besides, as Figure 1.1 illustrates, two digits of yearly performance gain became almost impossible to expect from hardware improvements due to the end of Dennard Scaling and Moore's Law. Therefore, delivery of better optimization decision is becoming more crucial.

To attack this lack of efficiency, we consider an alternative. To take humans out-of-the-loop and maximize performance, researchers have suggested diverse compiler auto-tuning methods [149, 70, 32, 89]. By exploiting information acquired from previous executions, these auto-tuning approaches improve the future execution. *Iterative compilation* is one of such tuning approaches [42, 90, 124]. Within the limited trials, iterative compilation techniques try to identify the best possible configurations of a set of optimizations for the target program. For example, GCC provides classic one-size-fits-all approaches with standard optimization levels (e.g., -O3) that essentially applies fixed configurations of a set of optimizations for every situation. Prior works have demonstrated that iterative com-

pilation methods can customize better configurations that outperform the most aggressive performance-oriented standard optimization for each situation. To generate promising candidates during a tuning process, these techniques equip diverse strategies of utilizing feedback, such as random search [70], genetic algorithm [68, 83], statistical methods [124, 119]. As machine learning techniques are demonstrated to be very powerful at recognizing the pattern and drawing decision boundaries in multi-dimensional space, various learning-based approaches have been also examined for the the effective use of the prior executions. For instance, Stephenson and Amarasinghe showed that machine learning classifiers can be adopted as a decision model for a single optimization by defining an optimization decision as a classification problem [149].

Inspired by prior approaches, this thesis endeavors to expand the applicability of auto-tuning technologies by examining complicated optimization decision problems in various representative compilers from diverse domains with different back-ends. Depending on the problem, the feasibility of diverse machine learning concepts and techniques is also investigated.

1.2 Challenges and Contributions

This thesis suggests a suite of auto-tuning methods that can effectively improve various optimization decisions inside modern compilers. To demonstrate their effectiveness, state-of-the-arts compilers from different domains are examined with diverse target hardware architectures.

1.2.1 Feasibility Study of Learning-Based Auto-Tuning Methods

To assess feasibility of learning-based auto-tuning, we examine the dynamic binary translator for mobile processor that has hard constraints to satisfy [121]. As memory resources are quite limited in mobile processor, there exist couple of budgets allowed for each optimization in dynamic binary translator. Also, since optimization will be dynamically applied during translation at run-time, each optimization should be completed within certain time limit. Therefore, an optimization decision model for the target system has to be fast and lightweight. Chapter 2 describes how the best affordable decision model can be automatically constructed by supervised learning for a single optimization in this system with tough restrictions. Its performance is evaluated with the hand-crafted heuristics designed by industry experts.

1.2.2 Customization for a Group of Compiler Optimizations

To deliver the optimization setting, state-of-the-arts compilers offer several standard optimization levels, which are a few fixed set of optimization settings under the hood. As the complexity of both hardware and software systems continuously increases, these one-for-all

approach cannot deliver the maximum performance anymore. Despite the obvious opportunity, finding the best use of existing optimizations is known to be highly difficult due to the vast search space and the presence of inter-optimization relations. To tackle down this challenge, Chapter 3 suggests *SRTuner* which customizes the competitive optimization setting by efficiently revealing the interaction between optimizations and using them for the effective tuning process. Due to its unique approach, this method can provide important synergistic optimizations for the given workload at no extra cost. This chapter also discusses how this information can benefit future compiler optimization design and auto-tuning studies.

1.2.3 Search for the Best Use of Diverse Backends

Strong needs for the powerful platform for the efficient Deep Learning (DL) execution has driven rapid evolution in various software backends. These backends include optimized library (e.g., cuDNN, cuBLAS), which provides efficient operator kernels, and inference engines (e.g., TensorRT), which offers a run-time to execute the deep learning models and apply various graph-level cross-kernel optimizations. To extract the best performance, it is key to use the right backend for each case given the strength and weakness of each backend. However, it is difficult to capture the full capabilities of diverse backends with different characteristics. Complex fusion patterns from advanced fusion engines add the complexity. Furthermore, tuning process is not easy since both topology of DL workload and the coverage of each backend should be considered together to find the legitimate candidate of execution plan with multiple backends. As a breakthrough, Chapter 4 proposes *Collage*, which is the auto-tuning system that efficiently customizes the optimal execution strategy with diverse backends. To fully leverage the capability of each unique backends, this system provides the seamless user interface that supports flexible description of pattern rules that each backend follows. For the effective search, this system adopts two-level tuning approach. By ignoring the cross-kernel optimizations in the inference engines, the first level tuner could introduce dynamic programming with efficient cost model that can customize the near-optimal execution plan for the real-life deep learning models within a few seconds. Then, based on the plan created by the first tuner, the second tuner provides the fine-tuning opportunity that makes up for the potential loss from the relaxation regarding the graph-wide optimizations.

1.2.4 Provision of Low-Cost Fault Protection

Increasing failures from transient faults necessitates the cost-efficient protection mechanism that will be always activated. To provide a low-cost software-level protection, Chapter 5 propose a novel prediction-based transient fault protection strategy. Instead of re-executing expensive computations for validation, an output prediction is used to cheaply determine an

approximate value for a sequence of computation. When actual computation and prediction agree within a predefined acceptable range, the computation is assumed fault-free, and expensive re-computation can be skipped. With this approach, a significant reduction in dynamic instruction counts is possible. As missed faults may occur, it is crucial to explicitly keep their occurrences to a small amount by adjusting an acceptable range and aggressiveness of approximation techniques properly. Therefore, with the suggested prediction-based scheme, this work also discusses how to control them accordingly.

1.3 Background

1.3.1 Compilers From Diverse Domains and Their Optimizations

To deliver the most efficient form of binary executable for the given workload and underlying hardware, various compiler stacks have been developed. GCC [21] is one of the most popular state-of-the-arts compilers that provides hundreds of code transformation and analysis passes to optimize C/C++ programs for CPUs. As CPUs aim for the effective general purpose computing, this compiler offers a wide collection of optimizations, such as redundant code eliminations, control flow/data flow optimizations, vectorizations, parallelizations, loop optimizations, and etc [19]. LLVM [22] is another popular compiler technology that is broadly adopted in both industry and academia. By providing the seamless modular design, compiler engineers can easily customize their own compiler and optimizations with LLVM. Likewise, this compiler also equips diverse code transformation and analysis passes [24]. Beyond the powerful CPU support, LLVM recently has started the GPU support [23, 25].

To seek for the optimization opportunity by leveraging the run-time information, the concept of Just-In-Time (JIT) compilation is proposed [94, 84]. Once the hot path is identified during the program execution, a JIT compiler may invest certain compilation overhead to dynamically re-compile this important code region with the run-time observations and extract the further performance gain. These dynamic information may help compilers to make more accurate optimization decisions by providing the run-time context. While using interpreter to offer the fast response time, most Virtual Machines (VMs) adopt the JIT compilation to complement its relatively less efficient quality of generated code [26, 27]. Binary translator [43] also employs the dynamic optimization to improve the translation quality at runtime.

Unlike CPU programs, which require careful handling of the complicated control dependencies, GPU mainly targets data-intensive programs with simple control flow. To promptly process the large volume of data in parallel, modern GPU architectures have thousands of computing units in a number of Stream Multiprocessors (SMs) and the efficient utilization of their resources is key to extract the peak performance. To do so,

the amount of computation should be well distributed to SMs in the target GPU with the careful consideration for its memory architecture. To attack this problem, advanced GPU compilers offer several optimizations, such as loop tiling, loop unrolling, and data layout transformation [128, 55, 117, 116].

OpenCL compilers have endeavored for the effective heterogeneous computing and the performance portability across a wide range of hardware devices with different design philosophy. By mainly targeting important computations, such as BLAS kernels, they apply various optimizations for data-intensive workloads, including loop tiling and loop unrolling [116, 117].

To provide the powerful platform for Deep Learning (DL) workloads, various DL frameworks have been actively developed. TensorFlow [10], PyTorch [9] and TVM [55] are the representative examples. By targeting a wide range of hardware devices, including custom accelerators [2, 1], these framework apply graph-level optimizations on the top of the conventional compiler optimizations. For instance, computation graph of the DL workload may be rewritten in the more efficient form as long as the correctness of program is guaranteed. Operator fusion [6, 115, 55] is also a powerful optimization technique that merges multiple kernels to improve memory usage and scheduling overhead. To automatically generate an effective operator kernel, TVM provides a template implementation of each operator with various optimizations, such as loop unrolling, loop tiling, and etc. When targeting accelerators, users may also consider tensorization to fully utilize their custom hardware.

1.3.2 Compiler Auto-Tuning

As modern compilers generally provide manifold optimization/analysis passes, it is known to be highly challenging to identify the best use of existing optimizations. Both the impact of a single optimization and the interaction between optimizations would be different depending across workloads and hardware devices, which adds the complexity of the problem. Thus, to extract the maximum performance for the workload, compilers often need to find out the best combination of optimization decisions [32, 56, 42, 89], the optimal ordering between optimization passes [93, 96, 95, 60], and the best use of various backends, such as optimized libraries (Figure 1.2). To attack these problems, one of the most popular methods adopted in the current state-of-the-arts compilers is to provide standard optimization levels, which is essentially a few set of fixed optimization settings hand-tuned by system experts. However, due to the increasing complexity of both hardware and software system, this approach cannot deliver the best performance to users anymore by making sub-optimal decisions for many occasions [121, 108, 70].

As a breakthrough, various auto-tuning techniques have been proposed [32, 56, 101,

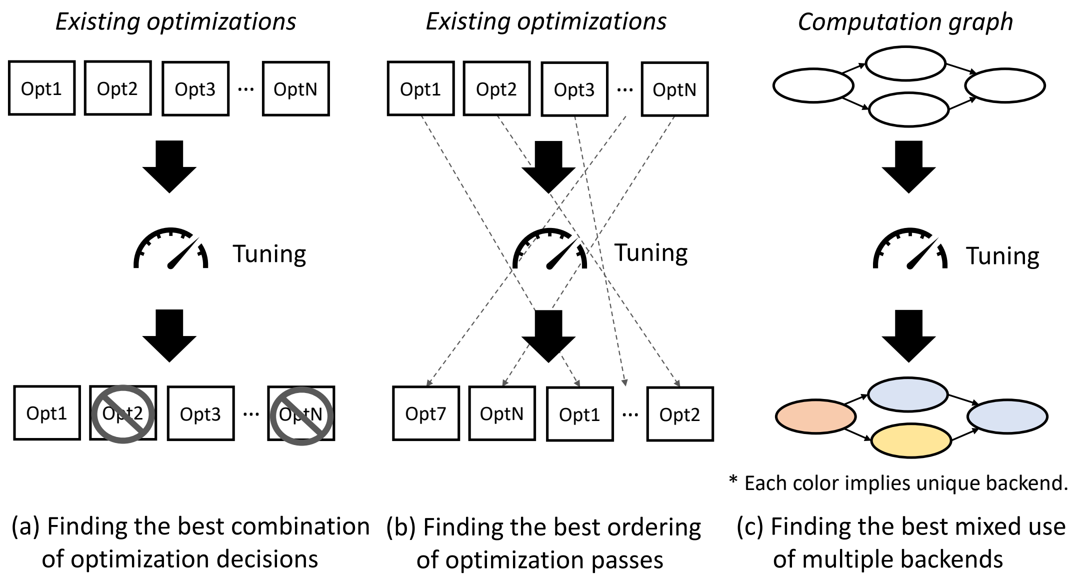


Figure 1.2: Representative tuning problems

108, 70, 121]. Largely, these technologies can be classified into two groups (Figure 1.3): (1) workload-specific tuning: This approach explores the large and complex optimization space by investing separate tuning time to customize the best possible optimization decisions for the given workload. Pure search methods [32, 90, 68, 119] is one of the representative examples that navigates the search space by only using the feedback acquired during the tuning process without any prior information, such as the cost model and prior tuning experiences. (2) generalization: This approach tries to recommend the good optimization settings for the unseen workload without the expensive investment of tuning time. By learning the relationship between the workload features and its customized optimization settings in the training dataset, these techniques can provide more flexible optimization settings efficiently [70, 34, 121, 75]. A high quality of training data is necessary for the effective training and thus, generalization methods often adopt the pure search methods to prepare their data.

In general, workload-specific tuning is not cheap and most of its tuning overhead is from the repetitive run-time evaluations that include compilation and run-time measurement. To alleviate the cost, researchers often introduce the hardware cost model which can approximate the performance of an optimization setting without the actual measurement on the hardware. However, due to the high complexity of the computing system, it is not easy to construct the effective cost model by hand. To resolve this issue, the idea of learning-based approach has been proposed [56, 28]. By utilizing the run-time samples acquired during the tuning process, this approach automatically constructs the effective hardware cost model.

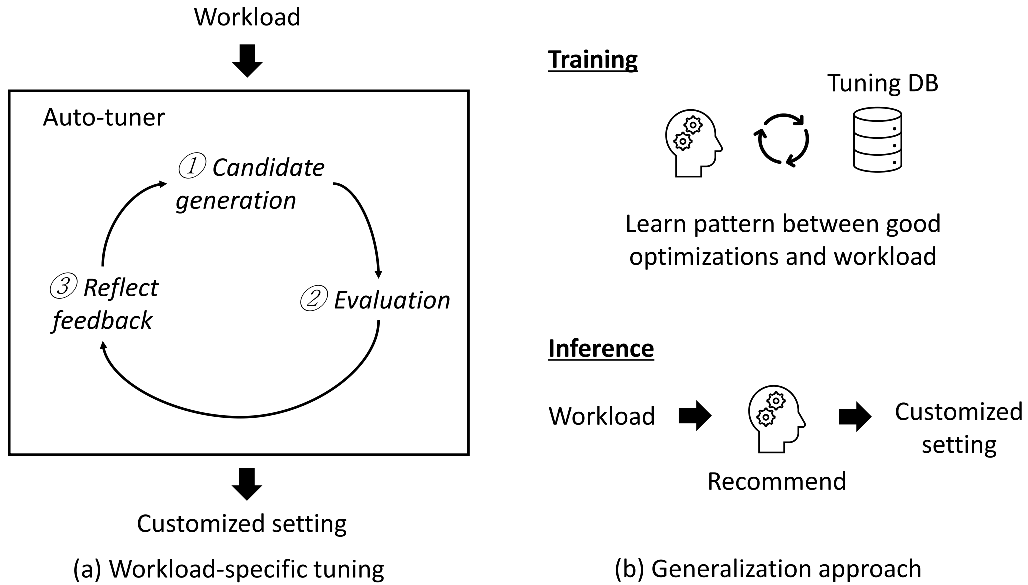


Figure 1.3: Representative tuning approaches

To maximize the efficiency, these methods often support transfer of the cost model across different tuning jobs.

To respond to high demands for the faster platform for DL workloads, there has been various active studies to fine-tune their executions. To automatically create the competitive implementation of DL operators, TVM offers two different auto-tuning technologies. If users want to customize the optimization settings within their manually defined search space, AutoTVM [56] can be used. Otherwise, Ansor [167] may be considered to traverse broader space with a hierarchical representation of the program. TVM also provides a graph tuner that optimizes layout transformation for CPU workloads [55]. By leveraging a set of re-writing rules, graph re-writing methods, such as TASO [85], explore the promising variants of the computation graph of the given workload to identify the best form for the underlying environment (e.g., hardware, compiler). For the efficient evaluation of each candidate, these methods often adopt the simple heuristic of cost model.

To construct the effective generalization model [121, 70, 34] by reusing the past tuning experience, it is key to have a high quality of training data that consists of workload features and their effective customized optimization settings as the ground-truth. As previously discussed, optimization settings can be automatically identified by various pure search approaches. For the effective learning, feature selection is crucial for the accurate workload characterization to provide important information that is necessary for the optimization decisions. Thus, to identify the key information for optimizations, it is often useful to understand which applications are similar from an optimization perspective. Although there

have been manual approaches that rely on the expert's insight [70, 34], this is generally known to be extremely challenging. To avoid difficulty of manual feature selection, compiler researchers have proposed automatic feature generation techniques [97, 98]. By navigating the given search space, these methods examine the impact of each feature and choose the ones with high importance.

The comprehensive survey for auto-tuning domain is well presented by Ashouri et al. [33].

CHAPTER 2

Multi-objective Exploration for Practical Optimization Decisions in Binary Translation

2.1 Introduction

Today, many important optimization decisions are made by heuristics, which often depends on the developers' expertise. With an expert level of understanding of the system and a huge amount of effort, programmers can create effective models that capture architectural characteristics [163]. However, due to the increasing complexity of architectural design, it becomes much harder to build effective human-made models. Additionally, the subtle interactions between software optimization phases add to the exploration space. As a result, heuristics often fail to make good decisions. Stephenson and Amarasinghe showed that the loop unrolling heuristics in Open Research Compiler (ORC) only achieve 16% prediction accuracy for optimal unroll factor [149].

To make effective optimization decisions, researchers build decision models by applying machine learning techniques [149, 110, 51, 150]. Particularly, given its system-wide impact, researchers have studied how to improve the optimization decision for loop unrolling. By relaxing loop-carried dependencies, properly applied loop unrolling can increase Instruction Level Parallelism (ILP) and bring more opportunities for subsequent optimization phases, resulting in significant performance improvement. However, it may cause code bloat or large numbers of register spills when it is applied too aggressively. For the best use of loop unrolling, Stephenson and Amarasinghe suggested defining loop unroll factor prediction as a classification problem [149]. By employing supervised classification techniques, they successfully improve prediction accuracy as well as loop performance over a baseline heuristic. These previous works are built by using statically known information and inserted into a static compiler. Since they target a static compiler, the overhead (computation and memory) of the built model is not considered.

On the other hand, hardware/software (HW/SW) co-design has been extensively studied

for mobile and embedded systems to achieve better performance or reduce design cost [37, 160, 12, 159]. A DBT system is a key component in such a co-design process. Since the DBT system conducts optimization during translation [43], the optimization decision has a direct impact on the quality of the translation. Thus, smart optimization decisions are necessary to provide high translation quality. To make better decisions, we examine an approach to tailor machine learning optimization decision models for a code optimizer in DBT. Particularly, we target a mobile processor that supports high-performance applications but still operates in a constrained environment (e.g., a mobile processor for an autonomous vehicle). Unlike previous works [149, 110, 97] in a static compiler, the prediction overhead (e.g., memory usage, performance overhead, and energy consumption) of a decision model may restrict the usage of a complex or large decision model in the mobile system.

As the first step, we choose *loop unrolling* as a representative optimization and investigate five different multi-class classification techniques and their diverse configurations to build its effective decision model. Note, this work focuses on the single optimization decision while leaving the decision for a group of optimizations for Chapter 3. This is also a multi-objective exploration observing the relationship between prediction overhead and accuracy. We build our approach with the industrial strength DBT infrastructure. Our code optimizer has an optional optimization for loop unrolling, called *smart unrolling*, that can remove redundant branches [107]. Since smart unrolling can have performance side effects, the built model must make an additional decision on whether to apply smart unrolling. Furthermore, unlike previous works [149, 110, 97] in a static compiler, some useful but expensive analyses like dataflow analysis (DFA) may not be available due to their high overhead in binary translation. Instead, new opportunities to utilize dynamic information (e.g., loop trip count, taken probability of side exits) collected during runtime are studied. These turned out to be important features for unroll factor classifiers.

For experiments, we collected 17,116 unrollable loops from 200 real-life programs and benchmarks in various domains. By employing all available features that might be crucial for loop unrolling decision, we suggest the best classification algorithm for our infrastructure given both prediction accuracy and cost. Then, we identify its significant features to prune the feature set and evaluate the model built by using selected important features. As a result, the best affordable classifier that is within the memory/time budgets for the decision process shows a 74.5% of prediction accuracy for optimal unroll factor and realizes an average 20.9% reduction in dynamic instruction count during the steady-state translated code execution when the ideal upper bound for instruction reduction is 23.8%. For comparison, the best current heuristic shows a 46.0% prediction accuracy with an average of 13.6% instruction count reduction.

The major contributions of this work are as follow:

- We show how machine learning techniques can improve loop unrolling decisions for dynamic binary translation on the mobile processor, which is a more challenging environment than static compilers used in previous works [149, 110, 97]. The demanding environment requires a more careful model selection with multiple objectives considered which is not necessary for a static compiler. New opportunities to employ dynamic information is also examined. This approach is instruction set independent and can be extended to other optimization decisions.

- We investigate the relationship between prediction overhead (i.e., time, memory) and accuracy for diverse classification algorithms and their different configurations. As a result, the best classification algorithm is discovered. Then, by applying feature space pruning technique, we provide its major features and suggest the best affordable decision model for our infrastructure given the specific budgets allocated for the decision process.

- We compare the current heuristics in the industrial strength infrastructure and the proposed machine learning based approach. Given that the infrastructure is already highly optimized, the heuristics are well crafted and provide a challenging baseline for comparison.

2.2 Background

2.2.1 Our Infrastructure with Binary Translation

Figure 2.1 sketches our HW/SW co-designed CPU. Although this CPU only accepts the binary code written in x86, it incorporates a hidden microarchitecture that supports additional Instruction Set Architecture (ISA) internally. By having a DBT system that translates legacy code into internal ISA and orchestrates execution between two different architectures, the internal ISA and its architectural design can stay invisible from the outside and be innovated without worrying about backward compatibility. To manage program execution, our infrastructure employs the region-level atomic execution similar to [154]. The dynamic binary translation system profiles the execution and constructs optimization regions to translate x86 instructions in the hot traces into internal ISA instructions that will be executed on its optimized microarchitecture. To maximize the performance benefit, it is crucial to produce a high quality of translated code. Therefore, during translation, the dynamic optimizer applies various code optimizations [154], including loop unrolling, to improve the translation quality.

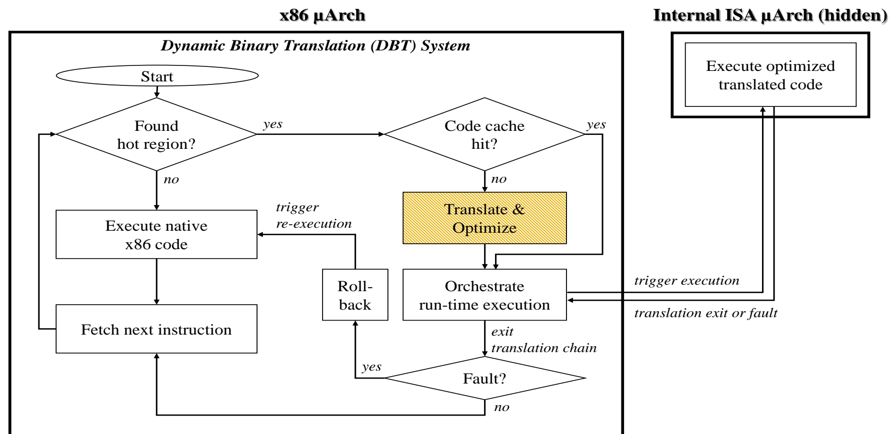


Figure 2.1: Big picture of our HW/SW co-designed CPU with a DBT system. It constains two different microarchitectures that each supports its own ISA. The highlighted box illustrates the translation process by a binary translator and a dynamic optimizer.

2.2.1.1 Target Loops

- Reducible (Single entry) loop.
- Innermost loop. The outer loop is only considered when the innermost loop is fully unrolled.
- Both *counting* and *non-counting* loops with a loop invariant trip count [59].

2.2.1.2 Smart Unrolling

Our dynamic optimizer has an aggressive version of loop unrolling which is an optional optimization when loop unrolling is enabled. By targeting counting loops, the technique tries to transform the loop structure and exit condition to minimize the number of branches [107]. For example, the optimization can eliminate loop exit branches in copies of a loop body. Also, the optimization phase inserts a run-time check outside of the loop to prevent exceptions (e.g. overflow).

Smart unrolling can have side effects. When the run-time check fails, deoptimization will be triggered to revert the code and degrade the performance. Also, applying smart unrolling to a loop with a low trip count may result in performance degradation due to the overhead of the transformed structure. Thus, smart unrolling should be applied carefully to realize performance improvement.

2.2.1.3 Current Heuristics

There are two unrolling heuristics in our dynamic optimizer: *Optimistic* and *Conservative*. By default, *Optimistic* is used to optimize identified hot traces. When speculation with

Optimistic fails repeatedly [61], *Conservative* is used to generate optimized code with less aggressive optimization. Both heuristics make a decision based on identical information such as trip count, the expected number of post-unroll instructions, etc. The only difference between the two is two parameters: the maximum number of post-unroll instructions and the maximum unroll factor. *Conservative* has lower values than *Optimistic*. Each heuristic predicts an unroll factor that is lower than its upper limits. The maximum unroll factor is set to 8 for *Optimistic* and 4 for *Conservative*, respectively. Also, although smart unrolling may have side effects, current heuristics always apply it on counting loops with the optimistic expectation.

2.2.2 Supervised Multi-class Classification

Supervised classification is a process identifying which set of classes (or labels) a new observation belongs to, based on the learning from training data. This section describes five representative classification techniques as background to deliver their main concepts with the pros and cons.

2.2.2.1 k-Nearest Neighbors (kNN)

kNN classifies a new observation based on the majority voting from k number of closest neighbors in training data. The idea is straightforward: find the most similar case from the training database and assign the same label to the new observation. Thus, in our case, kNN will search the most similar loops from the training data and assign the dominant label among them. Without having any form of generalization process on training dataset, training data will be saved and populated directly during the inference. Since kNN scans data points in the training set to find the closest neighbors, its prediction cost is proportional to the size of the training data set. Also, its learned model is unable to be interpreted and give any intuition to the system designers.

2.2.2.2 Support Vector Machines (SVM)

SVM makes a classification based on the decision boundaries. During training, the technique constructs decision boundaries that separate the training data points. Rather than focusing on minimizing prediction errors on the training set, it tries to minimize the expected generalization loss based on the probabilistic assumption for unseen data. Naturally, SVM is resistant to overfitting [138]. To divide data points under different classes while minimizing the generalization loss, SVM chooses each decision boundary that is farthest away from the observed training data among all possible boundaries. Thus, each selected decision boundary is also called the *maximum margin separator* and the points closest to the separator is called *support vector*. However, it takes a long time to train this model and the learned

model is not comprehensible. Fundamentally, *SVM* manages multiple two-class decision boundaries to conduct multi-class classification. Therefore, the prediction cost may become expensive as the number of boundaries increases.

2.2.2.3 Decision Tree

A decision tree is a function that makes a decision by conducting a sequence of tests [138]. Each node in a tree checks the value of one of the input features and guides to the next node until the final decision at the leaf node. During training, a decision tree learns what will be tested at each node. If a feature is numerical, the threshold will also be determined for each node. Fundamentally, the decision tree consists of nested conditional statements. Therefore, unlike other machine learning techniques, the learned model is able to be interpreted and easy to visualize. This is a noteworthy property in the sense that the learned model can give system designers insights that they may have been missing. In addition, the worst case for the prediction cost is proportional to the maximum height of the tree. Note, the computation at each node is quite cheap since it is usually just a simple comparison. Thus, if it is able to build an effective tree with the control of the maximum height, the model can be highly practical. However, the technique may suffer from overfitting.

2.2.2.4 Random Forest

Random forest manages a multitude of decision trees and makes a prediction by aggregating the predictions from trees in the forest (e.g., majority voting). Since the random forest is essentially a group of decision trees, the learned model is also easy to visualize and is able to give information about the relation between the feature set and the prediction. In general, the technique is able to convey high accuracy and efficiently handle a large dataset with high dimensionality of feature space. Also, the estimate of generalization loss will be computed during training, which can be used to enhance overfitting. However, when the forest grows a multitude of large trees in parallel, it may require a lot of memory [100]. This may cause an overhead to the memory system and increase the inference time affecting the translation time in our case.

2.2.2.5 Artificial Neural Network (ANN)

ANN is a learning method that mimics the brain activity mathematically. The model consists of multiple layers: an input layer, hidden layers, and an output layer. Each layer contains a large number of neurons and each neuron is connected to other neurons in the other layer. In this work, a fully-connected network, whose each neuron is connected to all neurons in the next layer, is assumed. In general, the neural network is known for its outstanding classification accuracy compared to the traditional learning techniques. However, it may

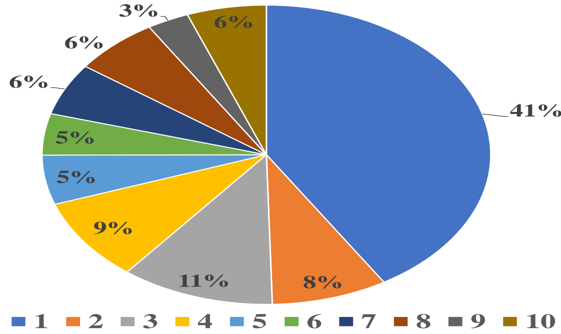


Figure 2.2: Distribution of optimal unroll factors.

need a large network and a large training dataset to achieve satisfactory accuracy [165]. The inference can be very compute-intensive, particularly with a large network. In addition, the decision-making process is like a "black box" so that the internal mechanism would hardly give any intuition to the system designers.

2.3 Motivation

Our current DBT system employs heuristic-based decision models designed by industry experts. Given the difficulty of creating effective models that describe subtle interaction between optimization phases while satisfying hard environmental constraints, we notice the pitfall that may exist: hand-made heuristic models might be biased by the designer's experience or insight. Therefore, we recognize the need to not only evaluate their effectiveness, but also develop the automatic and systematic approach to building decision models to lessen the burden required for their design process (e.g., analysis, tuning). Without an automatic method, system designers may need to repeatedly invest a huge amount of efforts to tune their optimization decision whenever other system component(s) are updated.

To investigate the feasibility of such an approach, we initiate the study by focusing on loop unrolling given its system-wide impact. By examining all possible configurations for the loop unrolling decision, the optimal unroll factor for each unrollable loop is identified and the performance of the current heuristic designs (i.e., *Conservative* and *Optimistic*) are measured. Section 2.5 explains our methodology in more detail. To evaluate if the upper bound of the unroll factor is reasonably set in the current heuristic design (e.g., 8 for *Optimistic*), we explore unroll factors ranging from 2 to 10.

Figure 2.2 shows the histogram of optimal factors across all collected loops. An optimal unroll factor of 1 represents the case that loop unrolling should not be applied. Other than the leftmost bar, there is no dominant unroll factor. This implies that the optimal unrolling factor varies depending on the loop characteristics and thus, the unrolling decision should

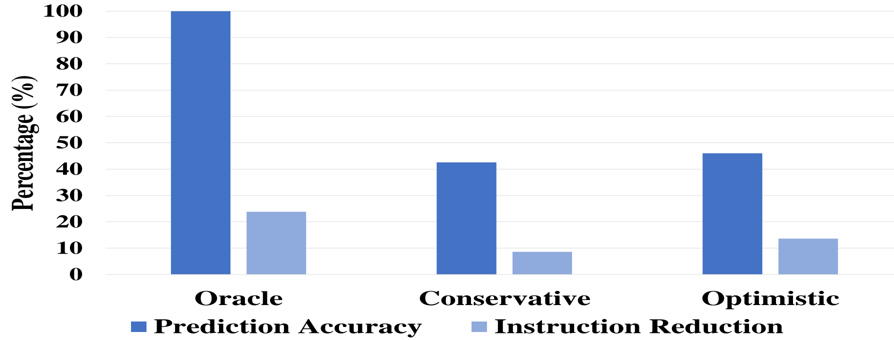


Figure 2.3: Evaluation of current heuristic design. Prediction accuracy for optimal loop unrolling decision and its impact on the dynamic instruction count in the resulting optimized code are measured.

be made carefully. Also, heuristics are unable to predict optimal factors of 9 and 10, which accounts for 8.7% of loops. This suggests the necessity for the higher upper bound on unroll factors.

Figure 2.3 depicts the effectiveness of current heuristic designs. Note, *Oracle* represents an ideal model that always makes the optimal decisions. Thus, loop unrolling can realize 23.8% of instruction count reduction at most. However, the best heuristic adopted by a dynamic optimizer in our DBT system only shows 46% of accuracy with 13.6% of instruction reduction in the translated code. This suggests the opportunity for improvement with a better decision model.

2.4 Challenges and Opportunities

This section explains why previous works with a static compiler [149, 110, 97] cannot be applied directly to dynamic binary translation for a mobile system. New challenges and opportunities in adapting machine learning based optimization decision models to our environment are threefold:

- **Limited analysis support and increased complexity to make optimization decisions:** Some useful information that is concluded important for a static compiler [149] are not available at the loop unrolling phase in our dynamic binary translation due to their high analysis overhead: dataflow analyses, live range size, instruction fan-in, critical path length, etc. Additionally, the optimization phases in the dynamic optimization are more tightly coupled to each other than a static compiler [83]. Furthermore, in our dynamic optimizer, there is an additional decision for smart unrolling. These add a complication to the analysis and broaden the exploration space for optimization decisions. Therefore, the optimization decision model in a dynamic optimizer should be able to make more complicated decisions with restricted information. A

key question is: Can an accurate machine learning model be created without high overhead analyses and handle the additional complexities presented in dynamic binary translation?

- **Strict restrictions on the overhead for an optimization decision:** Since optimization time is an extra run-time overhead in dynamic optimization, the cost for optimization decisions also incur run-time overhead [92]. Consequently, a dynamic optimizer needs to make smart decisions carefully to achieve a good balance between cost and overall performance benefit. To find a good balance, system designers often set strict time/memory constraints on each optimization phase. In our experiment, SVM and nearest neighbor, which are recommended for the static compiler [149], showed good accuracy improvement compared to the baseline heuristics. These techniques, however, presented $8,095\times$ and $660\times$ slower decision-making with significant extra memory requirements. Key questions to address are: Are these methods affordable for a dynamic binary translation in the mobile system? Can a better decision model be built for our environment?

- **The availability of dynamic information:** In dynamic optimization, a new opportunity to use dynamic information arises. Since translation is triggered during runtime, dynamic information, which is more accurate than profiled information for a static compiler [149], is available. However, collecting dynamic information also creates run-time overhead. Thus, the kinds of information that can be collected are limited. A critical question is: What type of dynamic information would be informative for the loop unrolling decision and also affordable in terms of time and storage overheads?

To answer these questions, we explore diverse machine learning techniques to evaluate the feasibility of each approach. Since the classifier will be tested to categorize a new observation based on the achieved knowledge from the learning, generating meaningful training data is a key process to build a good classifier. We describe our data generation process in Section 2.5 and assess each classifier in Section 2.6. Then, Section 2.7 illustrates our feature selection technique for the machine learning algorithm.

2.5 Data Generation

For each unrollable loop, a set of features and its optimal unroll factor are extracted and combined to generate data for the classifier. Details for data generation process is described in the following sections.

General loop property
Number of side exits, Number of outer loops, ...
Constraints from binary translation
Proportion of post-optimization instructions compared to its quota, ...
Benefit: opportunity for the following phases
Number of invariant loads, ...
Benefit: ILP → Unavailable
~~Number of parallel computations in loop, ...~~
Side effect: code size
Number of static instructions, ...
Side effect: register pressure → Unavailable
~~Live range size, Number of uses/defs, ...~~
Dynamic information
Trip count, Taken probability of side exits, ...
Instruction mix
Ratio of static loads/stores/branches, ...
Smart unrolling
Size of immediate operand in induction variable, ...

Table 2.1: A subset of features for loop unroll decision under different categories. The ratio of each static operation is computed by dividing the number of each operation by the number of static instructions. Note, unavailable features (e.g., live range size) in our infrastructure are crossed out. The total of 34 features are extracted and used for the experiment.

2.5.1 Feature Extraction

For an accurate decision, characteristics of a loop must be captured properly. Thus, important loop information that can affect the decision for loop unrolling is introduced as a feature. For a fair comparison with current heuristics, we use the information that is already available at the optimization phase without an additional profiling or heavy analysis.

For feature selection, we define 9 categories of features across various loop information as presented in Table 2.1. To manage the quality of translated code regions, our DBT infrastructure puts certain restrictions (e.g., an upper limit for the expected number of post-optimization instructions) in each optimization phase. Thus, the constraints from the binary translation are introduced as features. To estimate the impact of loop unrolling, we also define categories indicating the benefit/side effect. However, the information for instruction level parallelism or register pressure cannot be used for our experiment since it requires dataflow analysis which is unavailable in our loop unrolling phase. In addition, the information for loop characterization, smart unrolling, dynamic information, etc. is employed as a feature. In total, 34 features are extracted and used for the experiment. These 34 features include all 8 features employed by heuristics.

2.5.2 Optimal Factor Exploration

For supervised learning and prediction accuracy measurement, the optimal factor for each set of features should be identified. Thus, we follow an exhaustive approach. Each benchmark trace is executed multiple times with all possible optimization decisions. In this work, unroll factor prediction should make several decisions: (1) Whether loop unrolling should be applied. When loop unrolling is expected to bring performance degradation, the decision model should not enable it. (2) Whether smart unrolling should be applied. (3) Unroll factor. In terms of the unroll factor, we explore the range from 2 to 10. Therefore, each benchmark is executed 19 times with different configurations in loop unrolling:

$$N_{nounroll} + N_{smart} * N_{factor} = 1 + 2 * 9 = 19$$

Each $N_{nounroll}$, N_{smart}, N_{factor} represents the number of cases when unrolling is disabled, cases for smart unrolling (Enable/Disable), and cases for unroll factor (2 to 10) respectively. As we investigate 19 configurations, there exist 19 labels that the classifier considers. This exploration space is more than twice compared to previous work [149].

During each run, the dynamic optimizer identifies unrollable loops and dumps their feature sets. Then, the optimizer forces the given configurations on the loops and measures their performance. In this work, the number of dynamic instructions is measured and used as the performance metric. We discuss our approach further in Section 2.6. Different performance metrics can be employed depending on the objective of the system design.

After execution, the performance numbers between different configurations are compared to find optimal configurations for each unrollable loop. When the numbers draw, the smaller unroll factor is preferred as it has a smaller side effect (e.g. code size). Secondarily, if the numbers between smart unrolling and regular loop unrolling with the same unroll factor draw, smart unrolling is picked with the optimistic expectation: when the speculative assumption for smart unrolling holds, it would outperform regular loop unrolling. The identified optimal unroll factors for each feature set will be used as training data for supervised learning and testing data for the evaluation.

2.6 Evaluation

2.6.1 Experimental Setup

Classifiers with five different supervised multi-class classification methods are built by using the Python scikit-learn library [122]. To improve the performance, the library implements its core algorithm with Cypthon [39], a package for C-Extension, and compiles it with -O3 optimization level. The selected classification methods include nearest neighbor and SVM

which are recommended in previous study for the static compiler [149]. To observe the relationship between prediction cost and its benefit, we explore different configurations for decision tree and random forest. In the library, their configuration can be controlled indirectly by providing the maximum depth and the maximum number of leaf nodes. The parameter setting for each classifier is described in Table 2.2. For instance, by setting $k = 3$, *Nearest Neighbor* conducts majority voting from three most similar loops. The classifiers which are not covered in the table use the default setting. Due to the compute-intensive nature of neural network algorithm, we inspect a simple network to investigate its applicability.

For performance evaluation, we base our approach on the industrial strength DBT infrastructure and examine the quality of the translated code (i.e., internal ISA). In our experiment, both instruction sets run with a simulator that models our new SW/HW co-designed processor. Since only a functional simulator was available, alternatively, the number of dynamic instructions is measured and used as the performance metric. Although this approach has the restriction as it may not show the overall impact of the optimization directly, researchers often adopt dynamic instruction count as an approximation for the execution time [130, 79]. For example, Ravindar and Srikant use the dynamic instruction count to estimate Worst-Case Execution Time (WCET) [130]. For the same reason, we could not directly measure the overhead of classifiers by embedding them into our dynamic optimizer. Instead, performance improvement analysis and overhead analysis for each prediction technique are conducted separately. Thus, our performance improvement analysis shows the enhancement in the optimized translated code after the execution is stabilized. For overhead analysis, time and memory usage per prediction are examined for each decision model on an Intel Core i7-4700MQ mobile processor assuming the x86 architecture in our CPU design. For a fair comparison with classifiers, we implemented both heuristics algorithms (i.e., *Conservative*, *Optimistic*) in Python with C-Extension and configured with the equivalent optimization level to measure their overhead on the identical conditions.

By following a similar approach with SimPoint [123], we gathered data of 17,116 unrollable loops in the hot traces from 200 real-life programs and benchmarks in various domains. The representative benchmark suites and the breakdown of collected loops for each domain are as follows: embedded/enterprise/games (FPMark [14], Geekbench [15], SYSmark [16], TabletMark [17], 3DMark [13], etc.:49%), performance (SPEC CPU '06/'17 [80, 48]: 41%), machine learning (MLbench [103], etc.:10%). Diverse operating system environments are also considered: Windows, Linux, Mac, Android, etc.

Throughout the experiment, we conduct the *stratified 5-fold cross-validation* [164]. On average, we train each classifier with 13,652 loops and test it with 3,423 unseen loops. The

Classifier	Parameter Setting
Nearest Neighbor	k = 3
* Decision Tree_0	-
Decision Tree_1	max.depth = 20, max.leaf = 500
Decision Tree_2	max.depth = 15, max.leaf = 200
Decision Tree_3	max.depth = 10, max.leaf = 50
* Random Forest_0	trees = 10
Random Forest_1	trees = 5, max.depth = 10, max.leaf = 50
Neural Net	network size = 34 * 100 * 19

Table 2.2: Configurations for the classifiers. Annotated (*) configurations are generated without any restriction on both the maximum depth and the maximum number of leaf nodes.
loops are divided randomly and there is no intersection between them.

2.6.2 Prediction Accuracy

The accuracy is measured by counting the total number of correct classification in comparison to the optimal label out of all test data. For a better understanding of the classifier, we define four different classes for accuracy so that a single prediction can be evaluated in four different perspectives. Each class has a unique definition for ”correct classification” as follows:

- ***Exact***: Accuracy for exact prediction in all unrolling, smart unrolling and unroll factor decisions.
- ***Unroll***: Accuracy only for the loop unrolling decision.
- ***Smart***: Accuracy only for the smart unrolling decision.
- ***Dist_n***: Compare the unroll factor only and accept the difference in factors within ’n’. It ignores the decision for smart unrolling.

In the dynamic optimization, the decision of whether to apply the optimization has more importance than the one in the static compilation as the optimization process is a part of a run-time overhead. Therefore, we measure the accuracy of each optimization decision in loop unrolling: *Unroll*, *Smart*. Also, comparison of *Dist_n* gives the indication of how far the prediction goes from the optimal label in terms of the unroll factor.

Figure 2.4 presents the accuracy of heuristics and classifiers. For *Exact* class, *Conservative* and *Optimistic* show 42.6% and 46.0% of accuracy respectively. Overall, all machine learning techniques outperform heuristics. Especially, *Decision Tree_0* and *Random Forest_0* show the best accuracy which is around 75%. They are the biggest configurations for decision tree and random forest algorithm respectively. Their accuracy goes down with smaller

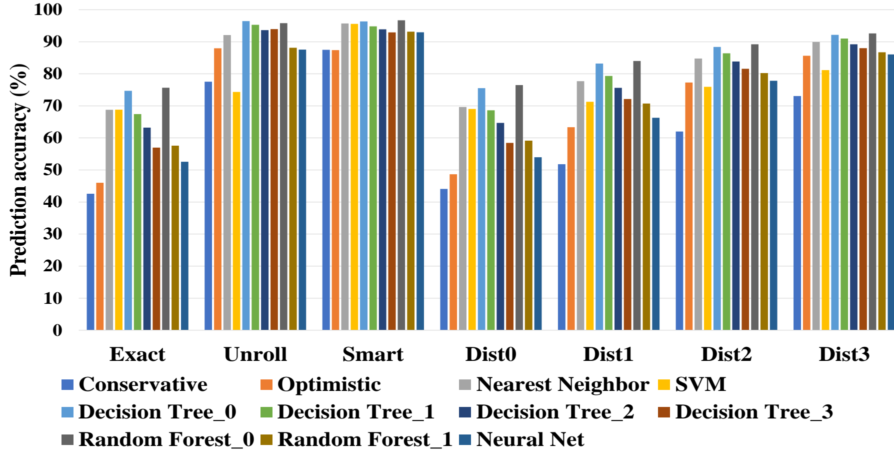
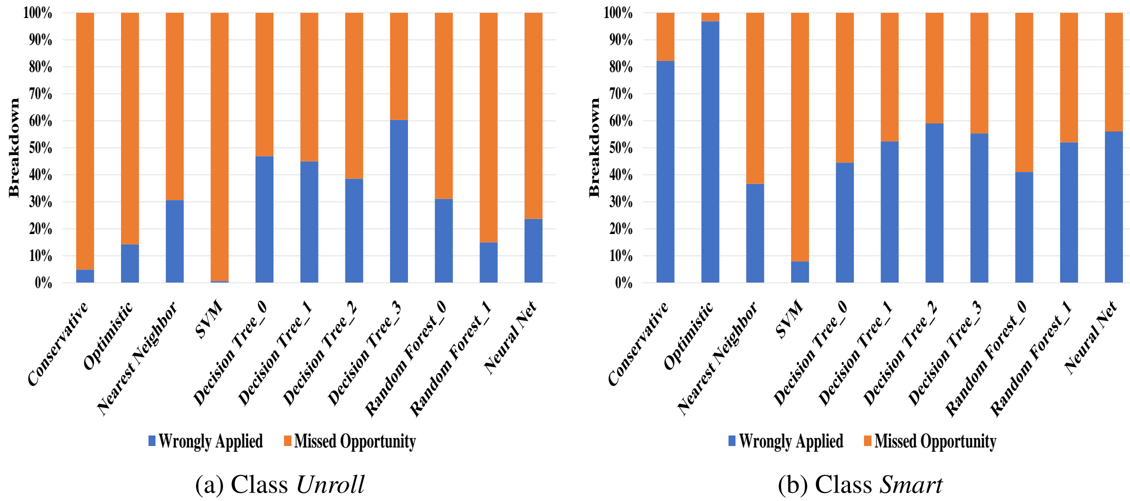


Figure 2.4: Prediction accuracy with various classes.



(a) Class *Unroll*

(b) Class *Smart*

Figure 2.5: The breakdown of wrong predictions.

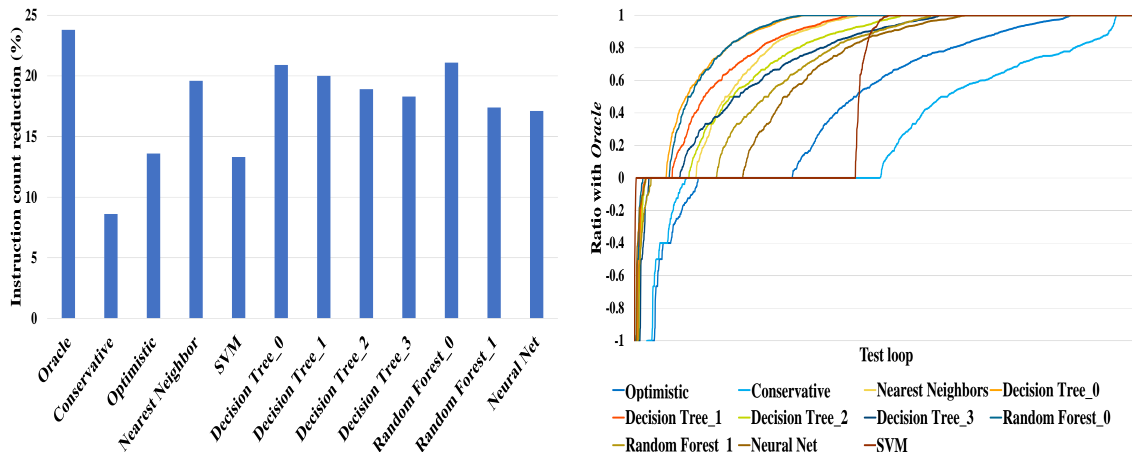
configurations. For *Unroll* class, heuristics work pretty well. *Optimistic* has 88.0% accuracy, which is better than *SVM*, *Random Forest_1*, and *Neural Net*. Interestingly, *SVM* shows the lowest accuracy among all predictors, even lower than *Conservative* although it shows the decent accuracy for *Exact* class. Other classifiers, such as *Decision Tree_0*, surpass the accuracy of heuristics. For *Smart* class, all machine learning models outperform heuristics by presenting the accuracy above 90% even though both heuristics shows desirable accuracy. Although heuristics' optimistic expectation generally holds, this suggests an opportunity for improvement in their current design. For *Dist_n* class, most models increase their accuracy as the allowance in the difference of factors grows. Particularly, the increases in *Optimistic*, *Decision Tree_2*, *Random Forest_1*, and *Neural Net* are notable. This implies they make a good deal of sub-optimal predictions that are close to the optimal labels. However, the

accuracy change in *SVM* is not significant compared to others. This might be related to its noteworthy low accuracy for class *Unroll*.

To understand the misprediction for *Unroll* and *Smart*, we collect its sources of prediction failures. A prediction failure falls into either one or the other: *wrongly applied*, or *missed opportunity*. Figure 2.5 illustrates the ratio of wrongly applied and missed opportunities among mispredictions in the decision model. The breakdown for class *Unroll* is shown in Figure 2.5a. In general, most decision models have turned out to have lots of missed opportunity which suggests a room for performance improvement. Especially, *SVM* has a conspicuous amount of missed opportunity resulting in significantly low accuracy for class *Unroll*. The classifier disables loop unrolling more than necessary, resulting in overly conservative predictions. This can also explain the observation of *SVM* in class *Dist_n*. Given that *SVM* shows high prediction accuracy in *Exact* class, the classifier is expected to have a very high accuracy to figure out when it should not apply loop unrolling. Figure 2.2 backs up this phenomenon. For around 40% of loops in our benchmark traces, it is the best decision to disable loop unrolling. When comparing *Conservative* and *Optimistic*, the former makes more missed opportunity than the latter although it has less wrong applications of loop unrolling. This is expected when considering the nature of the conservative approach. On the other hand, Figure 2.5b shows the breakdown for class *Smart*. Notably, many decision models wrongly apply smart unrolling in many cases. Especially, both heuristics have a significantly high ratio of wrongly applied which is greater than 80% and 95% respectively. This implies overly optimistic expectation from heuristics would result in misuse of smart unrolling around 10% of the time in their predictions. The high misuse rate is induced by heuristics' overly optimistic expectation towards the speculative assumptions for smart unrolling. Meanwhile, *SVM* presents its conservative approach by showing a significantly low ratio of wrongly applied.

2.6.3 Performance Improvement In Translated Code

To evaluate the steady-state performance of the translated code, expected instruction reduction for each loop is computed by using collected data. Figure 2.6a represents the average in instruction count reduction for test loops. The first bar, *Oracle*, shows the expected instruction reduction of a perfect predictor which always predicts optimal labels. Therefore, its number (23.8%) will be the ideal upper limit that the decision model can achieve. The best heuristic in our DBT infrastructure presents 13.6% of instruction reduction. All but *SVM* outperforms the best heuristic significantly. This can be explained by *SVM*'s significant amount of missed opportunity for loop unrolling as presented in Figure 2.5a. Other classifiers notably outperform the best heuristic. In particular, *Decision Tree_0* and



(a) Geomean of instruction reduction for test loops. (b) The ratio of optimized loops compared to *Oracle*.

Figure 2.6: Averaged instruction count reduction and the ratio of optimized loops compared to *Oracle* for each decision model.

Random Forest_0 show reductions of around 21%, which is close to the ideal upper limit. Smaller configurations like *Decision Tree_2*, *Decision Tree_3*, and *Random Forest_1* also show notable improvement. Figure 2.6b illustrates the ratio of loops that are close to the optimal performance. For each test loop, the instruction reduction of each classifier is divided by that of *Oracle*. *Oracle* can not have any negative value in the reduction since it would disable the loop unrolling when the side effect is shown. The calculated ratios are then placed in increasing order to clearly show how much proportion of loops are close to optimal. The points below zero represent the loops suffering from performance degradation due to the wrongly applied loop unrolling. When points are closer to one, corresponding loops are unrolled close to the optimal. In other words, the model with fewer points below zero suffers less from the side effect of loop unrolling while the model with more points close to one has more ideally optimized loops. In both heuristics, a greater number of loops suffer from the side effect of loop unrolling than machine learning techniques. Additionally, they have fewer loops unrolled by the optimal decision. Due to its conservative nature, *SVM* disables more loop unrolling than necessary for many loops. Thus, *SVM* has less loops near one while having many loops on zero. On the other hand, *Decision Tree_0* and *Random Forest_0* are observed to have ideally unrolled loops significantly more than others.

2.6.4 Prediction Overhead Analysis

Figure 2.7 and Table 2.3 illustrate the inference time and memory requirement for each decision model respectively. The memory requirements for main memory (e.g. DRAM) is computed by its entire model size while cache is calculated by its worst-case run-time

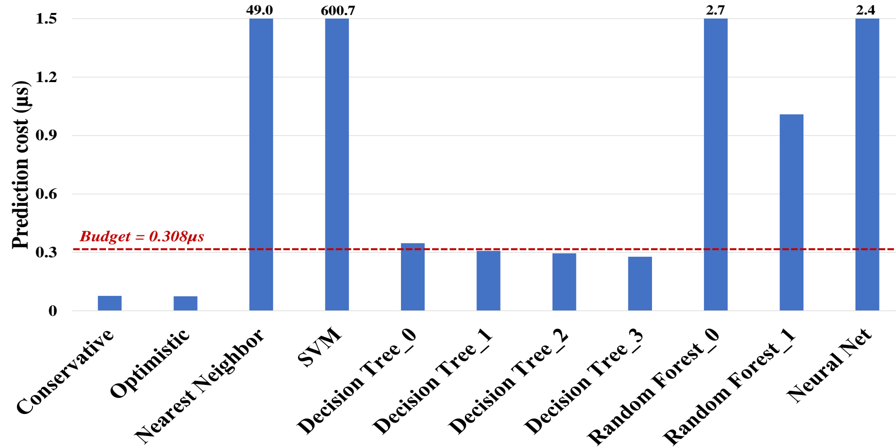


Figure 2.7: Inference time for each model with given budget.

memory footprint for an inference. As *Optimistic* and *Conservative* are essentially the same algorithm with different parameters, their inference times are almost identical. Notably, both inference time and memory requirement of *Nearest Neighbor* and *SVM* are significant. Since *Nearest Neighbor* populates the training data to make a prediction, the data should be stored and accessed during the inference. Thus, the large size of the training data may cause the high prediction overhead. On the other hand, *SVM* makes a prediction by using supporting vectors. Naturally, when the number of supporting vectors is high, the technique would suffer from high memory usage and slow inference speed. In our experiment, $11,028$ supporting vectors are built during training. This approach brings an excessive prediction cost for both memory and inference time ($8,095\times$ slower than heuristics). *Random Forest_0*, *Random Forest_1*, and *Neural Net* present large prediction costs as well. Because the random forest algorithm maintains a group of trees, it requires sufficient memory space to store all tree nodes and computing power to process them to achieve satisfactory prediction speed. The compute-intensive nature of *Neural Net* arises its high prediction cost. Given that a simple network with a single hidden layer is assumed in this experiment, a more complex network is expected to have a higher prediction overhead that would be excessive for our dynamic optimizer. The decision tree model makes a prediction by conducting a sequence of simple tests on the nodes along with the path from the root node to the leaf node. Thus, when the tree has a reasonable height, the model exhibits low prediction overhead.

2.6.5 Choice of Classification Algorithm

Since the optimization time induces a run-time overhead in dynamic optimization, system designers often introduce various budgets on dynamic optimization as the design criteria. Also, due to the limited resource in a mobile processor, there is a restriction on the mem-

Model	Main Memory (KB)	Cache (KB)
Nearest Neighbor	1,391	1,391
SVM	1,500	1,500
Decision Trees	$<$ 130	$<$ 0.66
Random Forests	$<$ 1,122	$<$ 6.96
Neural Net	21	21

Table 2.3: Memory requirement for each model. The requirements for decision tree and random forest configurations are represented by their largest configurations.

ory usage. In this section, we evaluate the feasibility of each classification methods by considering time/memory constraints and identify the best affordable method.

Firstly, we examine the feasibility in terms of time. Based on the rule of thumb widely used by experts in the industry, dynamic binary translation could use no more than $10k * N$ instructions to optimize the region of N instructions. Given that there are plenty of optimization phases and a decision model for the unroll factor is only a small fraction of loop unrolling phase, we assume 0.3% of the total budget as the time constraint for the decision model. Therefore, $30N$ instructions can be used for each unroll factor prediction. To calculate the time budget, we use the information of the processor used for the experiment: Intel Core i7-4700MQ mobile processor (Haswell) with 2.4GHz and its averaged IPC (1.621) [102]. Also, the averaged region size (40) [154, 64] is used for N. As a result, $0.308\,\mu s$ is introduced as the time constraint for each inference and used to evaluate the applicability of each decision model. Figure 2.7 visualizes the inference cost with the time constraint. Overall, all machine learning techniques but the decision tree algorithm present the inference time above the given budget. *Nearest Neighbor* and *SVM* especially show excessively high prediction overhead. Although the previous work [149] suggests that both techniques would be a good choice for unroll factor prediction in a static compiler, their prediction overheads are exorbitant in our dynamic optimizer. Also, the high inference time of *Neural Network*, which is over the budget, implies that even a simple network has excessive computation overhead for our infrastructure. While *Random Forest_0* presents the highest prediction accuracy, it turned out the decision model is not affordable due to its high prediction cost significantly above the time constraint. *Decision Tree_0*, which has the comparable prediction accuracy to *Random Forest_0*, shows faster inference speed compared to the other classifiers. But its prediction cost is slightly above the constraint. The smaller decision trees, *Decision Tree_1* and *Decision Tree_2*, exhibit relatively high prediction accuracy with fast prediction satisfying the given budget. This implies the necessity to consider the trade-off between prediction accuracy and cost in terms of the tree size.

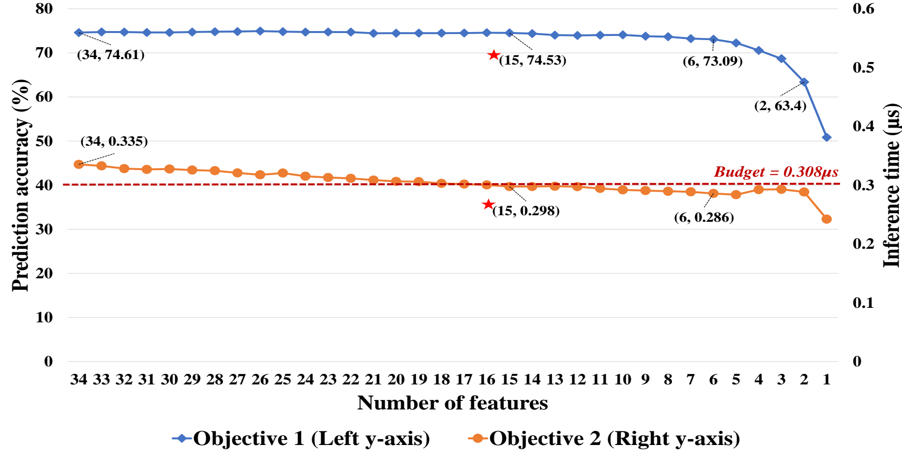


Figure 2.8: The change in prediction accuracy and inference time during greedy feature selection process. With Top 15 features (marked with stars), the decision tree model can show an almost identical level of prediction accuracy to the model with all 34 features while satisfying the time constraint.

We also assume 0.3% of the total memory budget as the memory constraint for the decision model. Table 2.3 represents memory requirements for storing (main memory) and running (cache) each model. Our assumed mobile processor can store all models without difficulty. However, the memory constraint on cache is challenging. Since our mobile processor contains 1 MB of L2 cache, the constraint for cache becomes 3 KB. Thus, only decision tree and random forest with small configuration can meet the restriction.

Overall, the decision tree is the only machine learning based model that satisfies both time/memory budgets on our infrastructure. It is also one of the models with the highest prediction accuracy. Thus, the decision tree would be a good choice as a cost-effective decision model for loop unrolling on our infrastructure. Additionally, the decision tree is an interpretable model which can give insight to system designers. The learned knowledge (e.g., how each feature is treated) from the decision tree can be used to improve the human-made heuristic algorithm.

2.7 Redundant Feature Pruning

To identify the important features and possibly reduce the number of features, we modify the greedy feature selection algorithm [149]. Every iteration, our greedy method drops the least significant feature by examining variation in prediction accuracy when each feature in the feature set is eliminated. Starting from a full feature set (i.e., 34 features in our case), the same process is repeated until no feature is left in the set.

Figure 2.8 presents how our greedy feature selection technique gradually narrows down the feature set for the decision tree algorithm which is identified as the best classification

Top 15 features
General loop property
• The number of operands
• Step of induction variable ✓
• The viability of non-linear loop conversion ✓
Constraints from binary translation
• Proportion of post-optimization blocks compared to its quota ✓
• Proportion of post-optimization instructions compared to its quota ✓
Benefit: opportunity for the following phases
• The number of loop invariants
• The number of loop invariant loads ✓
Side effect: code size
• Number of duplicated exit blocks
• Number of instructions in duplicated exit blocks
Dynamic Information
• Trip count ✓
• The taken probability of side exits
Instruction mix
• The ratio of static stores
• The ratio of memory operations
Smart unrolling
• The size of immediate operand from induction variable ✓
• The size of operand for the loop condition ✓

Table 2.4: Top 15 features for decision tree. Check marks (✓) indicate that the corresponding features are considered in the current heuristic design: 8 out of the top 15 features are employed by current heuristics.

technique for our DBT system in the previous section. The reduced feature set affects tree size and availability of test that the model can perform at each tree node. Mostly, both prediction accuracy and inference time are observed to decrease as the number of available features gets smaller. Since the decision tree makes a prediction at a leaf node by passing through a series of tests along the path, reducing average depth for leaf nodes brings the improvement in the inference time. The prediction accuracy sustains a similar level until only 6 features are employed and drops significantly after that point. Particularly, the decision tree model can present the almost identical level of prediction accuracy with the model employing a full feature set by using only 15 features (less than 0.1% of difference), while satisfying the time constraint. Also, the built model requires 0.61KB of space in cache which fits within our memory budget. Thus, in this work, Top 15 features are recommended to use.

Table 2.4 shows those selected features. Overall, features under various categories

defined in Table 2.1 are chosen. This suggests loop unrolling requires information from diverse aspects to make a good decision. Note, our binary translation puts constraints on upper limits for the expected number of post-optimization instructions and basic blocks. Therefore, the proportion of post-optimization instructions/basic blocks compared to its quota plays an important role in unrolling decision. Top 15 features also include information regarding loop invariants and the number of duplicated instructions/blocks. Each information provides an indication for benefit and side effect from loop unrolling respectively. Dynamic information, such as trip count and the taken probability of side exits, also takes a crucial role by helping the dynamic optimizer to capture the actual iteration count at runtime. On the other hand, the size of the immediate operand from the induction variable and the operand size for the loop condition are meaningful for the decision for smart unrolling. It helps the decision model to estimate the chance of the run-time check failure. The rest of the information helps the optimizer to characterize each loop better.

Interestingly, all 8 features employed in the current heuristic design are chosen as Top 15 features. The significant improvement in the machine learning based model comes from the difference in how each feature is treated (e.g., importance of each feature, threshold for each feature) and the missing features in the heuristic design. By using their own statistical approaches, machine learning based approach is capable of finer tuning in high dimensional space than the hand-made model. Furthermore, the approach can suggest important features that are missed by experts. Our investigation points out that information, such as the taken probability of side exits and the ratio of memory operations, should be considered for the accurate loop unrolling decision. Note, these 7 missing features are readily available or collected with the negligible overhead at the loop unrolling phase.

2.8 Related Works

There have been efforts to build an optimization decision model by hand [163, 143]. Although these results show quite impressive improvements, it is expected to become continuously harder for compiler designers to create an effective model by hand due to the increasing design complexity. The failures with the software pipelining heuristic which is designed to avoid the side effect from its overly aggressive usage are reported [104, 74]. Also, Stephenson and Amarasinghe [149] showed that their baseline compiler predicts optimal unroll factor only 16% of the time. To improve optimization decision, researchers started to employ machine learning techniques. Particularly, given its system-wide impact, loop unrolling is widely studied. Monsifrot et al. proposed an auto-generation of heuristics for a target processor by using the decision tree algorithm [110]. They built a binary classifier that decides whether to apply loop unrolling or not while leaving unroll factor determination to

the existing heuristic. To include the unroll factor in the automation process, Stephenson and Amarasinghe [149] suggested to consider this problem as a multi-class classification problem and solved it by introducing machine learning techniques. Their best classifier shows 65% of prediction accuracy which leads to a 5% speedup (software pipelining disabled) and 1% speedup (software pipelining enabled) over the SPEC 2000 benchmark suite. While the macroscopic approach is similar, our work assumes a dynamic binary translation in the mobile system which is more challenging environment than a static compiler assumed in previous works. As the prediction cost occurs in run-time overhead, five different machine learning techniques are evaluated with varying configurations to identify the cost-efficient machine learning algorithm and show the relation between classifier size and prediction accuracy. Leather et al. suggested the automatic feature generation mechanism for the decision tree model designed to predict the unroll factor [97]. They define the feature space by a grammar and automatized the exploration by using genetic programming and predictive modeling. This approach can be applied to our work to isolate the best set of features for the machine learning-based decision model.

The application of machine learning techniques is also explored for the dynamic compilation. In a Java Just-in-time (JIT) compiler, Cavazos and Moss [50] improved the program speed by employing supervised learning to predict whether blocks would benefit from instruction scheduling optimization. For the blocks expected to gain no advantage from the instruction scheduling, their approach bypasses the optimization to improve the compilation time at runtime. Cavazos and O'Boyle [51] proposed an automatic tuning method for function inlining in a Java JIT compiler. They designed a genetic algorithm that searches a large space of parameter values efficiently. In addition, the best optimization configuration is explored in JIT compiler. Hoste et al. [83] proposed the multi-objective evolutionary search algorithm to find Pareto-optimal in terms of compilation time and execution speed for JIT compiler. As a result, they gained up to 40% of improvement in compilation time and 19% of improvement in steady-state performance over the default setting of Jikes RVM. On the other hand, and O'Boyle [52] used logistic regression to determine which optimization should be applied to each method based on its features in Jikes RVM and achieves 29% of speedup compared to -O2 optimization level.

The comprehensive survey in the field of machine-learning based compilation is well described in the work by Wang and O'Boyle [156].

2.9 Conclusion

To improve the optimization decision model in binary translation, we propose a statistical and automatic approach by employing machine learning techniques. Focusing on loop

unrolling, our work examines the performance and feasibility of machine learning models. We evaluate our approach with the industrial strength infrastructure and 17,116 unrollable loops collected from various real-life programs and benchmarks. By considering both prediction accuracy and cost, the decision tree model is identified as the best classification model. Then, through the greedy feature selection method, its significant features are discovered. By using them, we successfully build the effective best affordable decision model that satisfies the given time/memory budgets and greatly outperforms the baseline heuristics by making better optimization decisions.

CHAPTER 3

SRTuner: Effective Compiler Optimization Customization By Exposing Synergistic Relations

3.1 Introduction

Increasing complexity in both hardware and software exacerbates difficulties in performance analysis (e.g., estimating the impact of a compiler optimization). As a result, the analysis often relies on overly simplified assumptions to capture system characteristics [149, 70, 121]. Therefore, even the most advanced compilers frequently fail in delivering best optimization settings for individual applications and leave substantial performance on the table [70, 124, 34, 119, 82]. As the end of Moore's Law slows processor innovation down, leveraging this latent opportunity is becoming more critical [78].

Although the opportunity seems obvious, maximizing the benefit from existing compiler optimizations is an ongoing problem that has not been solved for over 50 years [42, 89], and is still a highly challenging problem owing to the following several key observations:

First, the best optimization setting for one application may not be optimal for others [70, 34]. Even for the same application, the effect of each optimization largely depends on the underlying environment (e.g., target hardware, compiler) [69, 70]. To attack these challenges, various tuning methods have been studied to customize the optimization setting [32, 120, 42]. To tune any given workload, the tuning process often launches from scratch without any *prior* knowledge about the running environment, such as hardware cost model or previous tuning history, and navigates the optimization space by only utilizing the feedback acquired during tuning. This approach is called a *pure search* [49] (e.g., genetic search [83, 68]). Pure search is particularly important since it lays the groundwork for other customization approaches. Section 3.2 discusses further details.

Second, traversing a significant fraction of entire optimization space is practically

impossible. Modern compilers use over 100 settings at their highest optimization levels leading to more than 2^{100} possible combinations even when simply assuming binary (e.g., enabled/disabled) values for each. Thus, tuning methods typically examine a limited number of promising settings within the given budget (e.g., number of trials). To navigate extremely large search space effectively, state-of-the-arts adopt an iterative feedback-directed search where promising candidates are generated, evaluated, and reflected upon to then generate even more promising candidates in future trials [32, 56]. An intelligent tuning method should identify promising subspaces and prune the worthless search space by effectively utilizing feedback from its earlier trials.

Third, the optimization space is highly non-linear and contains numerous irregularly distributed local optima [42]. A tuning process may achieve low-hanging fruit when it wastes too much tuning budgets on local optima. Thus, it is crucial to escape local optima efficiently by balancing *exploration* (i.e., choosing a setting for the next trial randomly in the unknown space) and *exploitation* (i.e., choosing a setting for the next trial that is expected to be beneficial based on the past trials) to maximize the tuning quality. In general, exploration-only searches may waste an opportunity to safely achieve benefits while exploitation-only searches may be trapped at local optima. State-of-the-arts tuning methods suggest various approaches to handle exploration-exploitation dilemma (Section 3.2), yet there is still room for improvement.

Lastly, we note that the lack of proper understanding for compiler optimizations is another source of major issues in finding their best use. Given that state-of-the-art compilers apply hundreds of optimizations, it is challenging to understand how every optimization works and how to use it with other optimizations. This makes manual design of optimization heuristic labour-intensive and error-prone even for experts [121]. Thus, prior tuning methods often ignore or overly simplify inter-optimization relations [124, 119] since they can be complicated to analyze even for compiler experts. In reality, optimization decisions are deeply related, so without proper consideration of their relationships, these approaches are ignoring information that could be used to better guide the tuning process and improve overall quality.

As a breakthrough, we propose *SRTuner*, a novel pure search tuning method that estimates the impact of individual optimizations and their relationships, which allow it to identify promising search subspaces effectively. To generate a good setting for each trial, SRTuner configures optimizations one at a time via our multistage structure: each stage chooses the most encouraging configuration (e.g., on/off, parameter) of its corresponding optimization. As each optimization can make its decision on the basis of optimization decisions configured on the earlier stages, SRTuner can naturally consider the relation

between optimizations.

Based on our observation that configuring high-impact optimization actually helps later optimization decisions, we map each optimization to a stage in the order of its estimated impact. Despite continuous efforts, prior methods [124, 119, 57] have critical limitations in approximating an optimization impact, such as inability to consider various inter-optimization relations. To overcome such restrictions, we suggest a novel distribution-based estimation method. When examining performance of settings (i.e., combination of optimization decisions), the high-impact optimization would show clear separation in its performance distribution depending on its configuration. Thus, we measure the extent of the separation between distributions and consider it as the optimization impact.

As balancing between exploration and exploitation is key to maximize the tuning quality, we define an optimization decision problem as a multi-armed bandit problem to formulate the exploration-exploitation dilemma. Then, a tuning process becomes a game of playing a series of bandits, which allows the adaptive optimization decision at each stage in our multistage structure to deal with local optima efficiently.

Due to our unique tuning strategy, SRTuner can uncover important synergistic optimizations and provide this information to users at no extra cost. Section 3.6 showcases the utility of understanding such synergistic relations in improving future optimization design and customization methods.

The contributions of this chapter are as follows:

- Our work proposes an intelligent pure search technique, called SRTuner. We evaluate our strategy with three representative compiler stacks on CPUs/GPUs and demonstrate that SRTuner can extract $1.24\times$, $2.03\times$, and $34.4\times$ averaged speedups over each compiler's highest optimization level and also outperform state-of-the-arts.

- We suggest a novel methodology to estimate optimization impact in the presence of inter-optimization relations. Also, we design multistage structure that allows SRTuner to make adaptive optimization decisions by considering interactions between optimizations.

- SRTuner identifies important synergistic optimizations as a byproduct of its search at no additional cost. We discuss how knowledge of a workload's synergistic optimizations can be used to help future studies and compiler design.

3.2 Related Works

Largely, tuning methods can be categorized into two classes: (1) Workload-specific tuning: customizes optimization setting for any given workload by investing tuning time. Pure

search [90, 68, 119] is the representative approach. (2) Generalization method: learns from the previous tuning experience to predict the optimization setting for unseen workload without paying a separate tuning time [70, 34].

As workload-specific tuning usually requires long tuning time, generalization methods have been proposed to recommend optimizations across different inputs, programs or architectures. One of the most popular approaches is providing a few fixed sets of optimization settings manually tuned by compiler experts as standard optimization level. However, as these one-for-all approaches cannot handle the increasing complexity of computing system anymore, learning-based generalization methods [70, 34, 120] are proposed. With supervised learning, generalization methods recommend promising settings by capturing the relationship between representative workloads and their tuned settings in the training data [70, 49, 34]. Various studies regarding the performance portability of OpenCL programs across different hardwares [125, 148, 151] also give insights for generalization methods.

However, their potential is fundamentally restricted by the quality of training data. High quality of training would describe each workload accurately by using right features providing the exact information that matters for optimization decisions and contain the fine-tuned optimization setting [70, 34]. To collect the best possible settings for representative cases in training data, generalization models inevitably depend on the pure search and thus, the improvement in pure search has a particular significance. Section 3.6.2 discusses how our technique can benefit generalization model design.

By using the only information acquired during the tuning process, pure search methods traverse the optimization space in the feedback-directed manner. Depending on its strategy, pure search can be categorized as follows:

Exploration-oriented Without utilizing any feedback, an Random search (*RAND*) allows unbiased traversal in the search space. Due to its simplicity, generalization methods often adopt random search to generate their training data [70, 34].

Exploitation-oriented Pure search can adapt its tuning process by using feedback from previous trials [119, 68]. Franke et al. [68] proposed a strategy that modifies PBIL [38], which integrates genetic search and competitive learning, for the compiler optimization problem. We denote this as *mPBIL*.

Ensemble search To design *balanced* pure search, researchers have proposed ensemble techniques that manage multiple search methods with different strategies. As a sophisticated ensemble method, Ansel et al. [32] suggested *OpenTuner* that runs a collection of search algorithms with advanced trial distribution policy. During expedition, algorithms which perform well will be assigned more trials.

On the other hand, tuning methods usually have high evaluation cost due to the repetitive

evaluations. To resolve these issues, an idea of learning hardware cost model at tuning time is proposed [56, 28]. These model-based approaches periodically trains the hardware cost model with the machine learning method by using evaluation data accumulated during a tuning process. Then, the constructed cost model would be used by a search strategy to select the most promising settings that will be evaluated on the real hardware. As a core search method, *XGB tuner*, a model-based approach employed by TVM, adopts an ensemble pure search using parallel simulated annealing and random search [56]. On the other hand, beam search is chosen for model-based approach in Halide [28].

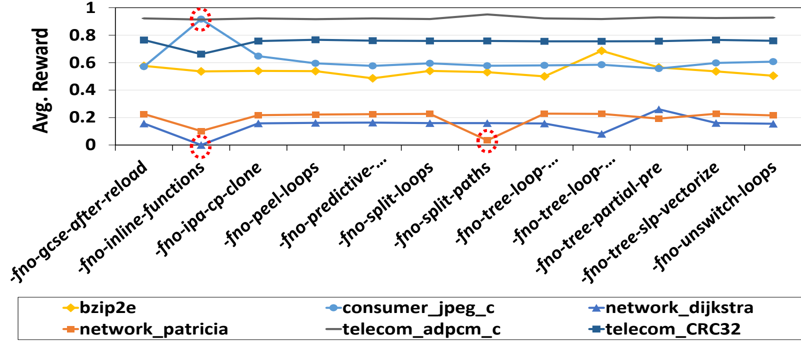
Domain-specific knowledge can be useful to narrow the search space effectively [129]. However, as its space is still too extensive, SRTuner can be also adopted for the effective search. Once SRTuner customizes effective optimization settings, users may also consider finding optimal phase ordering [93, 96, 95, 60]. The comprehensive survey for auto-tuning domain is well described in the work by Ashouri et al. [33].

3.3 Design Motivation

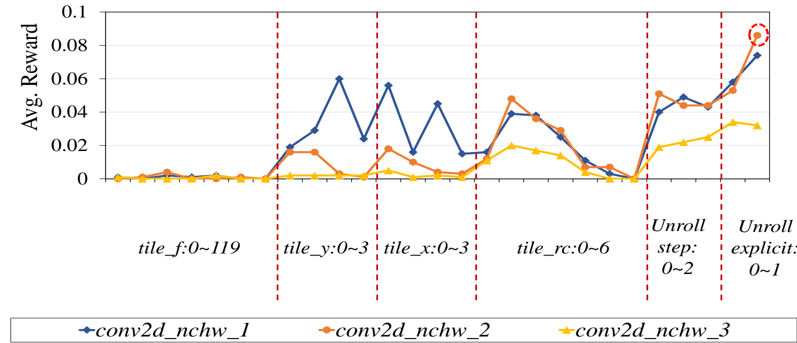
Due to the extremely large size of optimization space, an intelligent pure search should determine promising subspaces efficiently to return the best output within the given tuning budget. To tackle this challenge, we propose a novel search method that narrows down the search space by effectively determining the most promising configuration. This section describes the insights that motivate our design.

For the motivational experiment, we conduct statistical analysis by using 2,000 different flag combinations randomly sampled in the optimization space of two representative compilers from two different domains respectively: GCC (traditional C/C++ compiler) and TVM (machine learning compiler) [55]. 12 binary flags under *-O3* are investigated for GCC over representative CPU programs in cBench benchmark suite [71] [1]. Note that each flag follows a form of *-fno-⟨optimization⟩*: when a flag is turned on, it will disable the corresponding optimization. Thus, if no flags were turned on, we would be running *-O3*. For TVM, we chose three convolution kernels in real-life deep learning workloads (e.g., resnet-18). These kernels perform the same computation but have different incoming/outgoing tensor shapes. Compared to GCC, TVM targets the GPU and mostly handles fewer but highly parameterized flags. We denote each configuration in a form of *⟨optimization⟩: ⟨ordinal number of a configuration⟩*. To assign relevant feedback for the performance of each setting, we introduce a concept of reward. In this motivational experiment, we simply award ”+1” whenever a combination beats the fastest default setting (e.g., *-O3* for GCC). Otherwise,

[1] A collection of realistic benchmarks assembled by cTuning foundation [18] for the research on program and architecture optimization.



(a) GCC: Traditional C/C++ compiler



(b) TVM: Machine learning compiler

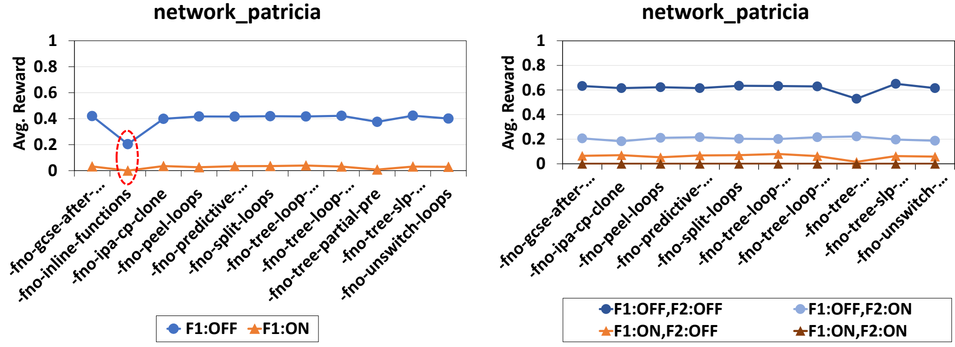
Figure 3.1: Motivational experiment with two representative compilers. A reward (+1) is assigned whenever a combination outperforms the baseline setting. By sampling 2,000 settings, averaged reward for configuring each flag is examined.

"+0" is assigned. In this case, the averaged reward implies the probability of generating a better combination than the baseline.

3.3.1 Multistage Structure

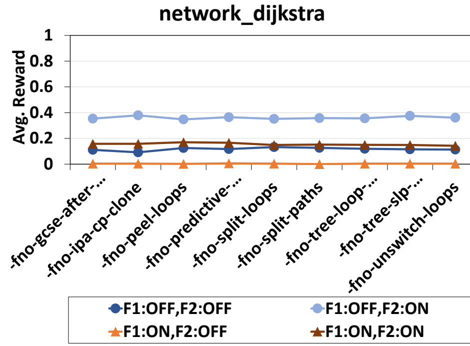
Figure 3.1 illustrates the averaged reward when each optimization flag is configured for target programs. A high average reward above 0.7 is observed in *telecom_adpcm_c* and *telecom_CRC32* implying that -O3 is quite out-of-tune on these benchmarks. In contrast, the baseline setting becomes harder to outperform in *network_dijkstra* and *network_patricia*. Figure 3.1b shows lower average rewards on TVM because of their highly parameterized optimizations.

We further see that optimizations can have drastically different effects depending on the workload (red circles). For instance, *-fno-inline-functions* is highly likely to be beneficial in *consumer_jpeg_c*, but likely to degrade performance in *network_dijkstra*. *-fno-split-paths* seems to slow down the execution in *network_patricia*. TVM also presents a similar outcome as in Figure 3.1b. Notably, *unroll_explicit: 1* (red circle) seems helpful in *conv2d_nchw_2* while any of *tile_f* does not. In these cases, workloads have an affinity to certain optimiza-

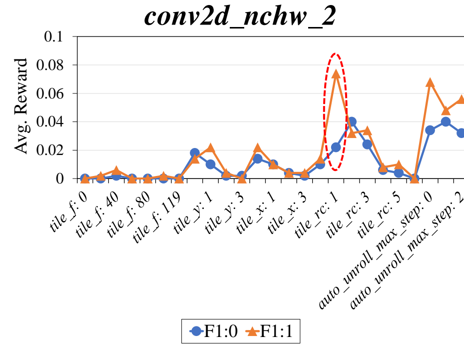


(a) *-fno-split-paths (F1)*

(b) Expose synergy between *-fno-split-paths (F1)* and *-fno-inline-functions (F2)*



(c) Expose conflict between *-fno-tree-partial-pre (F1)* and *-fno-tree-loop-vectorize (F2)*



(d) *unroll_explicit (F1)*

Figure 3.2: Averaged reward given extra condition: (a)-(c) for GCC, (d) for TVM. Starting from Figure 3.1, the high-impact flag is configured for each benchmark. (a) shows the average reward after a single stage. When more than one flags are given, the relation between optimizations can be unveiled. (b) and (c) present the positive/negative interaction between two given optimizations, respectively. (d) demonstrates the same approach can be applied to TVM as well.

tions, and configuration decisions are obvious; however, in general, inextricable interactions between various optimizations complicate the decision process.

Figure 3.2 depicts the averaged reward in typical benchmarks when high-impact optimization flags have already been configured. Once again, a reward of "+1" is assigned when beating the fastest default setting and "+0" is assigned otherwise. In this context, the reward is the conditional probability to outperform the default setting when the value for another optimization flag is predetermined. By conditioning on the configuration of a significant flag, we can unveil its important hidden interactions with other flags. When examining Figure 3.1a, configuring *-fno-split-paths* has a significant impact in *network_patricia* and thus, it is configured first. Figure 3.2a illustrates how the probability of generating promising combinations can increase notably by turning off *-fno-split-paths*.

Then, conditioned on *-fno-split-paths* being off, we find that *-fno-inline-functions* impacts the average reward the most (red circle). Since the average reward now has the highest sensitivity to *-fno-inline-functions*, we configure this flag in the second stage. Figure 3.2b shows the positive synergy between *-fno-split-paths* and *-fno-inline-functions*. Since *split-paths* helps increase the optimization window by splitting paths leading to backedges, *inline-function* can cooperatively broaden the optimization window and benefit latter optimizations like *dead code elimination* and *common subexpression elimination*. Therefore, it seems better not to provide the *-fno-split-paths* and *-fno-inline-functions* flags during compilation so we can exploit those optimizations.

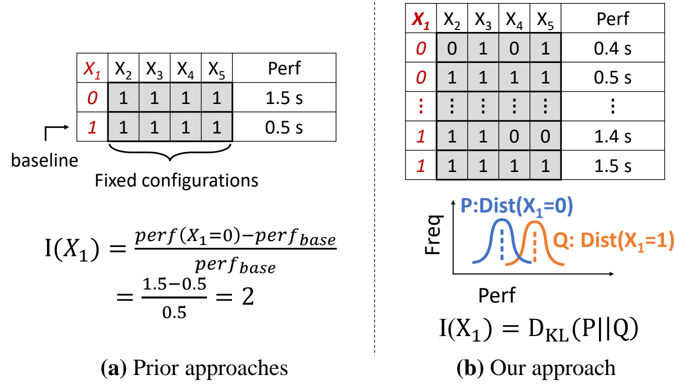
By applying the same methodology, Figure 3.2c exposes the negative interaction between *-fno-tree-partial-pre* and *-fno-tree-loop-vectorize* in *network_dijkstra*. It suggests that aggressive partial redundancy elimination negatively affects loop vectorization in this benchmark. Figure 3.2d presents the results with TVM. By configuring a significant flag correctly, we can focus on the promising subspace and expose the subtle interaction between two loop transformations, loop unrolling and tiling: *unroll_explicit* and *tile_rc* (red circle).

By walking through the outlined examples, we showed that promising search subspaces and interactions between optimizations can be revealed by configuring the highest impact optimizations first. The approach can be iteratively applied until all configurations are determined. This observation gave us the intuition to design a multistage optimization process where flag values are chosen one-by-one.

3.3.2 Optimization Impact Estimation

Up until now, we have assumed high-impact optimizations are given in order to motivate our multistage design, but it is also necessary to measure the magnitude of a flag's performance impact. Prior approaches estimate an optimization's significance by fixing configurations for the remaining optimizations and measuring the performance difference between the resulting code with the optimization and the one without it. Figure 3.3a showcases the methodology. Since they only consider a fixed configuration of the other optimizations from numerous possible combinations, the interactions between optimizations are ignored [124, 119]. Furthermore, such techniques cannot handle execution failures (e.g., compile error) since the estimation equation requires every run to provide valid performance numbers [119]. Various *Design of Experiments* methods also suffer from the same restriction [124, 57].

To overcome these limitations, we propose a statistical alternative. When estimating the impact of an optimization, we let the configurations of the others vary and examine how the resulting performance distribution is affected by the optimization decision (Figure 3.3b). By examining the distribution rather than two scalar performance numbers, the approach



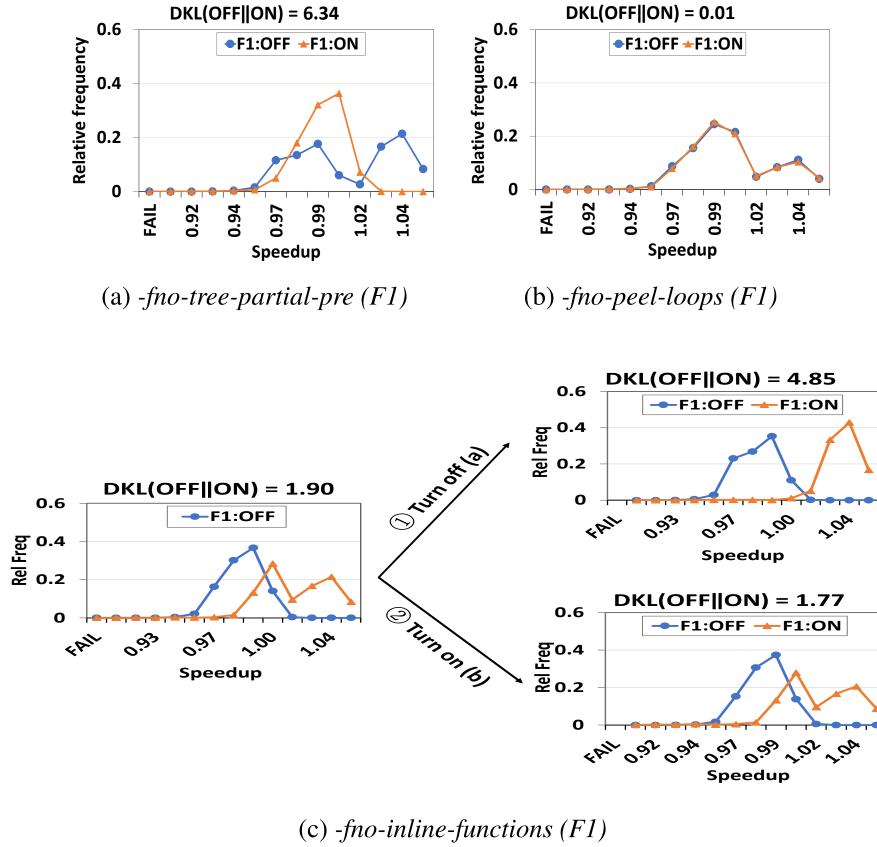
(a) Prior approaches **(b)** Our approach

Figure 3.3: Comparison between prior impact estimation method and our distribution-based method. To approximate an optimization impact, prior approaches ignore inter-optimization relations and measure the performance difference on the single fixed configurations of the rest. However, our method estimates based on the performance distribution to consider interactions with diverse configurations of others.

can handle execution failures as well. The approach is based on the observation that a significant optimization with high impact shows a clear separation between the performance distributions for its potential configurations. Figure 3.4 demonstrates our observation with GCC. Each graph presents a discrete performance distribution illustrated by a histogram of speedup w.r.t. the baseline performance, *-O3*, when a certain optimization is configured. Figure 3.4a and Figure 3.4b contrast the impact of an optimization on the performance distribution. Figure 3.4a shows that the configuration of *-fno-tree-partial-pre* has a significant impact on the performance distribution while Figure 3.4b presents the marginal impact of *-fno-peel-loops*. Based on this observation, we approximate the impact of an optimization by measuring the degree of separation between distributions of configurations. To quantify the extent of separation between two distributions P and Q, we adopt the following equation of Kullback–Leibler divergence ($\mathrm{D_{KL}}$) [105]:

$$D_{KL}(P||Q) = \sum P(x) \log \frac{P(x)}{Q(x)} \tag{3.1}$$

Figure 3.4c justifies the effectiveness of our idea of configuring optimization in the order of impact: the right configuration of the significant flag helps the separation of the performance distribution of the next flag by exposing their hidden interaction (①). Conversely, when the non-significant flag is configured first, it may disrupt the later optimization decision (②) as we can also confirm with $\mathrm{D_{KL}}$.



(a) *-fno-tree-partial-pre (F1)* (b) *-fno-peel-loops (F1)*

(c) *-fno-inline-functions (F1)*

Figure 3.4: Performance distributions and their KL divergence (D_{KL}) in *consumer_jpeg_c* given a certain GCC optimization. Relative frequency presents an occurrence normalized by the entire sample counts. (a) suggests to turn off the flag by showing a higher relative frequency of outperforming the baseline (i.e., speedup above 1.0) when disabled. (c) illustrates how a decision of earlier optimization affects latter optimization decision. Left graph presents performance distribution when there is no other optimizations configured and right two graphs display how the performance distribution changes when (a) or (b) are configured, respectively.

3.3.3 Handling Local Optima

Our motivational experiments demonstrate that configuring high-impact optimizations one-by-one actually uncovers hidden inter-optimization relations and greatly enhances the chance of finding good settings. However, it does not necessarily guarantee the discovery of the best setting. When such an approach falls into a local optimum, it may result in wasting trials in the area with low-hanging fruits. Therefore, it is important to visit multiple promising regions to maximize the chance of getting closer to global optimum. Thus, we formulate the exploration-exploitation dilemma by introducing the concept of the multi-armed bandit problem. At each stage, our multistage algorithm configures an optimization by weighing the expected benefits from exploration and exploitation to narrow down the search space

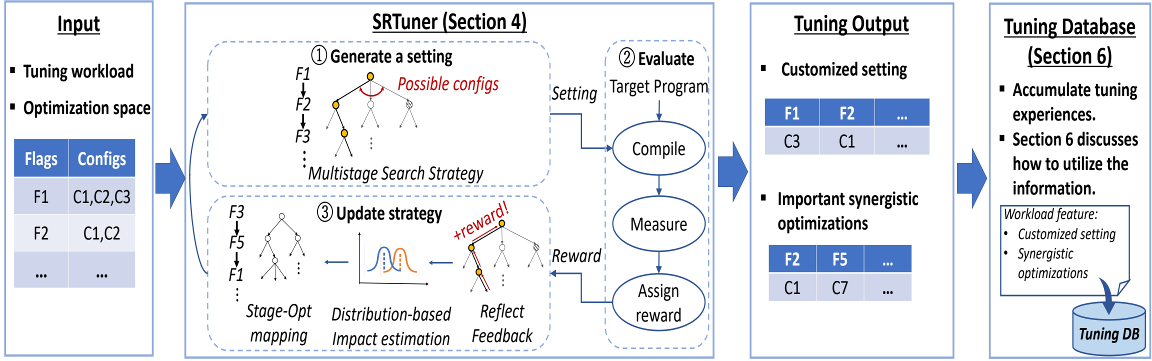


Figure 3.5: SRTuner overview

intelligently.

3.4 SRTuner

Figure 5.3 illustrates the overarching design of our tuning strategy. As our technique does not depend on the underlying environment, any compiler, hardware, or performance metric can be adopted. In this work, we focus on minimizing the execution time. Once the optimization space is defined, the search process begins for the tuning workload. First, SRTuner creates a promising combination via multistage generation algorithm (①). The framework then applies the corresponding optimization setting to the target program and measures its run-time performance. Based on the reward policy, reward will be computed accordingly (②). Once an evaluation is done, SRTuner adjusts its strategy by using feedback and re-map each optimization onto each stage by estimating the optimization impact (③). Within the given tuning budget, these steps will be repeated to extract the best result. Since search algorithm (①, ③) is entirely orthogonal to the rest of framework, it can be easily plugged-in to the existing tuning frameworks, such as a library tuner [116] or model-based approach [56], or ensemble method [32]. As a result of tuning, SRTuner outputs a fully customized setting as well as important synergistic relations among them. Optionally, every tuning experience can be accumulated in the tuning database for the future uses. Section 3.6 demonstrates how these data can benefit future optimization decisions.

To effectively balance between exploration and exploitation, we design a multistage process of creating an optimization combination. At each stage, the process configures an optimization by considering each decision as the multi-armed bandit problem [53, 47]. Thus, a process of generating a combination becomes a game playing series of multi-armed bandits. To formulate the exploration-exploitation dilemma for each bandit, *Upper Confidence Bound for Trees (UCT)* [91] is adopted. After N trials playing a K-armed bandit, UCT will pick the most promising arm $I \in \{1, ..., K\}$ as follows:

$$I = \underset{i \in \{1 \ldots K\}}{\arg\max} \{ \frac{r_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}} \} \tag{3.2}$$

where n_i, r_i represent the number of trials and the accumulated rewards for choosing i^{th} arm. c is the exploration constant. The first term of the equation describes the averaged reward from the previous trials which implies the expectation towards the exploitation. Thus, this term will have a large value when previous choices for the arm were rewarding. The second term represents the component for exploration that will have a bigger value when the arm has been less chosen. By adding up two components, UCT balances exploration and exploitation. Note, the accumulated reward r_i will be amortized by n_i. Thus, UCT will revert to exploration when the exploitation-oriented trials continuously fail in collecting rewards. In this metaphor, each bandit would represent an optimization which has arms as many as the number of possible configurations for the corresponding optimization. Naturally, our algorithm can handle options which have more than two configurations and quantify the exploration-exploitation trade-off by using Equation 3.2. Figure 3.6 illustrates our multistage structure that reflects our idea. The structure consists of nodes and edges arranged across the multiple stages. Each stage represents a single multi-armed bandit that configures an optimization. A node depicts an intermediate combination built from previous stages. It has as many outgoing edges as the number of configurations that the corresponding optimization supports. Thus, an edge represents a configuration of each optimization on the top of an intermediate combination of earlier optimizations. For an adaptive decision with Equation 3.2, each node maintains information about the number of visits and the total rewards achieved during trials. For implementation, we considered Monte-Carlo Tree Search (MCTS) [46]. However, our experiment showed that naive application of MCTS performs worse than random search. To make MCTS effective in our problem space, we adapted MCTS and also developed two policies that determine how to adjust strategy based on the feedback from previous trials: reward policy (Section 3.4.1) and stage-optimization mapping policy (Section 3.4.2).

Figure 3.7 shows an example of how SRTuner works on the structure. Each trial consists of three sequential phases: ① *Generate* ② *Evaluate* ③ *Update*. A trial starts with a generation phase creating an optimization combination to test. By applying Equation 3.2 to the information logged at each node, SRTuner makes an adaptive decision at each stage by balancing between exploration and exploitation. When a decision leads to a node that has not been visited before, new nodes will be randomly created until a complete combination is forged at a terminal node. Note, conventional MCTS only creates a single new node in the unknown space and conducts simulated play-out without further expansion. Given that the

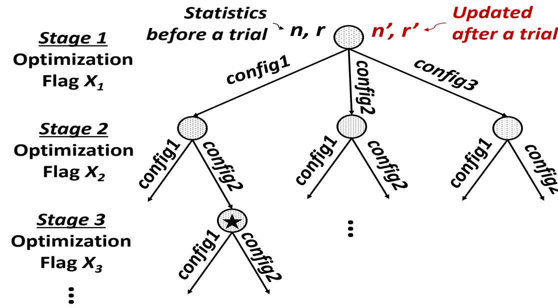


Figure 3.6: An example of multistage structure used in SRTuner. Each stage configures an optimization and each node represents a combination forged at earlier stages. For example, the annotated node ($\star$) will configure X_3 when *config1* and *config2* are chosen for X_1 and X_2, respectively. Each node counts the number of visits, n, and accumulates rewards, r.

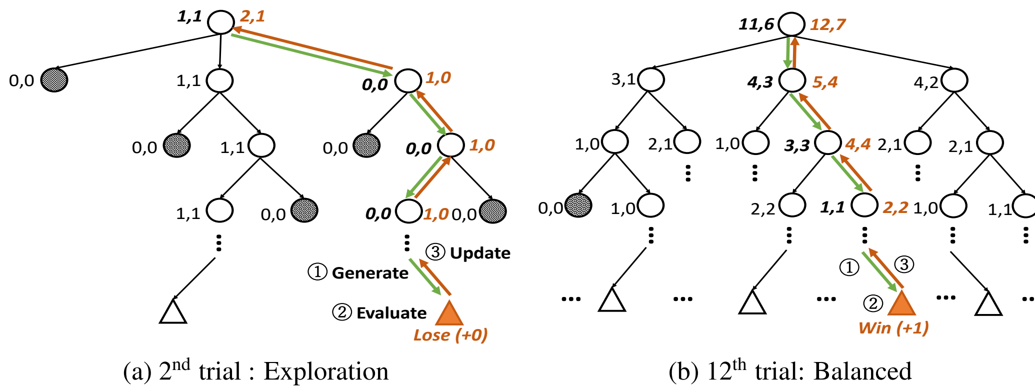
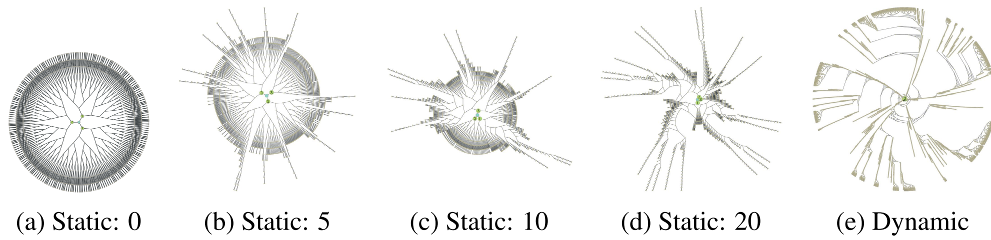


Figure 3.7: An example of SRTuner strategy. White nodes represent expanded nodes while black nodes imply topmost nodes in the unknown areas. +1 reward is applied when a trial outperforms the baseline. As tuning proceeds, SRTuner naturally exposes popular paths on the structure, which show the constructive relations between optimization decisions.

tuning budget is extremely limited compared to the size of the search space, our algorithm adopts a more aggressive approach. Later, we show that this also helps our optimization impact estimation for stage-optimization mapping. Once a combination is forged, SRTuner performs the evaluation phase that computes reward based on the actual performance of the corresponding setting. During the update phase, new findings, such as a reward, will be backpropagated to the nodes contributed during the generation phase and update their information. Figure 3.7a illustrates an example of early trials. Due to the lack of information to leverage, the search starts with random exploration. When the information is accumulated above a certain amount, algorithm estimates benefits from exploration and exploitation to choose the expectant configuration as in Figure 3.7b. As tuning proceeds, SRTuner naturally identifies synergistic configurations by examining the frequently chosen decision on each



(a) Static: 0 (b) Static: 5 (c) Static: 10 (d) Static: 20 (e) Dynamic

Figure 3.8: Visualization of different reward policies

Algorithm 1 *Reward policy*

Constants

LB : Lower bound margin.

C : Constant multiplication factor.

```
1: procedure GETREWARD(perf)
2:     reward ← 0
3:     ratio = (perf_best - perf)/perf_best
4:     if ratio > LB then
5:         reward ← max(C * (1 + ratio), 0)
6:     return reward
```

stage and utilize them to focus on the promising subspace.

3.4.1 Reward Policy

In Equation 3.2, a higher reward makes the search focusing on its subspace longer. Since excessively high reward may lead the search algorithm to be stuck at the local optima, it is crucial to have an intelligent reward policy. Figure 3.8 visualizes the effect of reward policy. [2] Without any reward, the search will traverse every direction uniformly as in Figure 3.8a. When a constant reward is awarded for settings beating the baseline, the algorithm can exploit the past experiences and narrow down its search space (Figure 3.8b). With a larger reward, the algorithm strongly invests more trials on the focused areas (Figure 3.8c-3.8d). However, when the constant reward is blindly awarded, these policies may guide search to the local optima. Ideally, to efficiently handle local optima, we want to assign high reward on the new observation of the best performance while lowering the reward when observation of the similar performance becomes frequent at later trials. Therefore, as in Figure 3.8e, a dynamic reward policy is designed to dynamically adjust its focus intensity in proportion to the information value of the performance observation.

Algorithm 1 outlines our reward policy. To find the best possible setting rather than many

[2] Expanded nodes (i.e., white nodes in Figure 3.7) are placed in the radial layout and non-diverging paths are omitted for simplicity.

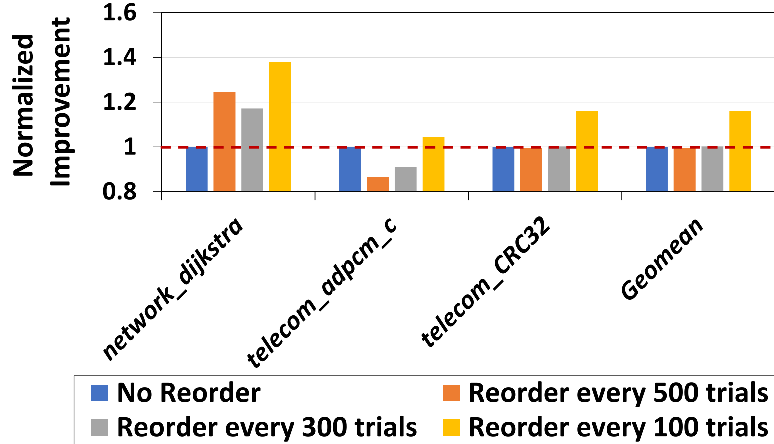


Figure 3.9: Analysis of reordering frequencies. Each performance is normalized by the one without remapping.

mediocre settings, our policy only encourages a combination that shows the competitive performance compared to the best-performance-so-far and dynamically adjusts the amount of reward according to its information value. In our implementation, past rewards are not re-computed at the update of $perf_{best}$ since influence of previous rewards will be gradually decayed as tuning proceeds.

3.4.2 Stage-Optimization Mapping

The observation in Figure 3.4 suggests us configuring high-impact optimization can help latter optimization decisions and thus enhance the quality of combination generation. Also, since the diverging path makes a node to have more visits than nodes in its subtree in our multistage structure, it would be more effective to locate important optimizations in the earlier stages so that they can make a better decision with more samples. However, since impact of an optimization is highly application/hardware-dependent, the ordering of optimizations cannot be statically determined. This motivates us to design reordering mechanism that periodically re-maps each optimization to a stage based on our distribution-based impact estimation method. With samples accumulated during the tuning process, optimization impact can be approximated by measuring separation between distributions as in Section 3.3.2.

To measure the impact of an optimization, we group run-time observations into bins to describe a discrete distribution for each configuration. Naturally, when more samples are available, more accurate estimation can be made. Because KL divergence is not symmetric, we refer each distribution of a configuration in a certain order. For a parametric optimization, two configurations presenting the maximum/minimum averaged performance are chosen for impact estimation. Unlike conventional MCTS, when the node expansion is necessary,

Algorithm 2 *Collect important synergistic configurations*

```
1: procedure EXTRACTSYNERGY(root, tuned)
2:     cur_node ← root
3:     syn_group ← ∅
4:     while cur_node do
5:         min_num ← inf
6:         chosen ← null
7:         cur_opt ← cur_node.optimization
8:         for all child ∈ cur_node.children do
9:             min_num ← min(min_num, child.num)
10:            if tuned[cur_opt] == child.config then
11:                chosen ← child
12:        if IsBiased(chosen.num, min_num) then
13:            syn_group.append(tuned[cur_opt])
14:        cur_node ← chosen
15:    return syn_group
```

SRTuner keeps expanding new nodes until reaching the terminal node. This allows the equal number of samples for each optimization for its impact estimation.

Figure 3.9 presents the effectiveness of our stage-optimization mapping policy. When remapping is performed every 100 trials, SRTuner can find a $1.16\times$ better performing setting than the tuning without remapping.

3.4.3 Extract Important Synergistic Relations

Due to our unique tuning strategy, SRTuner can suggest important synergistic relations. By examining the optimization decisions in tuned setting, SRTuner collects biased paths on the multistage structure that are expected to be more promising than others. Since each choice is made based on decisions in the earlier stages, naturally, it would expose synergistic relations between optimizations. Algorithm 2 illustrates our approach. By controlling thresholds for bias detection, user can change the confidence level of identified synergistic relations.

3.5 Performance Evaluation

3.5.1 Experimental Setup

For evaluation, we implemented SRTuner as a portable Python package and examined the acceleration from each auto-tuning strategy compared to the fastest setting provided in three representative compiler stacks from various domains on different target hardwares as in Table 3.1. While an independent tuning framework is built for GCC, SRTuner is inserted as a search strategy into existing tuning frameworks for TVM and OpenCL compilers. In total, 3 standard optimization levels (i.e., *-O1*, *-O2*, *-O3*) and 100 optimization flags in GCC-7

Compiler	Search Space	Backend	Benchmark	Implementation
GCC (traditional C/C++)	10^{30} (100 flags)	CPU	cBench [71]	Standalone
TVM (deep learning)	10^{7} (5-8 flags)	GPU	Deep Learning Workloads	Plug-in
OpenCL compiler (heterogeneous computing)	10^{5} (15 flags)	CPU/GPU	BLAS library [117]	Plug-in

Table 3.1: Experiment setup

are examined [3] On the other hand, TVM and OpenCL compilers provide several highly parameterized optimization flags. Although their search space is narrower than GCC, it is still too extensive to traverse without an intelligent strategy.

Once a tuning method customizes a setting, it is assessed by the averaged performance of 30 run-time measurements. The tuning quality of each method is evaluated by aggregating results of five runs where equal budgets are allocated for each run. By referencing prior studies [70, 56], we consider 1,000 trials are affordable and tried to maximize the tuning quality within this budget. To show the quality across tuning runs, max/min improvements are also presented. Usually, a tuning process took several hours and none of tuning methods is allowed to use more tuning time than others during experiment.

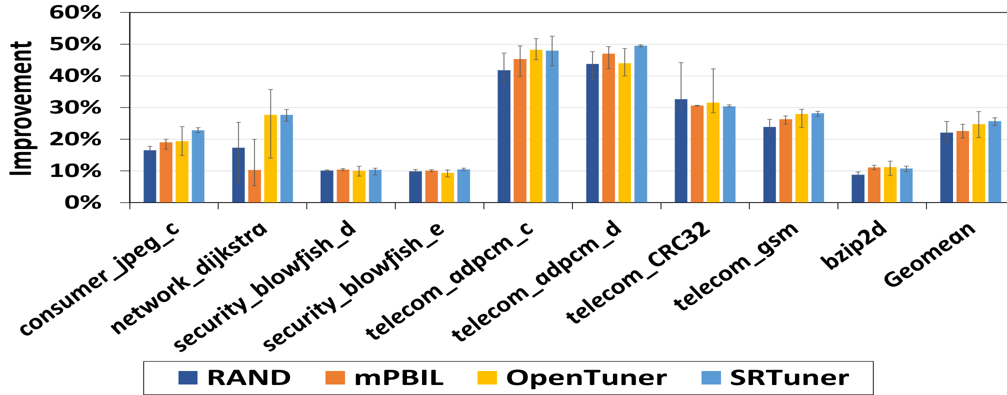
3.5.2 GCC

Figure 3.10 presents the results with GCC over programs in cBench [71], a suite of representative auto-tuning workloads [4]. For comparison, we chose well-known tuning methods designed for GCC: RAND [69], mPBIL [68], OpenTuner [32].

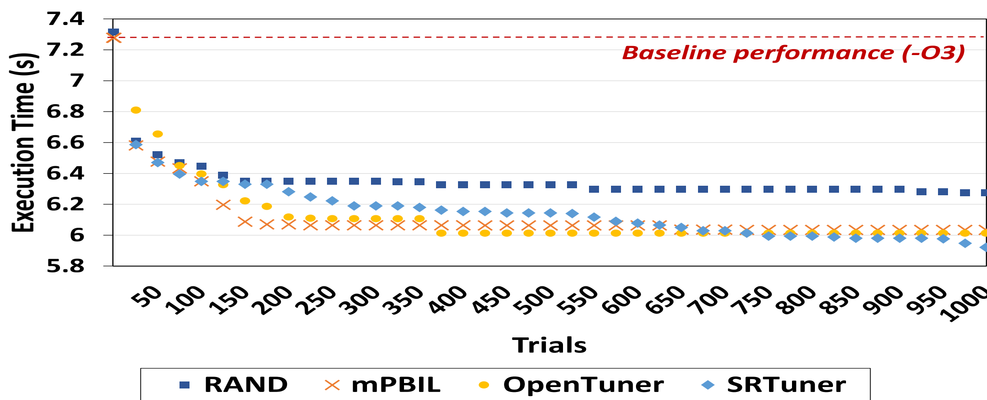
Figure 3.10a showcases the improvements from the tuning methods compared to *-O3*. Overall, balanced search methods (i.e., OpenTuner, SRTuner) outperform exploitation-oriented (i.e., mPBIL) and exploration-oriented (i.e., RAND) search methods by handling the exploration-exploitation dilemma. On average, RAND and mPBIL present similar improvement around 20% although each method performs better than each other at different benchmarks. With a sophisticated ensemble method with a pool of unique search methods, OpenTuner achieves further enhancement by showing 23.4%. However, unlike previous balanced methods with ensemble approach, SRTuner handles exploration-exploitation dilemma as a solo method and beats all prior tuning approaches. Given its extremely large size and the control-flow intensive property of C/C++ benchmarks, GCC has the most challenging optimization space among the compilers we investigated. Nonetheless, SRTuner can traverse the GCC optimization space efficiently and successfully accelerate the execution $1.244\times$

[3] Due to a slight difference in GCC distributions [19], the supported flags are confirmed by using *-fverbose-asm*.

[4] A few benchmarks that none of tuning methods brought 10% speedup are excluded for better visualization.



(a) Overall tuning result



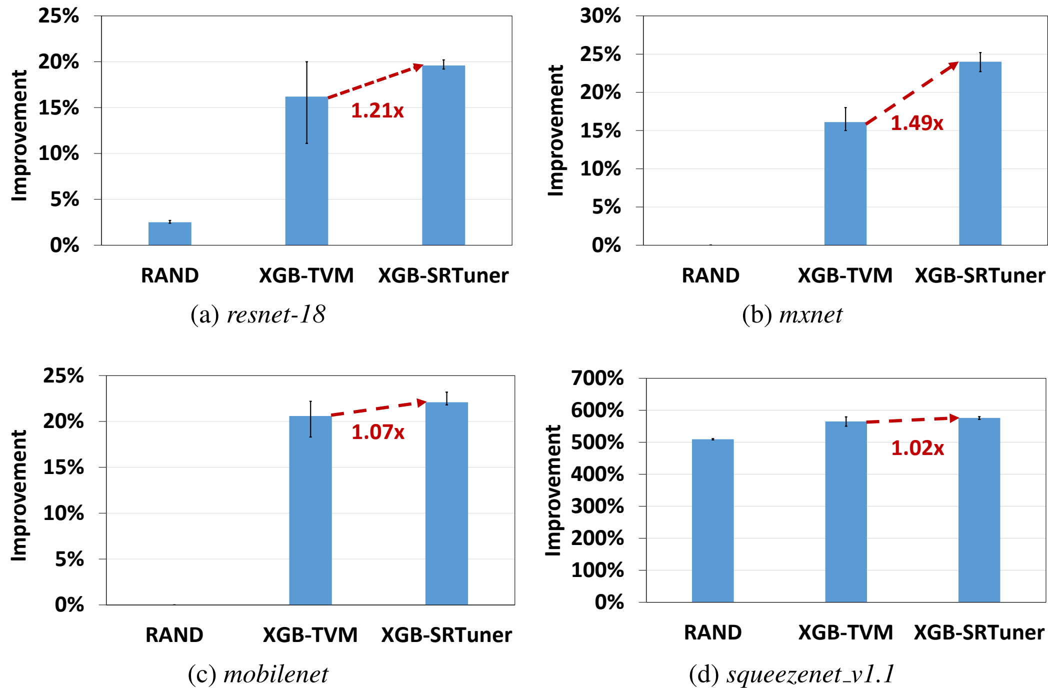
(b) Tuning progress on *consumer_jpeg_c*

Figure 3.10: Tuning C/C++ applications with GCC on Intel Xeon E5-2430.

faster. Especially, our method outperforms state-of-the-arts greatly on the applications like *consumer_jpeg_c* and telecom_adpcm_d. As we focus on delivering reliable tuning quality, SRTuner also shows the most stable improvements across the tuning runs.

Essentially, tuning is a process of correcting optimization misuses in the context of a given workload. Thus, due to the nature of tuning, there are some benchmarks without significant winners. It is unavoidable that for some programs default setting (-O3) may be already close to global optimum. In such occasions, any tuning process would converge to the similar performance gain. *security_blowfish_d* and *security_blowfish_e* might be the case. For applications like *network_dijkstra*, *telecom_adpcm_c*, and *bzip2d*, there can be numerous local optima with similar performance that may require many trials to escape. Thus, although OpenTuner and SRTuner extract further speedup than others, they may be stuck at the similar local optima and need more tuning budget to escape. Oftentimes, effective use of feedback from the previous trials is very hard. In such cases, RAND performs the best as in *telecom_CRC32*.

Figure 3.10b illustrates a representative tuning progress. While other techniques waste



(a) *resnet-18* (b) *mxnet* (c) *mobilenet* (d) *squeezenet_v1.1*

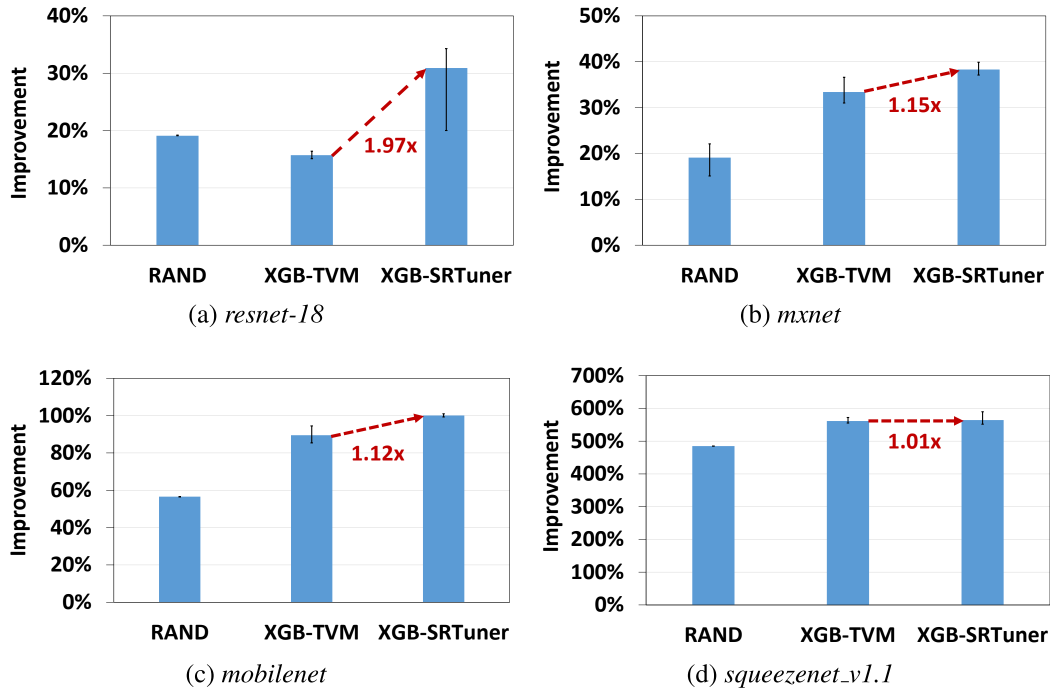
Figure 3.11: Tuning deep learning workloads with TVM on NVIDIA RTX 2080Ti.

lots of trials at the local optima, SRTuner shows more efficient escape from local optima and finds the better setting.

3.5.3 TVM

For efficient tuning, AutoTVM provides a model-based approach with the ensemble search. We denote this as *XGB-TVM* (See Section 3.2). For evaluation, SRTuner was plugged into this model-based approach as its pure search strategy and referred as *XGB-SRTuner*. Overall, four real-life deep learning workloads are examined on two GPUs: NVIDIA GeForce RTX 2080Ti and NVIDIA Quadro K620.

Figure 3.11-3.12 present the improvement from each tuning method compared to a default setting which is determined for the running environment by referencing the TVM database. Overall, SRTuner can extract substantial speedup across all workloads and devices we investigated. Since the landscape of optimization space greatly varies across different kernels and hardware devices, a default setting is hard to perform well unless its database covers the exact same case. Since RTX 2080 Ti has closer data point in the database than Quadro K620, optimization space on RTX 2080 Ti is more challenging to extract performance improvement, which often causes RAND fails as in Figure 3.11b-3.11c. However, our strategy can extract around 23% of enhancement. By effectively handling the exploration-exploitation dilemma, SRTuner outperforms both the balanced strategy of XGB-TVM and RAND. Especially, results in Figure 3.11a-3.11b, 3.12a-3.12c



(a) *resnet-18* (b) *mxnet*

(c) *mobilenet* (d) *squeezenet_v1.1*

Figure 3.12: Tuning deep learning workloads with TVM on NVIDIA Quadro K620.

are considerable.

3.5.4 OpenCL Compilers

CLTune [117] provides tuning capability for OpenCL BLAS kernels with three different strategies: RAND [69], Annealing [117] and Particle Swarm Optimization (PSO) [117]. Figure 3.13-3.14 present the tuning result on CPU and GPU, respectively. Across all benchmarks and hardware we examined, auto-tuning shows the most significant enhancement: SRTuner achieved $68.0\times$ on CPU and $17.1\times$ on GPU, respectively. Since OpenCL kernel supports a variety of hardware backends with totally different design philosophy, each optimization should be configured according to the target device. Thus, one-for-all approach misses the chance to realize substantially better performance. Overall, SRTuner identifies better customized setting than prior methods by escaping local optima efficiently.

3.6 Discussion

This section discusses potential ways for scientists and engineers to utilize information about synergistic optimizations to better current systems.

3.6.1 Identification Of Important Misuses

Fundamentally, tuning is a process of correcting optimization misuses in the context of a given workload. By finding optimization decisions that disagree with GCC among important

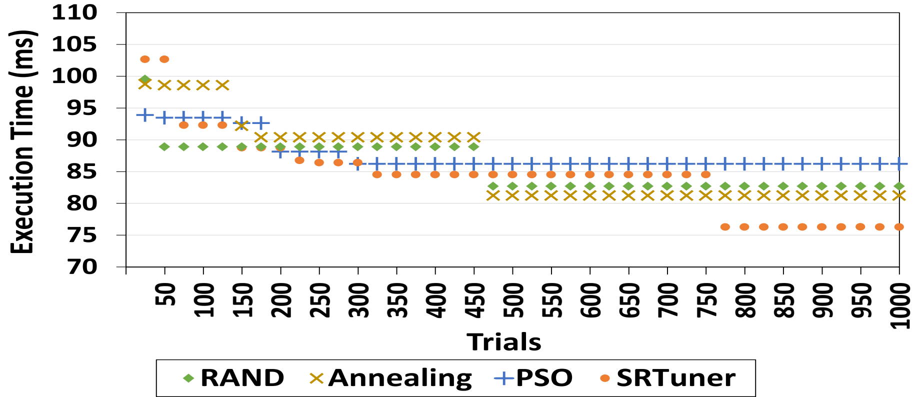


Figure 3.13: Tuning *gemm* with OpenCL on Intel i7-9700K.

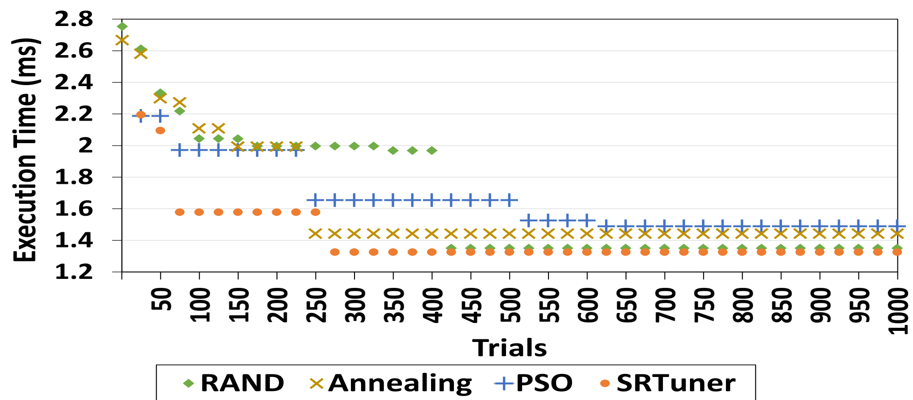


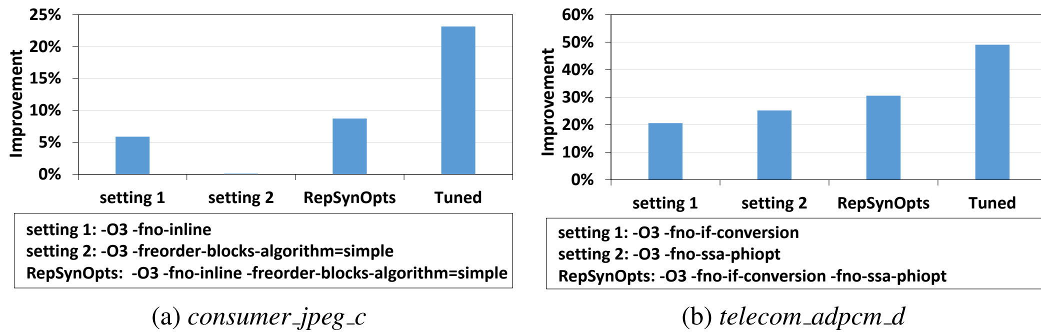
Figure 3.14: Tuning *gemm* with OpenCL on NVIDIA RTX 2080Ti.

synergistic optimizations uncovered by , we could identify high-impact optimizations that were configured sub-optimally for the given workload.

Figure 3.15 exhibits the performance consequences of GCC misusing the high-impact optimizations found with . We selectively enable each flag with a baseline setting of -*O3* and observe its impact. For reference, we also display the improvement from using a representative subset of the synergistic flags and the performance of the fully tuned workload.

Figure 3.15a shows that *consumer_jpeg_c* suffered from the side effects of excessive inlining. Therefore, we could extract 5% speedup by simply disabling inlining. When manually configuring the basic block reordering algorithm to the *simple* heuristic from *software trace cache*, which is the default for *-O2 and -O3*, we saw no notable change. However, when the *simple* heuristic is used when inlining is disabled (third bar), it can provide an additional 3% improvement over *-fno-inline*. This demonstrates the positive relationship between two optimization configurations identified by .

Figure 3.15b shows the serious misuses of if-conversion and phi-node optimization in



(a) *consumer_jpeg_c* (b) *telecom_adpcm_d*

Figure 3.15: Examine representative subset of synergistic optimizations revealed by .

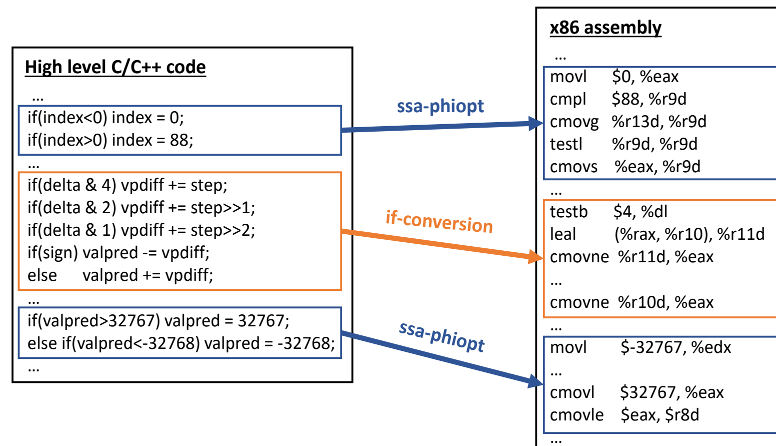


Figure 3.16: Hot code region in *telecom_adpcm_d* where if-conversion and phi-node optimizations are seriously misapplied by GCC.

-O3. These are basic optimizations in GCC and are both configured in the low standard optimization levels (i.e, *-O2* and *-O1* respectively). To optimize code with conditionals, the phi-node optimization pass conducts various code transformations such as load hoisting on diamond patterns and along with other pattern-based if-conversions [5]. By disabling each optimization individually, we could immediately achieve more than 20% speedup. Due to their synergistic relationship, disabling both together improves performance by an additional 5%. To better understand the source of the performance gains, we examined hot functions from *telecom_adpcm_d* and found the code region where these optimizations are making wrong decisions. Figure 3.16 illustrates how those two optimizations transform the corresponding region into x86 assembly. To optimize the conditionals, both optimizations blindly convert control dependence with if-blocks into data dependence with conditional instructions. Although the optimization window can be broadened by eliminating control

[5]For example, it finds min/max/abs patterns and replaces conditional blocks with conditional instructions like cmov.

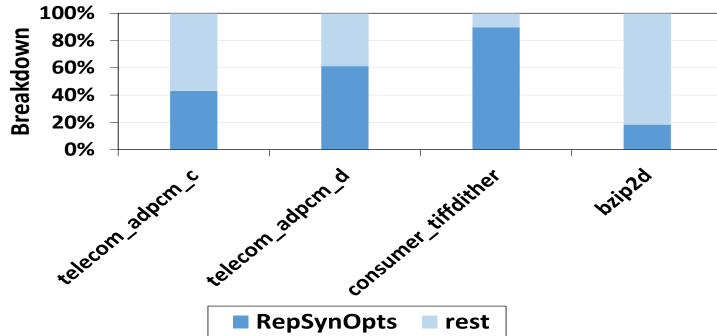


Figure 3.17: Diverse applications with *-fno-if-conversion* and *-fno-ssa-phiopt* in their synergistic optimizations. Breakdown shows their importance in each customized setting.

Model	Architecture	SM/SMM	Shading Units	L1 Cache
NVIDIA RTX 2080Ti	Turing	68	4352 (64/SM)	64KB/SM
NVIDIA Quadro K620	Maxwell	3	384 (128/SMM)	64KB/SMM

Table 3.2: GPU specification. We consider RTX 2080 Ti and Quadro K620 as high-end and low-end devices respectively.

flow, in this case, there is not much opportunity for latter optimizations (e.g., *instruction scheduling*) because of the data dependency on *vpdiff*. Therefore, the program reaps no benefit from eliminating control dependencies, while paying the cost of executing all control flow paths with conditional instructions.

To correct misuses, there are two viable approaches: (1) Improve optimization design. Currently, configuring *ssa-phiopt* applies various pattern-based optimizations all together or not at all. Although its design may be intuitive for compiler designers, it may provide sub-optimal performance since various sub-optimizations with different benefits/side effects (e.g., load hoisting and if-conversion) will be enabled/disabled by a single optimization decision. By separating sub-optimizations into different optimization passes depending on their effects, we could selectively disable the troublesome sub-optimization (e.g., min/max replacement) while applying others within *ssa-phiopt* without any performance penalty. (2) Build a decision model that recommends settings tailored for each workload. Next section motivates our future direction.

3.6.2 Generalization Of Tuning Experiences

Although our technique can significantly accelerate a program's execution, tuning programs is time-consuming and thus, we cannot expect to tune every workload. To make auto-tuning more practical while still providing high performance, our future work will investigate generalization methods [70, 34], which learn a pattern between good optimization settings and workload features from prior tuning experiences. Ideally, a high-quality training data will

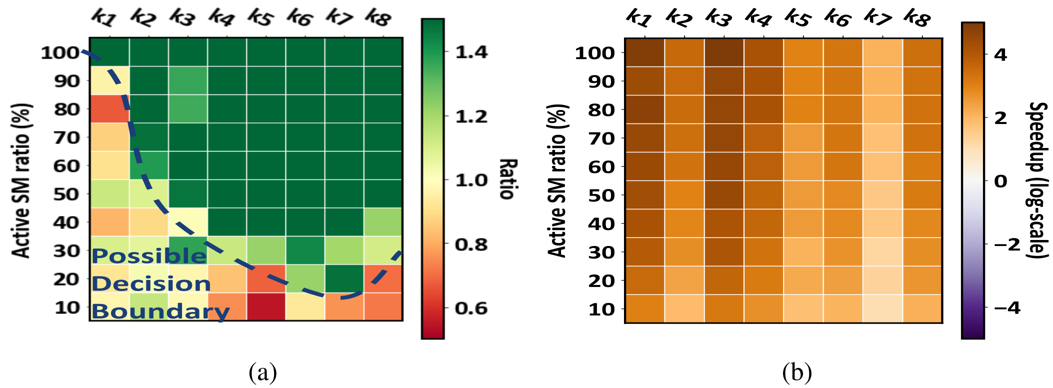


Figure 3.18: Experiment with eight different kernels (*k1-8*) and diverse SM activation ratios. (a) Performance ratio between high-end and low-end settings. Green and red cells imply cases where high-end and low-end settings should be applied respectively. Blue line represents a possible decision boundary. (b) Speed-up when choosing one of high-end and low-end settings in the right places. Speed-up is computed against the performance of default setting with 100% SMs.

be generated by tuning representative small programs with to tailor generalization models that predict promising optimization settings for big unseen programs. We believe would be helpful when developing a generalization method for two main reasons: (1) can provide high-quality optimization settings for a generalization method to learn from. (2) Synergistic relations can provide key insights for designing a generalization model. When constructing a generalization model, it is important to understand which applications are similar from an optimization perspective. In general, this is known to be extremely challenging. However, we believe that a workload's synergistic optimizations could be used as a way to group these seemingly dissimilar programs. Figure 3.17 presents diverse applications that exhibited the same synergistic optimizations. Breakdown shows how much those synergistic optimizations contributed to the fully customized performance. Although those benchmarks seem quite different from a programmer's perspective, this suggests that they share certain program properties which makes them similar from an optimization perspective. If the property can be identified, compiler experts could characterize each workload more accurately by including it as part of the program features, allowing the generalization model to learn more effectively [70, 34]. On the other hand, to avoid difficulty of manual feature selection, compiler designers may consider using automatic feature generation techniques [97, 98]. Essentially, these methods train a model to capture the pattern between workload features and their optimization decisions by adopting supervise learning. Thus, by grouping programs at the optimization perspective by using our synergistic relations, we believe we can do better than manual approach in generating high-quality training data even for such automatic

feature generation models.

Instead of focusing on only program similarities, we could cluster hardware devices that respond well to the same optimizations. For experiment, we tuned optimization settings in TVM for eight different kernels from real-life deep learning workloads. For each kernel, we acquired a tuned setting for a high-end device and a low-end device (See Table 3.2). To explore how different hardware configurations would respond to these two sets of optimization settings, we simulate various hardware devices by controlling the number of active Stream Multiprocessors (SMs) inside the RTX 2080Ti via Multi-Process Service (MPS) support [20] [6]. In this experiment, a lower active SM ratio is akin to a low-end device while a high ratio would imply a high-end device. It is worth noting that even when 10% of SMs are activated, it would imply a higher-end device than the Quadro K620 (i.e. 6 SMs with better technology including enhanced shading units, caches, and etc.). For each simulated hardware, we examine which customized setting is faster (high/low-end) and if the default setting can be outperformed. Figure 3.18 illustrates the results. Figure 3.18a showcases the pattern that a generalization model can learn via training (i.e., blue line suggests one possible decision boundary). In general, the high-end setting favors the hardware configurations with sufficient resources (green cells) while the low-end setting can be useful when resources are limited (red cells). However, each kernel shows a different transition point. This suggests that generalization requires careful consideration for both application and hardware properties to characterize each workload accurately. Figure 3.18b presents the speedup that an ideal generalization model at different resource budgets could achieve over the default settings at 100% SMs. For every case, the ideal generalization model is observed to outperform the baseline. Interestingly, even with significantly less resources, it was possible to outperform the default setting with full resources.

3.7 Conclusion

To find the best use of existing optimizations, we introduce a novel auto-tuning method, called SRTuner that effectively exposes inter-optimization relations and uses them directly to identify promising subspace. Our evaluation demonstrates that SRTuner can provide competitive performance compared to prior tuning approaches as well as useful information regarding optimization relations. We also discuss use-case scenarios how to utilize this information.

[6] We also tried targeting different architectures with nvcc, but no notable change was observed.

CHAPTER 4

Collage: Auto-tuning Deep Learning Execution Plans With Diverse Backends

4.1 Introduction

Due to the explosive popularity of Deep Learning (DL) applications, there are tremendous demands for powerful software/hardware stacks that allow fast and efficient execution of DL workloads. This strongly drives both industry and academia to invest significant amount of efforts in developing various cutting-edge hardware devices [2, 1, 7] , optimized libraries [6, 5] , inference engines [8] , etc. Naturally, both hardware and software stacks are diversified and result in a giant DL ecosystem.

Within this complex ecosystem, users have various choices of backends [1] to generate efficient binaries of their DL workloads on target device and the right use of each backend is a key to maximize their performance. Users may lower DL operator primitives to low-level implementations by using vendor libraries (e.g., cuDNN [6], cuBLAS [5], MIOpen [86]). Alternatively, users may choose to leverage optimized operators automatically generated by TVM or offload their entire network to inference engines (e.g., TensorRT [8]).

To deal with diverse backend choices, existing DL frameworks, such as Tensorflow [10], PyTorch [9] and TVM [55], provide an execution plan hand-written by system experts. By leveraging their insights and manual analysis, these frameworks offer the rule-based compilation strategy. In general, these rules try to offload entire workload to the vendor-provided inference engine, if available. Otherwise, they utilize the optimized libraries based on their fixed priorities for each operator. For example, cuDNN would have the highest priority for convolution while cuBLAS would be the first choice for matrix multiplication.

However, our observation demonstrates that current state-of-the-arts may leave substantial performance on the table. We found many occasions where the inference engine does not

[1] We define a backend as a software library or inference engine enabling efficient execution of DL workloads on target device.

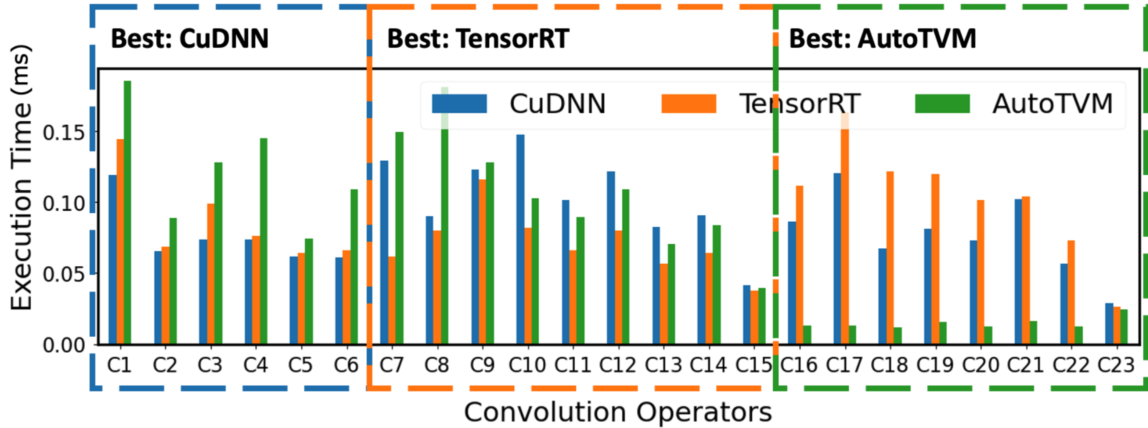


Figure 4.1: Performance of various convolution operators in ResNext-50 on NVIDIA RTX 2070; Note that there is no single backend that always delivers the fastest implementation.

provide the fastest execution. Even within the same type of operator, the most performant backend varies depending on the underlying hardware and the input configuration of an operator (e.g., input tensor shape, padding, etc.). Figure 4.1 showcases our observation. Due to their unique strength and weakness, the best backend choice can be drastically different depending on the configuration of convolution. Interestingly, the competitive performance of AutoTVM indicates that the vendor library/inference engine may not be the best choice on many cases. Furthermore, given that the DL ecosystem is evolving fast, the landscape of backend performance can continuously change. As such analysis result can be easily outdated, it is inevitable to repeatedly tune the execution strategy and this makes the current manual tuning approach inefficient and error-prone in terms of the performance maintainability.

To overcome the current limitation, we aim to design an intelligent auto-tuning method that customizes the best use of diverse backends for the given DL workload and underlying hardware device. Key challenges are as follows:

First, it is difficult to accurately describe the full capability of a library or inference engine. In practice, different libraries and inference engines have their own unique capability and coverage. For example, optimized libraries, such as cuDNN and cuBLAS, provide efficient low-level implementations for DL operators while inference engine, such as TensorRT, takes entire network and creates the network-wide best execution plan with cross-kernel optimizations. Even within these libraries, each library would cover different set of operators: cuBLAS mainly offers matrix multiplication kernels while cuDNN supports efficient implementation of important DL operators, such as convolution. Also, each backend may have its unique way of fusing multiple operators into a single kernel. Beyond a few enumerated patterns of fused operators, advanced fusion engines [55, 6, 115]

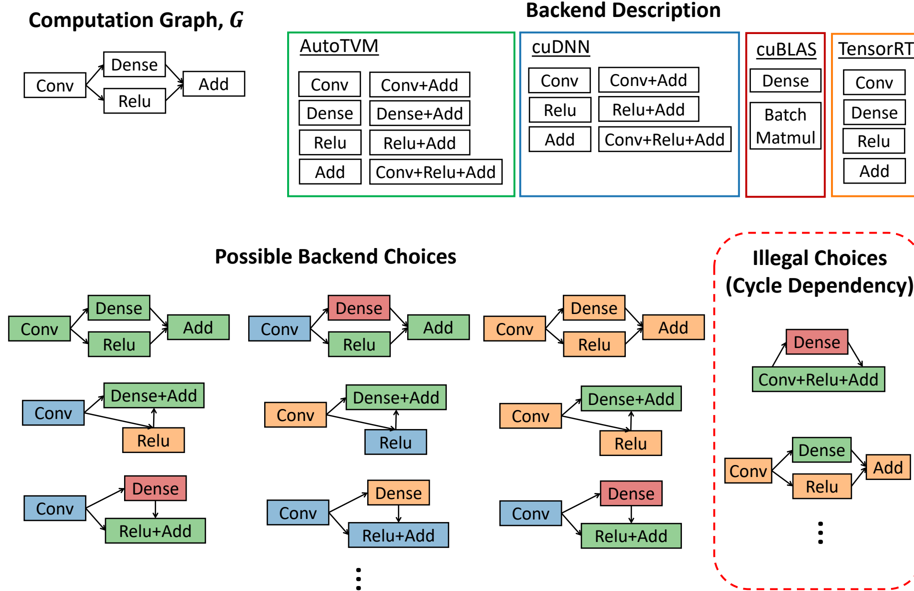


Figure 4.2: An example of possible execution strategies for a simple workload with four different backends. This illustrates the explosive size of the search space and the difficulty in the efficient exploration. To assign backend, both graph topology and the supported backend pattern should be matched. In case of lowering to fused operators or offloading partial graph to inference engines, matched operators should not have any cyclic data dependency to avoid deadlock.

follow complicated fusion rules to allow more flexible application of fusion. Thus, naive pattern enumeration would not be able to capture full capability of advanced fusion engines anymore.

Second, search space is extremely large and complicated. Although we simply assume two backends for a network of 100 operators, the possible number of possible mappings would be 2^{100} which is practically impossible to examine even its significant fraction. In reality, there are more number of backend choices and bigger size of networks. Furthermore, the search space traversal is difficult since both graph topology of the network and capability/coverage of each backend should be considered together. Figure 4.2 demonstrates this challenge with a toy example. Even for a small network, it is not easy to navigate the possible candidates and identify the best strategy.

As a breakthrough, we propose a novel auto-tuning system, called *Collage*, that efficiently searches for the best use of diverse backends and generates the near-optimal execution strategy customized for the given DL workload and underlying hardware device. For the accurate description of various backends, *Collage* provides the straightforward user interface that allows users to specify supported patterns of operators via two ways: users may explicitly enumerate patterns for simple cases or bring the pattern rules to implicitly generate more complicated legitimate patterns for the workload with our pattern generator.

Once backend description is ready, our system explores the possible matches with those patterns on the computation graph of the workload to identify how to partition the graph and assign the most performant backend on each subgraph with the consideration for the underlying environment (e.g., hardware device, tool chain, etc.).

To overcome excessively large search space and the difficulty of graph topology, *Collage* propose a two-level approach that can efficiently navigate promising candidates. As optimized libraries (e.g., cuDNN) offer operator-level point of view while inference engines (e.g., TensorRT) are designed to provide graph-wide optimizations (i.e., cross-kernel optimizations), we take two different search strategies to deal with their differences. Our first-level explores the promising candidates at the operator-level perspective, without the consideration for the effect from graph-wide optimizations. Without cross-kernel optimizations, the tuning problem becomes a problem of how to lower operators on the computation graph into a set of independent low-level kernels from available backends. As fused operators would pack multiple operators into a single kernel, this problem should consider operator fusion as well. To enable the intelligent navigation, *Collage* defines a search as a Dynamic Programming (DP) problem based on the additive cost model while taking into account the topology of the computation graph and backend patterns. This cost model also enables the cheap evaluation of a promising backend placement by reflecting factors, such as driver overhead. With this approach, our first-level tuning method can efficiently customize the best multi-backend execution strategy only within a few seconds. Then, based on the decision from the first-level, our second-level applies fine-tuning by identifying which parts of the graph would benefit from such graph-wide optimizations to fully leverage the capability of inference engine. For the efficient search, we represent a state with a sequence of digits and navigate its space with the off-the-shelf genetic algorithm [67]. Because of the challenge in incorporating the graph-wide optimizations for the efficient cost model design, our current approach evaluates each sample with the actual measurement which leads to the longer tuning time compared to the first level. Thus, for now, we recommend the graph-level tuning as the optional step for the users who are interested in squeezing the last drop of performance.

All in all, our contribution is as follows:

- We propose a tuning system, called *Collage*, that automatically customizes the best hybrid execution strategy with diverse backends for the given DL workloads and hardware device. Our evaluation demonstrate *Collage* could not only enhance the performance $1.18\times$ speedup compared to the best backend choice for each workload, but also outperform the hand-written approach in the state-of-the-arts framework by $1.31\times$.

- To fully leverage the capability of each backend, *Collage* provides the seamless user interface that allows easy and accurate adoption of diverse optimized libraries with their sophisticated fusion engines and inference engines. Our pattern generator enables efficient descriptions of supported patterns beyond naive enumeration.
- *Collage* suggests two-level approach to efficiently navigate the search space with diverse backends of different characteristics. By ignoring the cross-kernel optimizations from the inference engines, our first level tuner with the dynamic programming method could customize the effective plan within a few seconds. To handle the potential loss from the first level, we introduce the second tuner to fine-tune the plan with evolutionary search.

4.2 Related Work

Diversified Backends To extract the best performance from the underlying hardware, there have been substantial efforts to design powerful backends. Hardware vendors have released various specialized optimized libraries and inference engines. NVIDIA has actively developed cuDNN [6] to deliver optimized implementations of DL operators, cuBLAS [5] to offer efficient BLAS kernels, and TensorRT [8] to create the fast execution plan for DL workloads by considering graph-wise optimizations. To eliminate the overhead from the plan generation within the inference engine, TensorRT allow users to cache the generated plan. Meanwhile, Intel has released oneDNN [3] for optimized DL operator kernels and openVINO [4] as an inference engine for Intel CPUs. AMD also has driven MIOpen [86] which is an open source GPU library for DL primitives. As a third-party stakeholder, TVM [55] provides optimized operator kernels automatically generated for various targets devices across different vendors and hardware architectures. Users may also use AutoTVM [56] to fine-tune their operators and extract further speed-up. *Collage* is designed to automatically find the best use of these backend by intelligently dealing with their diversity and characteristics.

Operator Fusion Fusion is one of the most powerful techniques to optimize DL workloads by combining multiple high-level operations into a single kernel. To maximize the benefit, advanced fusion techniques introduce their own unique fusion rules to apply this optimization more flexible beyond a few special cases. For instance, by iterating over every operator in the workload, TVM endeavors to seek for an opportunity to merge each operator with its neighbors and maximize the fused operators with the union-find algorithm [55]. To efficiently explore the fusion opportunities, DNNFusion is developed to employ a detailed classification of potential scenarios. To identify the best fusion plan, FusionStitching conducts Just-In-Time tuning [168]. NVIDIA has actively improved the fusion engine in cuDNN to merge certain patterns of operators at runtime [6]. Internally, TensorRT also

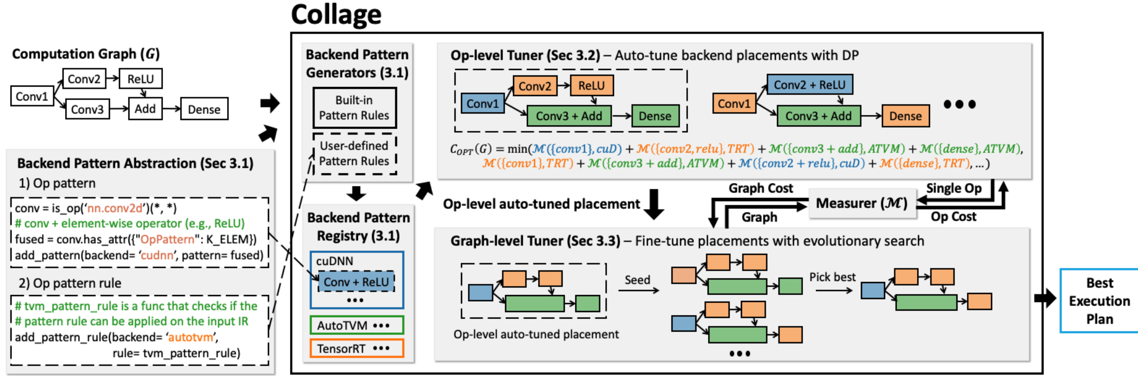


Figure 4.3: System overview of *Collage*

actively apply the fusion to optimize the memory access and scheduling overhead [8]. By offering the highly flexible user interface to describe the pattern rules, *Collage* can support these advance fusion engines as backends.

DL frameworks To provide easy and powerful platform of running a variety of DL workloads, different frameworks have been continuously released and improved. Google maintains TensorFlow [10] and XLA [11] to optimize the execution on various hardware devices including TPUs [2]. By providing a dynamic environment, Pytorch supports dynamic computation graphs more efficiently [9]. As an open-source C++ library and compiler suite for CPUs, Intel has launched nGraph [58]. Also, to support diverse hardware backends and offer a seamless interface to customize the compilation process, TVM [55] is developed. On the other hand, Glow [137] is proposed to efficiently generate the optimized code for multiple targets of heterogeneous hardware. We design *Collage* to help these frameworks customize their ideal execution strategy with multiple backends and extract further performance gain from the current manual approach.

4.3 Overview

Figure 4.3 illustrates the overarching design of *Collage*. As the input, our system takes a DL workload and the specification of backends that will be used for the execution. *Collage* offers two options in the user interface to describe each backend. For simple cases, users may enumerate the supported operator patterns manually. However, this may not be enough to cover the full capability of backends with advanced fusion engines [55, 115, 6, 8]. To enable more flexible specification, *Collage* allow users to bring their pattern rules. When those rules are provided, the backend pattern generator automatically identifies the legitimate operator patterns on the given computation graph and adds them into the backend pattern registry. Section 4.4 provides more details in depth.

Once patterns are ready, *Collage* launches the auto-tuning process in two steps. To

Listing 1 Example of the user interface for the backend description

```python
import collage

# Pattern language to describe conv2d + add + relu.
conv = is_op('conv2d')(wildcard(), wildcard())
conv_constr = conv.has_attr({"data_layout": "NCHW"})
conv_add = is_op('add')(conv_constr, wildcard())
conv_add_relu = is_op('relu')(conv_add)

# Introduce new backend pattern to Collage.
collage.add_backend_pattern(backend='cuDNN',
                            pattern=conv_add_relu)

# Python function describing a pattern generation rule.
# It will be called by our pattern generator.
def pattern_rule(isSingleOpPattern, **kwargs):
  # Checks single op pattern
  if isSingleOpPattern:
    # Single op is always valid pattern
      return True
  # Checks fused patterns
  else:
    # Pattern generator passes this arguments accordingly.
    # cur_type: type of current fusion group
    # src: seed operator node
    # sink: post-dominator of src
    cur_type, src, sink = **kwargs
    # If current fusion group contains a conv/matmul
    if cur_type == kFusable:
      # Helper functions can be defined.
      def fchecker(node_pattern):
        return (node_pattern == kElemwise)
      # Check if every operator between src and sink.
      # Helper function can be passed as a checker.
      if collage.check_path(src, sink, fchecker)):
        return True
    # ... rest of fusion rules ...
  return False

# Introduce new pattern generation rule to Collage.
collage.add_backend_pattern_rule(backend='AutoTVM',
                                 rule=pattern_rule)
```

effectively prune the worthless search space, our first tuner will customize the best execution plan without consideration for the effect of cross-kernel optimizations in the inference engines. By matching the patterns in the registry on the computation graph, our *op-level tuner* explores promising backend placements by adopting the Dynamic Programming (DP) approach with the additive cost model which enables an efficient evaluation of each candidate. Then, as the second tuner, *graph-level tuner* will fine-tune the plan to compensate the missing opportunity from the first tuner by examining the impact of cross-kernel optimizations. By representing each candidate with straightforward encoding, our second tuner traverses the search space with the genetic algorithm [67].

4.4 Backend Pattern Abstraction

To efficiently navigate the vast search space, effective pruning of invalid backend placements is crucial. Thus, auto-tuning algorithm should be well aware of the capability

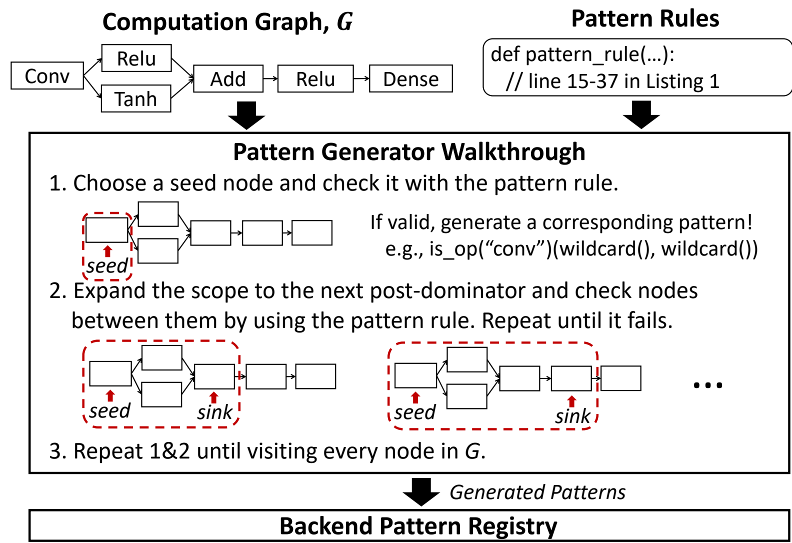


Figure 4.4: Example illustrating how the backend pattern generator would automatically generate valid patterns with the pattern rule assumed in the listing 1.

of each backend so that it would not consider the subspace outside of the valid coverage. To provide accurate specification of the capabilities of various backends, *Collage* offers an intuitive user interface to allow users bring various backends easily. Accurate and easy interface is important to cope with the fast evolving DL backends. For the efficient management of numerous patterns and pattern matching, *Collage* uses a pattern language that extends the Relay pattern language [136].

Listing 1 presents an example of use-case scenario. If a backend only supports a few simple patterns, users may define those patterns manually and add them directly to the backend pattern registry (line 4-11). Users can easily check the operation (line 4), the relation between operators (line 6-7) and attributes, such as data layout or kernel size (line 5). Wildcard operator can be used as a special placeholder that matches with any operator.

To fully support complicated patterns in the advanced backends [55, 115, 6, 8], *Collage* allow users to describe their pattern rules to incorporate more flexible specification beyond naive enumerations (line 15-41). Before the auto-tuning starts, automatic pattern generator in *Collage* will search for valid operator patterns satisfying these rules and update the backend pattern registry accordingly. Figure 4.4 exhibits how the pattern generator would search for the legitimate patterns on the given computation graph. By visiting every node in the computation graph, the pattern generator investigates how far a pattern can grow without breaking the given pattern rule. To assess if a group of operators can be fused, advanced fused engines [55, 115] classifies the type of each operation and determines the fusion decision based on the relationship between different types. For example, line 28-35 defines that element-wise operators followed by operation type of kFusable, which includes

convolution or dense, can be fused. However, it is not valid if a kFusable operator is followed by another kFusable operator. Whenever a group of operators satisfying the rule is found, it generates a corresponding pattern and expands the scope of interests to see if a bigger pattern can be found. With this approach, we could successfully adopt advanced backends, such as AutoTVM, cuDNN, and TensorRT, for our evaluation without any missing pattern.

4.5 Auto-tuning Execution Plan

4.5.1 Backend Placement Problem

To optimize the execution plan, *Collage* focuses on the backend placement problem. Consider a computation graph $\mathcal{G}$ and a set of backend operators $\mathcal{B}$ that are predefined in *Collage* or provided by users. Note that $\mathcal{G}$ is a Directed Acyclic Graph (DAG) where each node represents a mathematical tensor operator (e.g., convolution, batch normalization) and a backend operator is an optimized low-level implementation that each backend provides for the target hardware. With M matched subgraphs g_i and backend operators b_i for $i = 1 \cdots M$, let $\mathcal{P}(\mathcal{G}) = \{(g_i, b_i) | b_i \in \mathcal{B}, \bigcup_{i=1}^{M} g_i = \mathcal{G}, g_i \cap g_j = \emptyset$ for all $i, j \in \{1, 2, \cdots, M\} where i \neq j\}$ be a on a computation graph $\mathcal{G}$ and $\mathcal{M}(\mathcal{P}(\mathcal{G}))$ be the cost function of a placement $\mathcal{P}(\mathcal{G})$. We aim to find an optimized backend operator placement $\mathcal{P}(\mathcal{G})$ such that $\mathcal{M}(\mathcal{P}(\mathcal{G}))$, which is the run-time cost (e.g., execution time) of the placement, is minimized. This problem can be formalized as follows:

$$\mathcal{P}_{opt}(\mathcal{G}) = \underset{\mathcal{P}(\mathcal{G})}{\arg\min}\, \mathcal{M}(\mathcal{P}(\mathcal{G})) \tag{4.1}$$

With two-level tuning approach, *Collage* endeavors to customize the best execution plan by identifying $\mathcal{P}_{opt}$.

4.5.2 Op-level Auto-tuning

To enable the cost-efficient evaluation of a graph with a backend placement candidate and prune the invalid search space effectively, *Collage* conducts an op-level tuning as the first step without the consideration for cross-kernel optimizations. Its goal is to lower all operators on the computation graph into the most efficient set of low-level implementations from user-provided backends. As discussed earlier, the second tuner (Section 4.5.3) would make up for the possible performance loss from this simplification.

As the relaxation would make low-level kernel implementations being independent to each other, the following additive relationship holds between g_1 and g_2, which are two different subgraphs of $\mathcal{G}$ without any interaction, in terms of their run-time cost for the

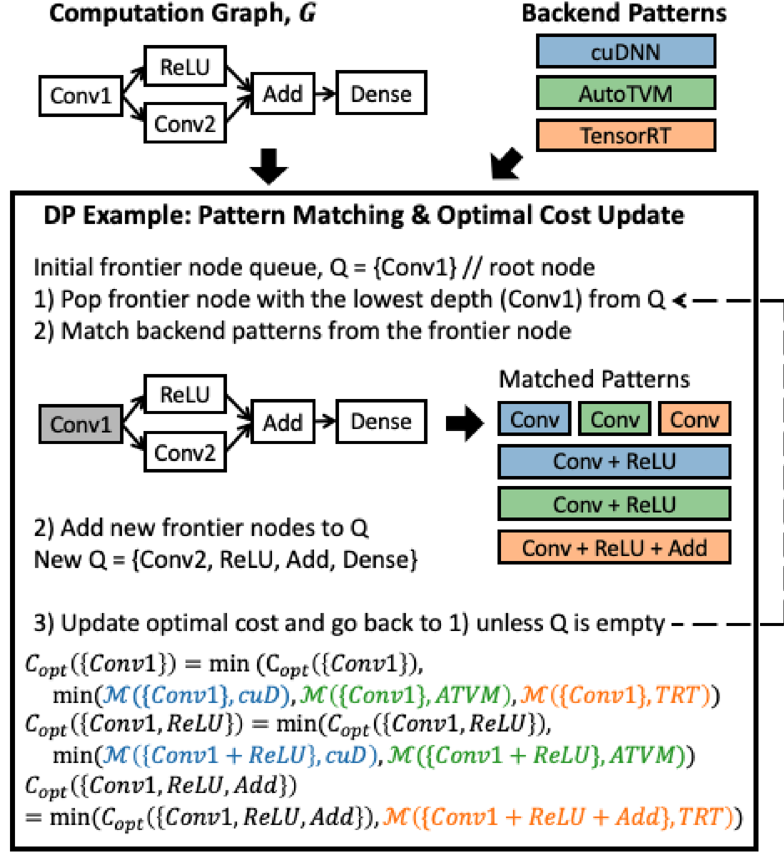


Figure 4.5: Example of Dynamic Programming (DP) procedures for pattern matching and optimal cost update

single device execution [85]:

$$\begin{aligned} Cost(\mathcal{P}_1 \cup \mathcal{P}_2) &= \mathcal{M}(\mathcal{P}_1) + \mathcal{M}(\mathcal{P}_2) + \epsilon \\ \mathcal{P}_1 &= (g_1, b_1), \mathcal{P}_2 = (g_2, b_2), g_1 \cap g_2 = \emptyset \end{aligned} \tag{4.2}$$

ϵ implies a constant for factors, such as driver/switching cost. With this cost model, it is possible to cheaply approximate the cost of a graph by fragmenting a graph into smaller chunks and using their cost to estimate for the original graph. However, since there are numerous ways to split a graph and the number of user-provided backends expands the possibilities in a combinatorial manner, an intelligent method is necessary for the effective tuning.

As a breakthrough, we propose a Dynamic Programming (DP) method as an op-level tuning approach, which effectively navigates the vast search space while handling the complexity from the graph topology and diverse available backend patterns. By leveraging the additive relation (Equation 4.2) between the costs of subgraphs s and g, the following

Algorithm 3 Op-level auto-tuning: DP

Input: Computation graph $\mathcal{G}$ and set of backend operators $\mathcal{B}$
Output: Optimized placement $\mathcal{P}_{opt}(\mathcal{G})$

```
1: // v_0: a root of G, Q: a priority queue sorted by node depth
2: Q = {v_0}
3: repeat
4:     v_s = Q.dequeue()
5:     for b_i ∈ B do
6:         if b_i matches any subgraph g rooted at v_s then
7:             // F is a set of frontier nodes after matching
8:             for v_j ∈ F do
9:                 if v_j has never been added to Q then
10:                    Q.enqueue(v_j)
11:
12:            // P(g) = {(g, b_i)}
13:            // M is an evaluation function
14:            // S is a set of subgraphs forming a complete partial graph of G between v_0 and v_s
15:            // ε is a constant for factors like driver cost
16:            for s_j ∈ S do
17:                if C_opt(s_j ∪ g) > C_opt(s_j) + M(P(g)) + ε then
18:                    C_opt(s_j ∪ g) = C_opt(s_j) + M(P(g)) + ε
19:                    P_opt(s_j ∪ g) = P_opt(s_j) ∪ P(g)
20: until Q = ∅
21:
22: return P_opt(G)
```

equation could be deduced:

$$\begin{aligned} \mathcal{C}_{opt}(s \cup g) &= \mathcal{C}_{opt}(s) + \mathcal{M}(\mathcal{P}(g)) + \epsilon \\ \mathcal{P}_{opt}(s \cup g) &= \mathcal{P}_{opt}(s) \cup \mathcal{P}(g) \\ s \cap g &= \emptyset \end{aligned} \tag{4.3}$$

where $\mathcal{C}_{opt}$ and $\mathcal{P}_{opt}$ keep track of the optimal cost and placement, respectively. To traverse the graph and manage each state, we introduce a *frontier* which also determines s and the starting node of g. In this context, s represents a valid set of subgraphs that can form the part of $\mathcal{G}$ between its root node and the current frontier. g means the part of the graph that will be matched with backend patterns. It can be either single operator or multiple operators that are going to be fused. For the efficiency, our evaluation function $\mathcal{M}$ is designed to be able to memoize the cost of g to avoid the repetitive and expensive measurement overhead (i.e., compilation + multiple runs on the actual hardware) except the cold run.

Figure 4.5 exhibits an example of how our DP method searches for the optimal backend

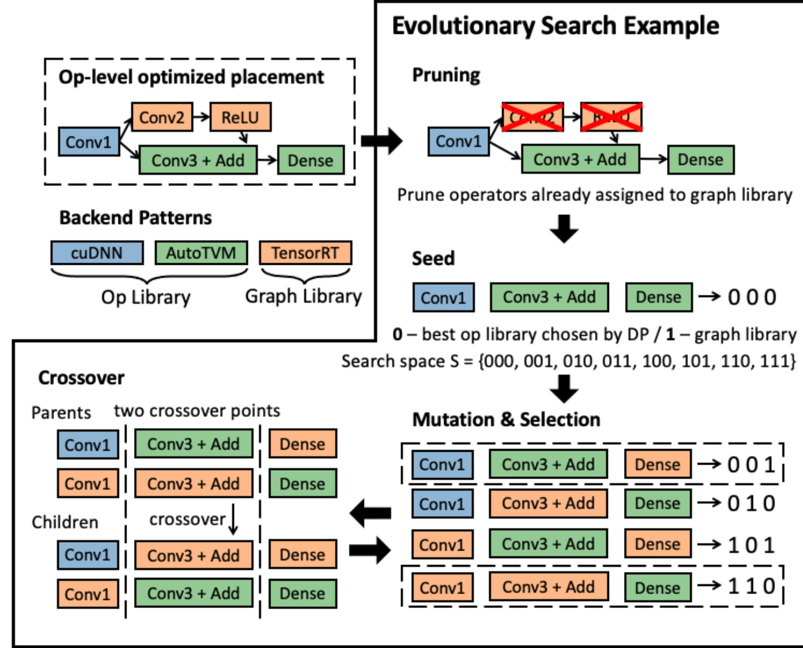


Figure 4.6: Example of Evolutionary Search (ES) procedure

operator placements. As inference engines, such as TensorRT, can also provide efficient operator-level implementations, they were also included in the first level tuning [2]. As an initialization, we put a root node in the priority queue as an initial frontier node. This would imply an unmatched operator node with the lowest depth in the computation graph. By iterating over the queue of frontiers, this method examines if there exist valid backend patterns that can be matched and evaluates each of new possible placement. Whenever a better placement is found, the optimal cost and the placement of the corresponding state will be updated. This process will be repeated until the entire graph is searched over. With this approach, the powerful exploration for the vast search space could be possible by eliminating the redundancy in the search process while having the highly efficient cost evaluation method.

Algorithm 3 describes the detailed algorithm of our DP approach. Note that we iterate frontier nodes in the increasing order of the depth to make sure we explored the all possible states (i.e., to have complete S) before we expand the scope and consider new fragment g. In addition, when matching fused operators, it is crucial to verify that the placement would not result in any cyclic data dependency.

4.5.3 Graph-level Auto-tuning

As the execution plan customized by the op-level tuner is established by ignoring the effect of cross-kernel optimizations in inference engines, *Collage* introduces the graph-level tuner

[2]Fusion rules of the inference engines are implemented based on their documentation [8].

to fix the potential backend mis-placements in the plan from the first level. To do so, we need to examine which operators are better off assigning to the inference engines when the graph-wide optimizations are considered. However, it is still a challenging problem as there are myriad ways to split the graph into pieces and decide whether to offload each piece to the inference engine or not.

For the efficient exploration of promising candidates, we represent each backend placement state by using a sequence of digits where each digit implies the offloading decision to inference engines. To narrow down the search space, we exclude operators that are already mapped with the inference engine from the encoding. Fused operators will be checked whether those can be also fused within the inference engine. If so, they will be considered as a group and a single digit will be used to represent them in the encoding. Otherwise, each operator will have separate digits in the encoding. With this straightforward encoding, we could examine diverse candidates without handling the complexity from the various splits on the graph topology. For the effective search, we adopt evolutionary search algorithm [67].

Figure 4.6 illustrates an example of how the graph-level tuner operates. For an encoding, we assume 0 as to keep the decision from the first level and 1 as to override the decision and offload to TensorRT. With this representation, the evolutionary algorithm can naturally fit with the search problem and apply several rounds of mutation, selection and crossover to customize the better placement. To help the search process, we include the first-level's output as a seed to provide a good starting point.

4.6 Evaluation

This section aims to answer the following questions:

- Can *Collage* find the best mixed-use of diverse backends that outperforms the conventional single backend approach?
- Can *Collage* automatically customize more effective execution plans than the manual approach in the existing DL frameworks?
- Is tuning time affordable? What is the trade-off of *Collage* between tuning time and performance?

4.6.1 Experimental Setup

For evaluation, we implemented the core of *Collage* in the form of a portable Python library and plugged it into TVM. By targeting NVIDIA RTX 2070, four backends with different characteristics were adopted and their best mixed use is investigated: cuDNN [6], cuBLAS [5], AutoTVM [56], and TensorRT [8]. To leverage their full capabilities, their

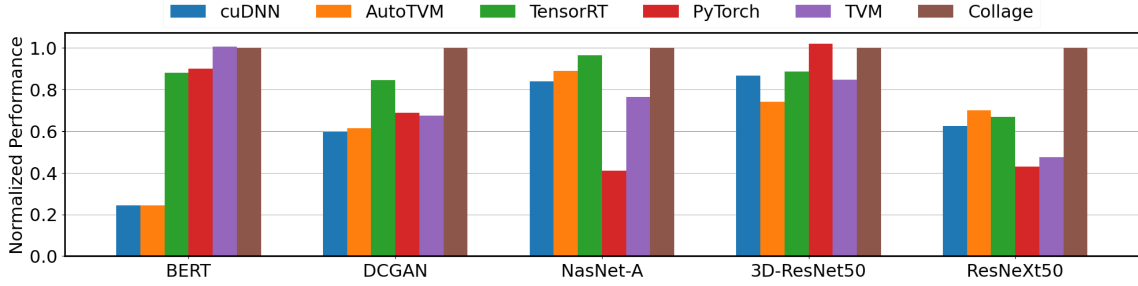


Figure 4.7: End-to-end performance of diverse strategies in five different workloads on NVIDIA RTX 2070. Each performance is normalized by the performance of *Collage*. Note that following state-of-the-arts backends are employed for each framework according to their capabilities: PyTorch (cuDNN, cuBLAS), TVM (cuDNN, cuBLAS, AutoTVM) and (cuDNN, cuBLAS, AutoTVM, and TensorRT).

supported patterns and pattern rules are provided based on their official documentation and code bases.

For experiments, five popular real-life DL workloads, which cover a wide range of applications, are examined. BERT [62] is a transformer-based language model that achieved the state-of-the-art performance on a spectrum of natural language processing tasks. DC-GAN [127] is an extension of the GAN [73] with an unsupervised representation learning mainly for image generation. NasNet-A [170] is one of the most popular machine-generated DL workloads that show strong performance on popular image recognition tasks. 3D-ResNet50 [76] is an extension of widely adopted ResNet50[77] for 3D image tasks such as action recognition. ResNeXt50 [161] introduces a grouped convolution to ResNet50 architecture and improves it in terms of model accuracy and computational complexity for image recongnition.

4.6.2 End-to-end Evaluation

Figure 4.7 presents the end-to-end performance of *Collage* and other popular approaches to utilize the diverse backends of the GPU. As the single backends, we investigated cuDNN, AutoTVM and TensorRT. PyTorch and TVM are examined as the representative deep learning frameworks that provide manually designed execution plan with multiple backends. Note that we denote AutoTVM as a backend to specifically refer the operator implementations automatically generated and tuned within TVM. Due to the limited coverage of cuBLAS, it is not directly compared to other approaches although it is still employed by multi-backend approaches (i.e., PyTorch, TVM, *Collage*). In addition, while PyTorch does not officially support TensorRT, TVM would try to offload the entire graph to TensorRT if possible.

When looking at the performance of single backends, TensorRT offers quite decent performance for every network even though it may underperform by 5% than AutoTVM

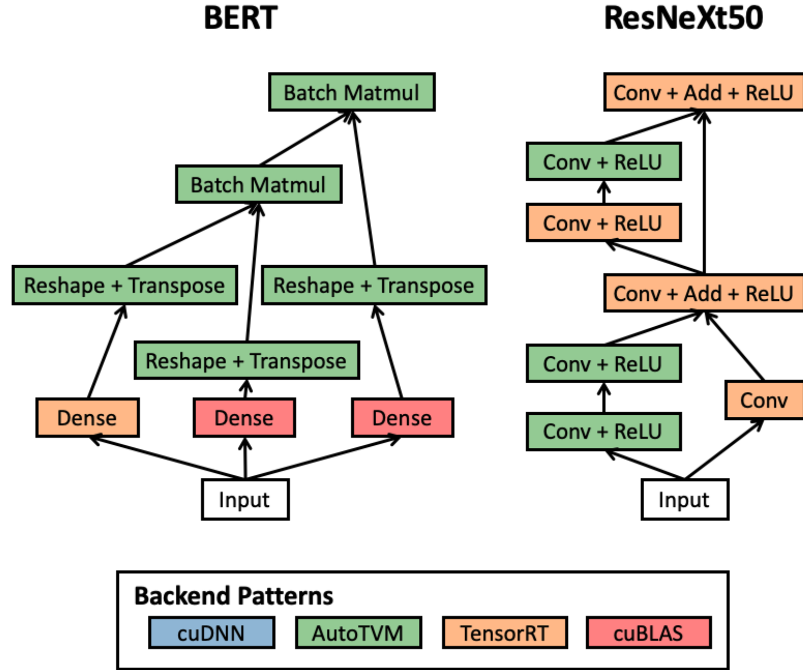


Figure 4.8: Backend placement discovered by *Collage* on ResNeXt50 and BERT.

on ResNeXt50. cuDNN and AutoTVM compete each other by showing their own strength on 3D-ResNet50 and ResNeXt50. Powerful implementation of 3D convolution kernel makes cuDNN show the competitive performance to TensorRT. Because of the lack of the competitive implementation of dense operation, AutoTVM and cuDNN suffer from the significant slow down.

Bars for PyTorch and TVM present the performance of hand-written multi-backend execution plan. Due to the lack of the fine-grained adjustment for each workload, their approaches often show the worse performance than the best performant single backend. Particularly, Pytorch shows more than $2\times$ and $1.44\times$ slower performance compared to each worst single backend choice on NasNet-A and ResNeXt-50 respectively. However, *Collage* showcases the best performance across the all five networks by efficiently customizing the execution plan. Especially, the improvements on DCGAN and ResNeXt50 are significant by extracting 19-43% of additional speedup from the best single backend. Meanwhile, our approach also outperforms manual plans with the superior plans presenting almost $2\times$ faster performance on ResNeXt50. On average, *Collage* could improve the performance $1.18\times$ compared to the best single backend choices while presenting $1.31\times$ enhancement compared to the most performant manual strategies for each workload.

4.6.3 Analysis of Backend Operator Placement

Figure 4.8 illustrates the representative mutli-backend plans customized by *Collage* on ResNeXt50 and BERT. Even within the single network, we could observe that same operation is mapped with different backends due to the performance diversity depending on its configuration as pointed out in Figure 4.1. Dense operation in BERT is the representative example. Interestingly, even with its overhead, the inference engine can be still a good choice for the operator-level implementation as in a convolution operator in ResNeXt50 and a dense operator in BERT. Also, this figure demonstrates that *Collage* is capable of leveraging various fusion patterns from each backend. As in the single operator, our hybrid method chose the different backends according to each context for the identical fusion pattern of Conv+Relu in ResNeXt50. Although this figure does not show the selection of cuDNN, it is also chosen for the other parts of workloads.

4.6.4 Tuning Time

To discuss the tuning time of each tuner in our two level approach, the time took at each level is measured separately. Figure 4.9 presents the result of our operator-level tuning with DP. If the tuning is launched from the scratch, the entire tuning process takes around 20 seconds up to less than two minutes. This tuning time consists of two parts: measurement of the operator cost and overhead from the DP algorithm. Due to the high evaluation cost, the tuning time is dominated by the profiling overhead. However, as discussed in Section 4.5.2, the repetitive profiling for operator cost can be avoided by saving the cost of each operator. When the cost of every operator is profiled in advance, our first level tuner only takes a few second to navigate the search space and customize the effective plan.

Figure 4.10 exhibits the tuning process of our graph-level tuner. As going through several generations of mutations and cross-overs, the evolutionary searcher could gradually improve the performance by leveraging cross-kernel optimizations. In BERT, the effect of cross-kernel optimization is quite notable and thus, our second tuner could enhance the execution plan $1.06\times$ from the plan created by the first tuner. Overall, most of workloads are observed to reach the saturation within an hour. Due to the lack of the efficient cost model that can factor in the cross-kernel optimization effect, it has expensive evaluation overhead that leads to the longer tuning time compared to the first level. Thus, we recommend the graph-level tuning as the optional tool for the users who are interested in maximizing performance at any cost.

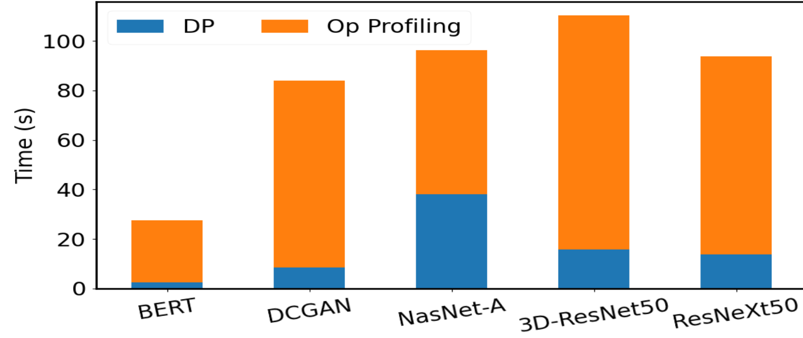


Figure 4.9: Op-level tuning (DP) time breakdown for five different workloads. On average, profiling overhead for operator cost measurements takes up 82% of the entire tuning time. Note that profiling is only necessary for new operators. Once the cost of a new operator is measured, its information will be saved in the logging database in *Collage* to avoid the repetitive profiling.

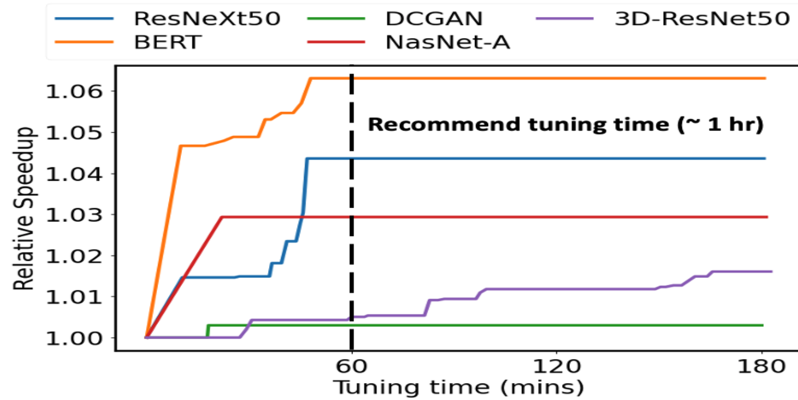


Figure 4.10: Performance improvement of graph-level tuning over time for five different workloads. The y-axis presents the speedup relative to op-level tuning (DP). We observe that one hour of graph-level tuning empirically provides good trade-off between tuning time and speedup for most workloads.

4.7 Conclusion

This work investigated the best uses of various software backends and suggested auto-tuning system, *Collage*, that produces the execution plan with multiple backends customized for the deep learning workload and the underlying hardware. To utilize the full capability of diverse backends, Collage offers a seamless user interface that allows the flexible description of operator patterns supported in each backend. For the effective navigation of the vast search space, Collage introduces two-level auto-tuning strategies to cope with different characteristics of backends with different approaches. For operator-level tuning, this work suggests a dynamic programming method with the consideration of the graph topology and backend capability. Without taking into account the effect from cross-kernel optimizations,

this tuner efficiently customizes a hybrid execution plan. Then, to make up for the potential loss from the relaxation, second tuner fine-tunes the execution plan on the top of the plan customized from the first tuner. The experimental results demonstrate that *Collage* could extract $1.18\times$ speedup compared to the most performant backend for each workload while outperforming the hand-written approach in the state-of-the-arts deep learning framework by $1.31\times$.

Acknowledgement

This is a collaboration work with Byungsoo Jeon, Peiyuan Liao, Sheng Xu, Tianqi Chen, and Zhihao Jia at Carnegie Mellon University.

CHAPTER 5

Low-Cost Prediction-Based Fault Protection Strategy

5.1 Introduction

Over the past decades, researchers have strived to reduce diverse design margins (e.g., noise margin) to maximize performance. However, as tight noise margin is often unable to handle electromagnetic noise properly, computer systems are becoming more susceptible to transient faults which may lead to execution failures [113, 169, 114, 45, 44]. To prevent such failures, diverse techniques introduce redundancy at different levels ranging from hardware to software. Since a transient fault can occur in random location at any given time, the protection strategy must constantly be activated. Also, given that a failure from a transient fault happens rarely at each device perspective, the protection scheme should be fast and cost-efficient to be practical. In general, hardware techniques are powerful but inevitably require design modification and often at high cost. To reduce excessive hardware cost, researchers have proposed software techniques such as Redundant Multithreading (RMT) [72, 112, 133, 147] or instruction duplication under the guidance of the compiler [118, 135, 134, 63]. To provide protection, RMT-based techniques run a redundant thread simultaneously on available built-in resources and validate computations through complex communications between threads. The doubled resource utilization allows for the fast concurrent execution of redundant thread but generally suffer from high energy consumption [106]. Rather than creating a redundant thread, instruction duplication based techniques clone instructions and compare values inside the thread without any additional resources and complicate validation process. Continuous works from Princeton, including the detection strategy called SWIFT [135] and full protection (both detection and recovery) strategy called SWIFT-R [134] [1], propose compiler-directed instruction duplication techniques and

[1] SWIFT with TMR-based recovery mechanism.

provide inspiration to recent explorations [88, 63]. By optimizing instruction duplication, their techniques successfully protect a program with the improved performance overhead. However, slowdown due to increased dynamic instructions remains a concern which needs to be addressed.

This chapter proposes a novel prediction-based protection strategy that greatly alleviates run-time overhead of instruction duplication scheme. Without any extra hardware, our strategy focuses on reducing dynamic instructions that must be executed for protection. We postulate that replicated instructions employed by software protection techniques (referred to as *re-computation*) can be bypassed if software approximation techniques can predict fault-free computation result correctly. Otherwise, re-computation must be triggered to check for a possible transient fault. In such cases, mispredictions cause run-time overhead, but not incorrect output like traditional approximate computing techniques [29, 145, 140]. Avoiding any direct impact on the final output of a program, the prediction is used only for validation. This approach may result in missed faults, but we show that their occurrences can be effectively controlled. Sacrificing some protection quality for significant performance improvement is not a new idea [88, 153]. However, our work investigates a different approach in leveraging the trade-off.

We introduce *RSkip*, a prototype of the automatic compilation system that provides the prediction-based fault protection. The system accepts unprotected source code and creates a fast and resilient executable. Since we observed traditional instruction duplication works [135, 134, 63] often suffer at the loop due to its recurring synchronization points, RSkip focuses on improving the protection cost for the loop. As an accurate and lightweight predictor, *dynamic interpolation* is proposed to estimate computation results in the loop by capturing local trends at runtime. A large saving in run-time instruction overhead is possible by replacing the expensive re-computation with a linear equation. As a fallback predictor, *approximate memoization* [140] is also employed.

Throughout this work, a *Single Event Upset* (SEU) fault model is adopted and the memory system is assumed to be protected given that the commercial DRAMs and caches equip Error Correction Codes (ECC) [135]. Our major contributions are as follows:

- We propose a novel prediction-based protection strategy and demonstrate that software approximate computing techniques are not limited to performance optimization in applications that can tolerate output error. Rather, they can be effectively adapted to detect transient faults in a cost effective and accurate manner. Although this work demonstrates the effectiveness of our new protection strategy by using two loop output estimation techniques targeting several popular computation patterns, its applicability can be broaden with new approximation technique that has a wider target.

- By leveraging the acceptable loss in protection rate [2], our work reduces the number of dynamic instruction greatly to 42.82% while alleviating overhead $1.8\times$ compared to the conventional approach.

- Two approximation techniques are adopted as the prediction model: *Dynamic Interpolation* and *Approximate Memoization*. We newly propose dynamic interpolation that utilizes redundancy for accurate low-cost value prediction. Also, performance and efficiency of approximate memoization are improved from the previous work [140]. With both models, RSkip can bypass 81.10% of re-computation in our target loops.

- Run-time management system is inserted to handle run-time dynamics (e.g., diverse inputs). *Context signature* is suggested to diagnose the current situation and give an indication to tune approximation aggressiveness.

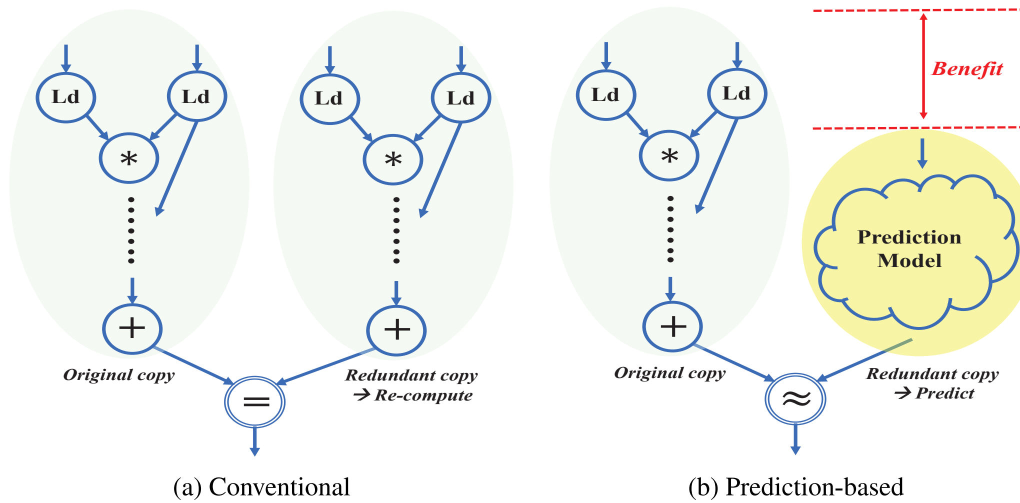
5.2 Motivation and Idea

Conventional instruction duplication techniques [135, 134, 63] replicate the exact same sequence of instructions for fault detection [3] and compare computed values at synchronization points to identify a mismatch which can be a transient fault. Synchronization points include *store*, *branch* and possibly *function calls* depending on the implementation. If a fault is detected, a recovery mechanism will be triggered. Because of the extra instructions for re-computation and validation, previous approaches often suffer from high run-time overhead that may not be feasible in many situations (e.g., mobile device). Thus, we consider alternatives to reduce overhead.

Figure 5.1 illustrates the overarching idea of our prediction-based approach in comparison to conventional approaches. Two strategies adopt distinct approaches toward managing redundant copies of instructions for verification. In the figure, recovery routine is omitted for simplicity.

Figure 5.1a highlights the concept of the conventional instruction duplication techniques. Even without considering extra overhead for recovery, the previous detection techniques perform more than $2\times$ dynamic instructions as the unprotected program due to extra instructions for duplication and validation [4]. As a result of parallelism inside modern processors, slowdown of conventional detection techniques is reported less than $2\times$ [135]. Yet implications suggest that previous approaches would suffer from a large overhead when it is unable to fully utilize parallelism to cover increased dynamic instructions. For instance,

[2] 5 percentage point loss is considered acceptable. See Section 5.7.3 for details.
[3] Recovery requires additional instructions.
[4] With recovery mechanism, the full protection scheme may need to execute more than $3\times$ [134].



(a) Conventional

(b) Prediction-based

Figure 5.1: Idea of each protection strategy. **(a)** Re-compute and validate. **(b)** Estimate and fuzzy-validate.

periodic reaching of synchronization points adds dynamic instructions with dependencies for validations, often causing previous approaches to struggle at the loop. Given that compute-intensive programs consume significant amount of time on the loop, it can be a serious drawback. This phenomenon will be discussed further with the experimental data in Section 5.7.

Rather than replicating identical chains of instructions, prediction-based approach estimates computation results by using software approximation techniques as shown in Figure 5.1b. If prediction and original computation agree within the predefined acceptable range, the original computation is assumed correct and used to proceed the rest of program execution without re-computation. In this work, *relative difference* is used to define acceptable range. Naturally, benefit will be the cost gap between re-computation and prediction model. Otherwise, a value is considered as a perturbation which may be a possible fault as in perturbation screening [126, 139]. In such a case, re-computation will be triggered for the exact validation. **Thus, mispredictions (false positives) result in run-time overhead without having a direct impact on the program output since estimations are only used for validation.** Traditional approximation technique requires high prediction accuracy as it replaces the original computation and thus directly influences program output quality. However, in our approach, prediction accuracy requirement can be relaxed: we only need enough accuracy to recognize perturbation at runtime.

Due to the existence of fuzzy validation, a small error may avoid fault detection. A fault must satisfy two conditions to avoid prediction-based fault detection and cause a failure. First, a transient fault should modify the value in the original copy within the acceptable

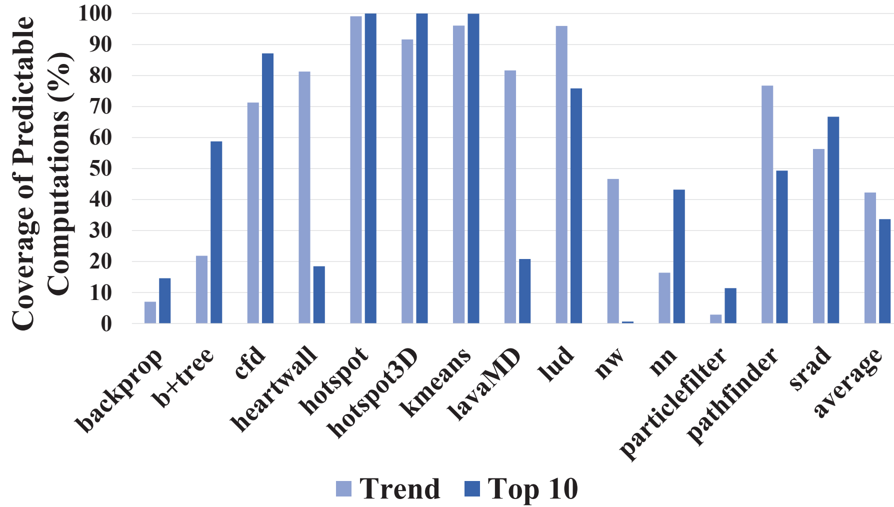


Figure 5.2: Proportion of dynamic instructions whose computation outputs can be estimated. Measured in Rodinia [54] benchmark suite.

range. Otherwise, correction will be invoked. Note, a fault occurred in the redundant copy does not influence the final output of the program since the copy is only used for validation. **Therefore, missed faults (false negatives) can occur, but their occurrences can be explicitly kept to a small amount with proper acceptable range. With reasonable acceptable range, we can effectively control the occurrence of false negatives with consideration for the trade-off for the significant performance improvement.** To set a reasonable acceptable range, the occurrence of false negatives is investigated in Section 5.7.2. Secondly, the corrupted value must be fatal to program fidelity. To avoid this situation, we do not apply approximation on particular types of values, such as pointers or loop induction variables, that can significantly impact program correctness. Thus, they are protected with traditional instruction duplication. Usually, their impact on performance is marginal due to their low computational overhead.

To maximize benefit from prediction-based protection, a approximation model should naturally be lightweight and accurate. Since prior works struggle from high protection overhead over the loop on which a compute-intensive program usually spends significant time, we choose the loop as the main optimization target and design specialized prediction models for the loop. To find an opportunity, we investigate outputs of major loops in Rodinia benchmark suite [54] and observe the following phenomena: (1) outputs produced in consecutive iterations tend to share a certain trend. (2) since the same computation is conducted repeatedly in a loop, there may exist many repeating outputs. A trend in loop outputs occurs along with *spatio-value similarity* [142] that arises when data elements with spatial locality tend to be inherently consistent. If a trend can be captured accurately, it would

be possible to efficiently estimate outputs that require a significant amount of computations. Compared to previous work [142] that simply groups nearby similar values, trend-based prediction can be more effective way of utilizing *spatio-value similarity* by having a wider coverage (values on the same trend might not necessarily have close values.). Likewise, frequent output values can be used for prediction. Figure 5.2 suggests the potentiality of such approaches. We measure prediction accuracy of each method and reflect impact of predictable computations. For the trend-based prediction, data elements showing less than a certain amount of changes in consecutive iterations are considered residing in the same trend. However, sometimes, a few outliers irritate the trend-based prediction. In this motivational experiment, we manually handle those corner cases. Also, we examined the effectiveness of a prediction method that only uses the top 10 most frequent values. Our motivational experiment shows that both prediction models present promising result, implying the chance of bypassing more than 33% of dynamic instructions of the entire program.

However, existing techniques are discovered to be inaccurate or relatively expensive in capturing varying trends. A regression analysis, for instance, requires a process of splitting data elements to determine the coverage of a single equation. In general, however, such a split process is out-of-scope for a regression analysis [111] and naive split policy would result in incorrect estimations due to the tendency of total dependence on the input. Also, a regression analysis may require high training overhead since training should be done for each equation separately. To overcome these challenges, *dynamic interpolation* is proposed. Dynamic interpolation learns how to split data elements to capture varying trends. Rather than learning each equation during offline training, dynamic interpolation computes a linear equation at runtime by using two endpoints in each split and use the equation to estimate values between those endpoints.

Although dynamic interpolation can capture rapidly changing short trends (e.g., data with low spatio-value similarity), there is an upper bound for skipping re-computation as interpolation cannot estimate values for endpoints. To skip re-computations further, we introduce *approximate memoization* [30, 31, 140] as a second-level predictor. Without dependence on trends, the method substitutes expensive computation, such as the function call, with a single access to a lookup table that stores popular repeating values.

Since approximate memoization is generally more expensive than dynamic interpolation, the first prediction will be made by dynamic interpolation. When the interpolation turns out to be wrong, approximate memoization creates a second prediction. Yet, this is profitable as the overhead for two consecutive predictions can still be lower than the overhead of the expensive re-computation. In *blackscholes*, the relative cost between dynamic interpolation, approximate memoization, and re-computation measures 1:1.84:4.18 justifying our

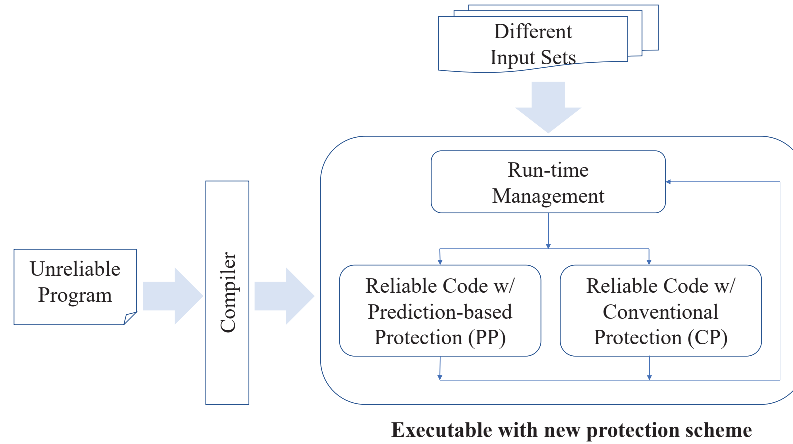


Figure 5.3: System Overview

approach.

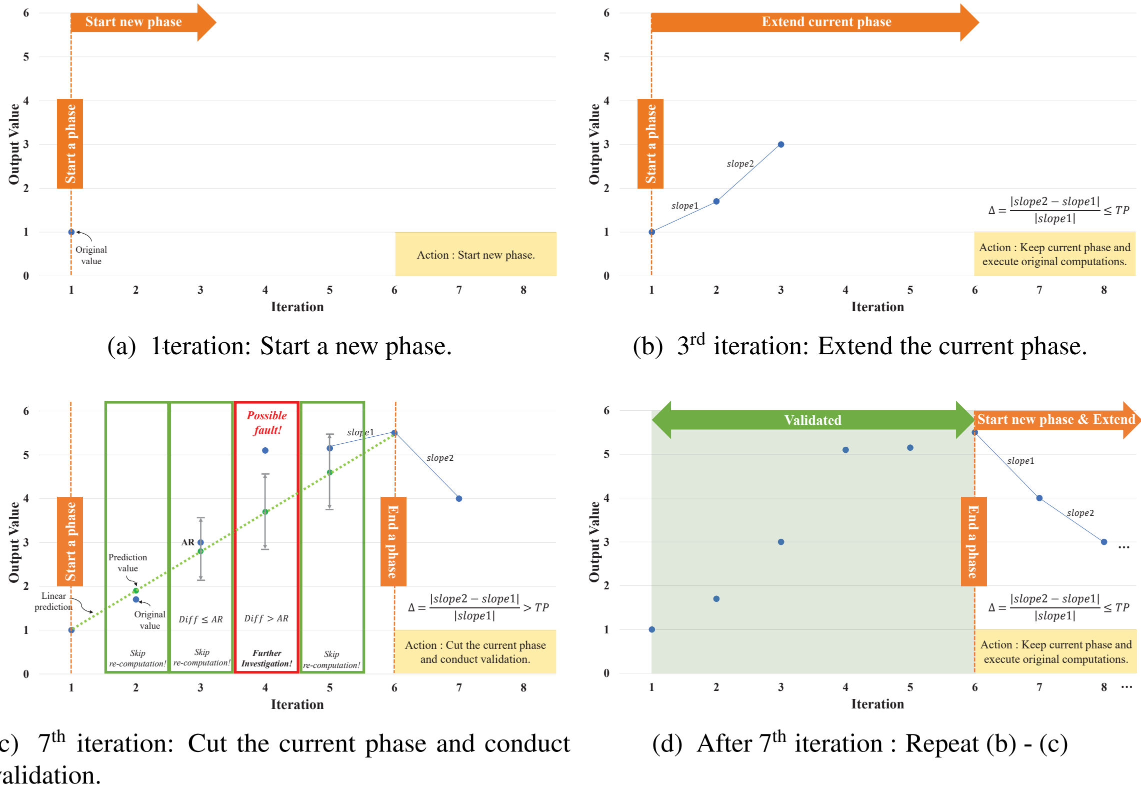
5.3 System Overview

RSkip is a prototype of fully automatic compilation system that provides prediction-based protection. The system takes unreliable source code as an input and generates a lightweight resilient executable. It requires no hardware modification and no preprocessing on source codes or input data sets. [5]

Figure 5.3 outlines the concept of RSkip. By exploring source codes, the compiler conducts a thorough static analysis (e.g., def-use chain) and detects optimization candidates. Since the current prototype focuses on the loop, the compiler identifies the loop that available approximation techniques target. Once a target loop is isolated, two different versions of the loop will be created: a *reliable version with prediction-based protection (PP)* and a *reliable version with conventional protection (CP)*. Later, at execution time, one of two versions will be chosen by the run-time management which is designed to handle the run-time dynamics. CP will be chosen in cases where PP is expected to have no benefit. Code regions which are not selected as the optimization target will be transformed into the CP without PP and interaction with run-time management.

After one-time compilation, the executable starts the offline training process. It observes the run-time context (e.g., trend pattern for dynamic interpolation) to recognize the current situation and learns how to adjust the approximation aggressiveness. Run-time context can be altered when the program runs with different input sets or the same code is executed with different live-in values within a single run. To react such a run-time dynamics, run-time management uses the run-time data to create a *context signature* for each of transformed code

[5] In instances where an user desires the highest protection rate in a specific code region, the acceptable range can be specified as zero by using pragma. Otherwise, a default acceptable range will be applied.



(a) 1iteration: Start a new phase.

(b) 3rd iteration: Extend the current phase.

(c) 7th iteration: Cut the current phase and conduct validation.

(d) After 7th iteration : Repeat (b) - (c)

Figure 5.4: Sketch of dynamic interpolation. Whenever slope change is above tuning parameter (TP), a phase is defined. During the validation process at (c), a data element is considered as a possible fault if the difference between original value and prediction value is greater than acceptable range (AR).

regions with PP. A signature depicts the run-time context and is used to adapt approximation methods. Section 5.5 describes further about context signature and run-time management.

5.4 Prediction Techniques

As previously discussed, we chose approximation techniques specialized to the loop. Since *store* is a synchronization point which often requires heavy computation for its value operand, our approximation techniques estimate values that will be saved in the memory within the loop. However, loops with certain types of value computation (e.g., pointer) or low computation overhead (e.g., initialization) are not considered as the approximation target loop. These will be filtered out by the static analysis with the cost estimation. For simplicity, we target the legitimate types of value computation containing the loop or the user function call that has the number of instructions above threshold as in Figure **??**. The current approximation target may include data that is not generally considered as approximable. However, since the approximation would be used only for validation, it would not harm the protection quality until a missed fault occurs on the corresponding code segment in the

original computation. The missed fault would impact the protection quality according to its influence on the program fidelity, thus our experimental result reflects this factor.

5.4.1 Dynamic Interpolation

5.4.1.1 Target Computation Pattern

Dynamic interpolation targets computation that updates values in the memory at every iteration. Either the expensive function call or computation with the loop can be replaced with a single linear equation. Dynamic interpolation also works on multi-dimensional array.

5.4.1.2 Implementation

Skip rate (i.e., the ratio of iterations skipping re-computation in the loop) of dynamic interpolation largely depends on how a phase (i.e., consecutive data elements that a single equation covers) is sliced onto data elements to capture local trends. Particularly, dynamic interpolation should be intelligent enough to maximize stride (i.e., phase length) on a long trend. When values fluctuate widely, it should lower stride. In general, due to the tendency of total dependence on the input, phase cannot be determined statically. To overcome the challenge, we notice the opportunity to utilize the redundancy inserted for protection mechanism: a copy of the computation can be used as run-time guidance to cut the phase of a redundant computation dynamically.

Figure 5.4 demonstrates the dynamic interpolation algorithm step by step. To decide approximation aggressiveness, we introduce a tuning parameter (TP), which represents optimistic expectation towards outliers on phase slicing. With a higher TP, the algorithm tries to extend a stride more aggressively by ignoring outliers. Although this example assumes a TP is already known, in reality, it will be properly adjusted by the run-time management. More details are explained in Section 5.5. Figure 5.4a shows the setup stage for a new phase. Once the first point is generated, the algorithm proceeds to next iterations to see if an extension is possible. Figure 5.4b presents the extension stage. If a change in the latest two slopes is less than TP, a point is considered to reside in a current trend. While the condition satisfies, the algorithm keeps the current phase and proceeds to the next iteration. Up to this point, only original computations are performed and their results are saved in their original memory space without allocating any extra space. In case that the loop reads and updates same memory locations (e.g., A[i]+=1), we allocate temporary space to keep the original value and use it for re-computation. Although managing temporary space occurs some overhead, our evaluation shows significant improvement is still possible (See benchmark *lud* in Section 5.7.1.). If a slope change is above TP, the phase is cut at the previous iteration. The cut stage is described in Figure 5.4c. When a phase is defined, data elements in the phase are validated with a linear prediction computed by two endpoints. If

the original computation and the prediction agree within the acceptable range, the algorithm assumes fault-free and bypasses the re-computation. Otherwise, it suspects a possible fault and triggers further investigations. Once the validation for the current phase is done, the next phase starts. In this case, the setup stage is no longer necessary. Therefore, the extension stage and the cut stage are repeated after iteration 7 as shown in Figure 5.4d. Note, our work did not set any upper bound for a stride.

5.4.2 Approximate Memoization

5.4.2.1 Target Computation Pattern

To apply this method, the computation should generate the identical output on the same input set without any side effect (i.e., I/O execution). Also, the number of inputs should be reasonably limited as lookup table size is expected to grow exponentially. These strict requirements force the technique to have narrower applicability than dynamic interpolation.

5.4.2.2 Implementation

Due to the limited memory space, a lookup table cannot contain all possible cases. Therefore, approximate memoization will group the nearest inputs through the quantization process and make them access the same entry in the lookup table. Naturally, its accuracy and performance entirely depend on the table size and the quantization-based address calculation. In general, a larger table shows higher accuracy, but its size should be determined with consideration for both memory size and performance benefit. An overly large table may not fit in the cache and induce complexity to the quantization process along with the address calculation. Given that the technique should perform the quantization-based address calculation for every inference, its complexity will affect access time towards the memorized result. If the prediction target is computed by many inputs with a diverse spectrum of values, the lookup table size needs to grow large enough to cover a variety of input combinations while being able to return saved results faster than original computation time. Therefore, it is important to find an intelligent lookup table management strategy and the quantization becomes a key process in lookup table construction. In general, the construction algorithm determines the number of quantization levels for each input by assigning a certain number of bits in the address bits. Since the width of the address bits is decided by the lookup table size, the smart management strategy should be able to distribute the limited number of bits to each input while maximizing prediction accuracy. To overcome this challenge, Samadi et al.[140] propose the systematic quantization method with the bit tuning process. By allowing more bits, the technique enables inputs with higher impact on the final output to better differentiate their input sets. After bit tuning, both minimum and maximum values for each input are used to determine the coverage of each quantization level. With the assumption

Table 5.1: Selected benchmarks. The impact of skipping re-computation can be imagined by provided computation type and location. Both training input and test input are randomly generated or selected without any intersection.

Benchmark	Application domain	Computation type of prediction target	Input
conv1d	Signal processing, Machine learning	A reduction loop	4048 integers
conv2d	Signal processing, Machine learning	Nested reduction loops with conditional statement	200*200 integers for input vector, 15*15 integers for kernel
sgemm [152]	Linear algebra	Nested reduction loops	1024*1024 integer matrices
kde [109]	Machine learning	Nested reduction loops	1500 float vector
forwardprop [54]	Machine learning	A reduction loop	1024*1024*1024 network
backprop [54]	Machine learning	A reduction loop	1024*1024*1024 network
blackscholes [40]	Finance	A function call	65536 cases
lud [54]	Linear algebra	A reduction loop with a varying trip count	1024*1024 float matrices
YOLOv2 [132, 131]	Machine learning, Computer vision	A reduction loop	0.1 ~10MB images

that inputs are uniformly distributed, they equally divide the region between the minimum and maximum values into the assigned number of quantization levels. But, when inputs do not follow a uniform distribution, significant inefficiency may arise with this approach. Therefore, in this chapter, the coverage of each quantization level is dynamically determined based on the profiling analysis. For dynamic determination, we first build a histogram with narrow-ranged uniform-length bins and gradually combine nearby less-crowded bins. It becomes apparent the new approach is able to build a more efficient lookup table than the previous work. At *blackscholes*, the previous approach encodes three inputs out of six with 15bit-wide address. Upon the same function, our approach encodes six inputs with an equal width of the address. Naturally, accuracy is improved from 96.5% to above 99% when testing with given representative inputs in the benchmark suite [40]. Once compiler statically identifies candidate loops, lookup tables will be constructed and examined during training phase. If the lookup table shows good prediction accuracy with training data, it will be deployed at runtime. However, when run-time performance is not as good as expected, run-time management may disable the deployment.

5.5 Run-time Management

Since traditional approximation techniques replace the original computation, Quality-of-Service (QoS) management is a key process to guarantee a certain level of output quality while providing significant performance improvement [141, 36, 99]. Green [36] is the representative framework that guarantees QoS of approximation techniques at runtime. Before its deployment, Green constructs a QoS model at offline by using user-provided inputs. Later at deployment, the framework observes the run-time context and adjust approximation decisions accordingly with its QoS model. To handle input diversity, we follow a similar approach. In our work, the prediction accuracy of each approximation

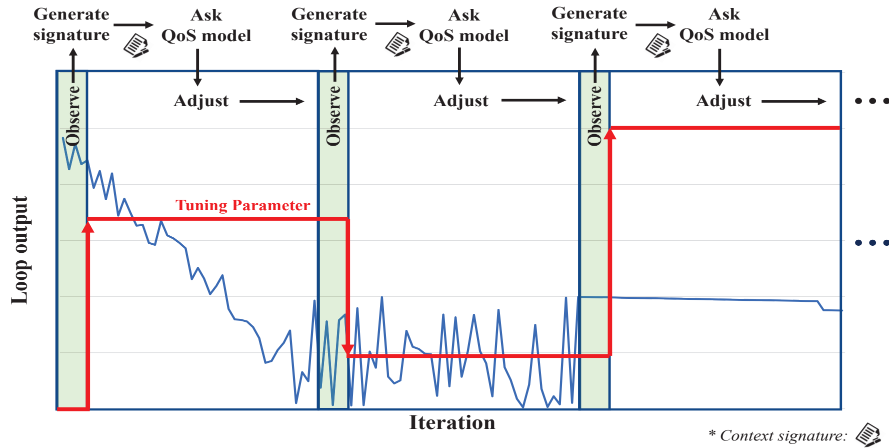


Figure 5.5: An example of run-time management for dynamic interpolation. Run-time management periodically generates context signatures by summarizing the current run-time context to adjust TP based on the QoS model.

method is adopted as QoS metric. To provide high QoS, a *context signature* is defined to provide each approximation technique with a meaningful summary of the run-time context. For dynamic interpolation, the statistics of local trends can be a useful indicator reflecting the run-time context. In our experiment, we generate a signature by using histogram of slope changes which implies the impact of TP. For example, signature ”312” means 3rd bin has the largest count followed by 1in and 2nd bin. Since the run-time context may change during an execution, the management system periodically triggers observation and adjustment processes. Thus, an executable may generate more than one signatures during its execution. Figure 5.5 illustrates how the run-time management adjusts a parameter for dynamic interpolation by using the context signature. Blue line indicates output values across iterations. Note, TP needs to be adjusted based on the slope changes rather than the number of outliers. Since our dynamic interpolation algorithm will split phases into two at the outlier when its divergence is greater than TP, a large TP is preferred on a big trend to ignore small outliers. Therefore, our QoS model will escalate TP in a long trend to extend the phase aggressively. In contrast, the parameter should be decreased in widely-fluctuating short trends to avoid wrong predictions. The QoS model will monitor the prediction accuracy at run-time and may disable the dynamic interpolation at low accuracy. However, we could not observe this case in our experiment. Since approximate memoization does not depend on the parameter, its QoS model simply monitors the occurrence of misprediction and disables its usage at poor run-time accuracy.

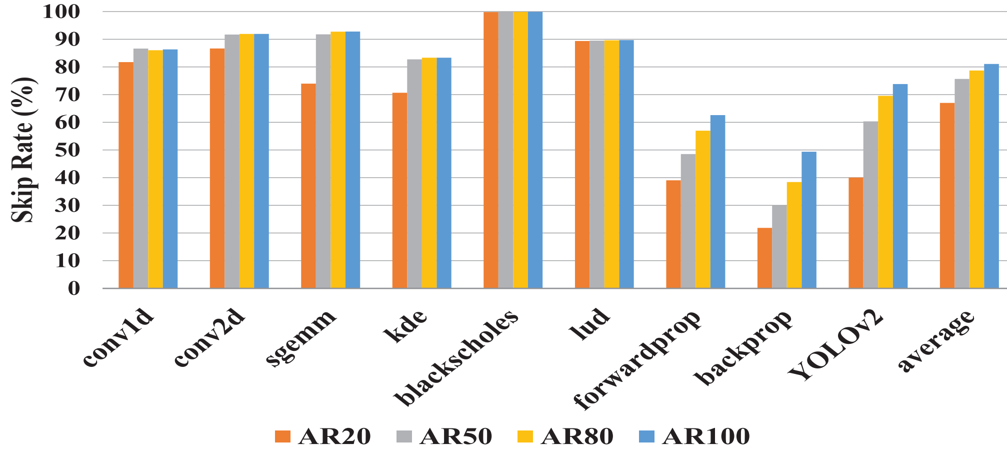
5.6 Training

During the offline training phase, RSkip will build prediction models and construct their QoS models. By utilizing the training set provided by users, RSkip samples outputs from detected loops. For dynamic interpolation, RSkip simulates its algorithm on samples by sweeping various parameters and monitors performance (e.g., skip rate) to identify the best parameter for each signature. Note, we ”simulate” algorithm (i.e., phase-splitting and prediction) without repeatedly running a real program to minimize training time. Once the best parameter is identified, RSkip builds a QoS model which includes a table containing (signature, best parameter) pairs. Later at runtime, RSkip simply reference this table and load the learned parameter when a signature is found. Otherwise, we kept the previous tuning parameter although different policies can be chosen. In addition, the lookup table will be built by following our construction algorithm discussed in Section 5.4.2. Then, during the inference phase, approximate memoization makes predictions by simply reading memorized results in the lookup table.

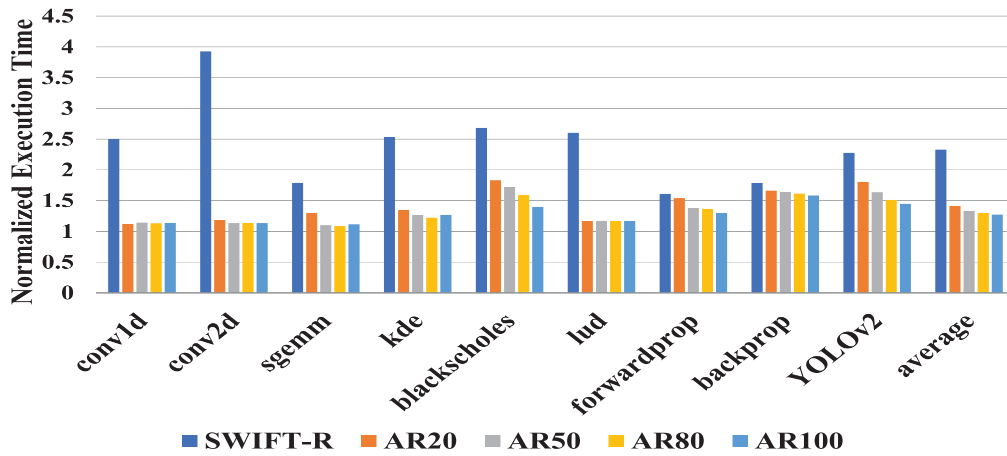
5.7 Evaluation

As described in Section 5.4, our prototype implementation of RSkip targets a computation containing the loop or the user function call as a candidate for prediction-based protection scheme to evaluate its potential. When a program does not contain such computations, they are protected solely by the conventional protection scheme. Therefore, nine applications with target computation types are chosen as benchmarks from various domains. For a case study, diversity of detected patterns is also considered during benchmark selection. The selected benchmarks represent compute-intensive applications containing dominant loops. Especially, *YOLOv2* is a successful real-life object detection application. Table 5.1 explains characteristics of each application. To ease preparation effort, both training inputs and test inputs are randomly generated by using the input-generator or selected from well-known image archives without any intersection between them. If inputs can be selected manually, better training quality would be expected. Throughout the experiment, 20%, 50%, 80%, and 100% acceptable ranges [6] are assumed for RSkip (labeled as AR20, AR50, AR80, and AR100 respectively). The rationality of the acceptable range will be discussed in Section 5.7.3. Note, approximate memoization is only applied to *blackscholes* due to its strict requirement as previously discussed in Section 5.4.2. Without focusing on the improvement of the recovery mechanism, the re-computation based recovery technique is chosen in this work and its impact is reflected in our analysis. Since fault detection and fault recovery mechanism can be investigated independently [135, 88], a better recovery

[6] Relative difference is used in this work.



(a) Average skip rate



(b) Normalized execution time

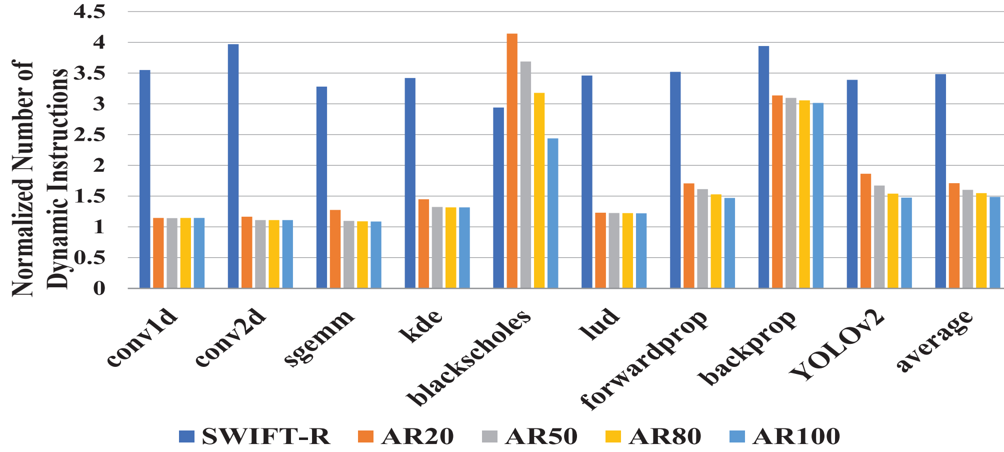
Figure 5.6: Test result with test inputs of each benchmarks (AR : Acceptable Range)

mechanism can be incorporated if applicable. To handle interaction with the unmodified library function, both schemes set the function call as the synchronization point. As baseline, SWIFT-R [134] is chosen since it is one of the most well-known instruction duplication techniques that provide full protection.The execution is forced to use a *single thread* and the experiments are conducted after an automated training session with training inputs. The extension for multi-threaded execution remains as the future work.

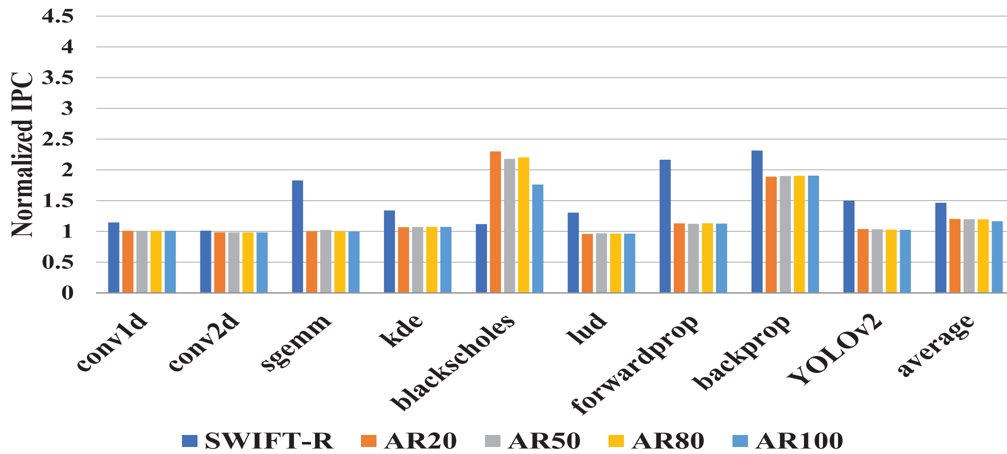
5.7.1 Performance Overhead

The performance of each protection scheme is measured by running benchmarks on an Intel Xeon CPU E31230 with 3.20GHz. Also, *papi* library [158] is used to measure the number of dynamic instructions and Instructions Per Cycle (IPC).

Figure 5.6 and Figure 5.7 show the performance with test inputs for all benchmarks. Figure 5.6a presents the ratio of iterations skipping re-computation in the detected loop.



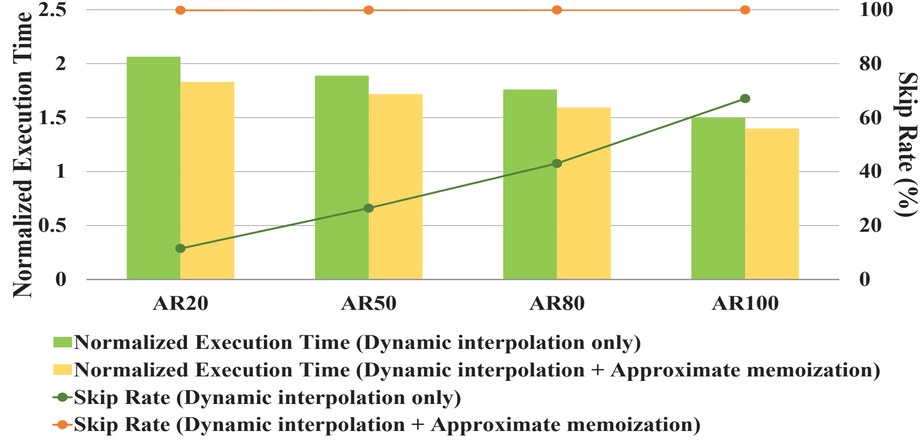
(a) Normalized number of dynamic instructions



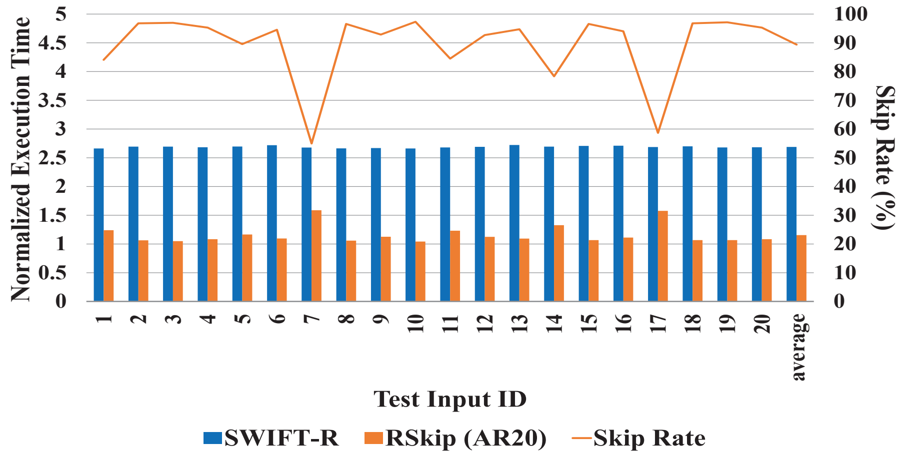
(b) Normalized IPC

Figure 5.7: Test result with test inputs of each benchmarks (AR : Acceptable Range)

Overall, 67.03% of re-computation can be bypassed with the value prediction at AR20. With a wider acceptable range at fuzzy validations, the skip rate increases steadily with AR50 (75.67%), AR80 (78.73%), and AR100 (81.10%). The performance overhead is also presented in Figure 5.6b. **The entire program execution time of each protection scheme is normalized by the execution time of the unprotected program.** On average, SWIFT-R suffers from 2.33× slowdown due to repeating synchronization points. A 20% acceptable range shows a 1.42× slowdown with 67.03% skip rate. The performance also improves due to the increasing skip rate as the acceptable range grows. When the acceptable range is set 100%, the slowdown is measured only 1.27× with 81.10% skip rate. By using skip rate, frequency of recovery mechanism can be rougly estimated. To further understand the experimental results, the normalized number of dynamic instruction and IPC for the entire program execution are presented in Figure 5.7a and Figure 5.7b respectively.



(a) Test result in *blackscholes*.

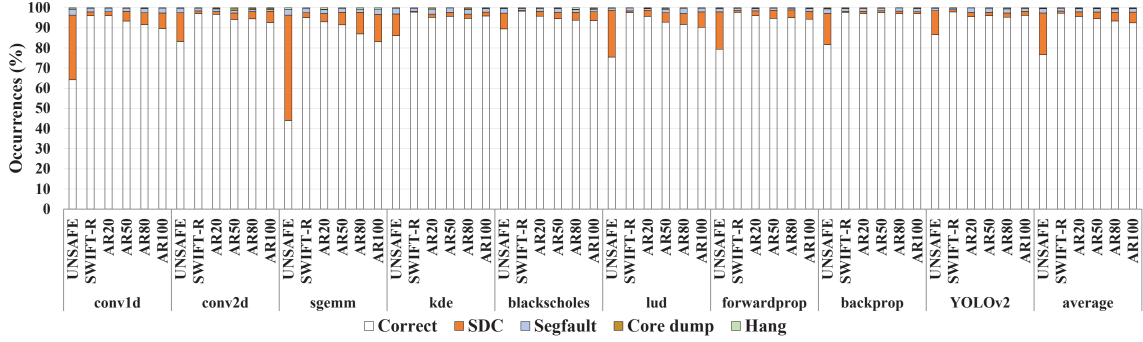


(b) Test result of AR20 in *lud*.

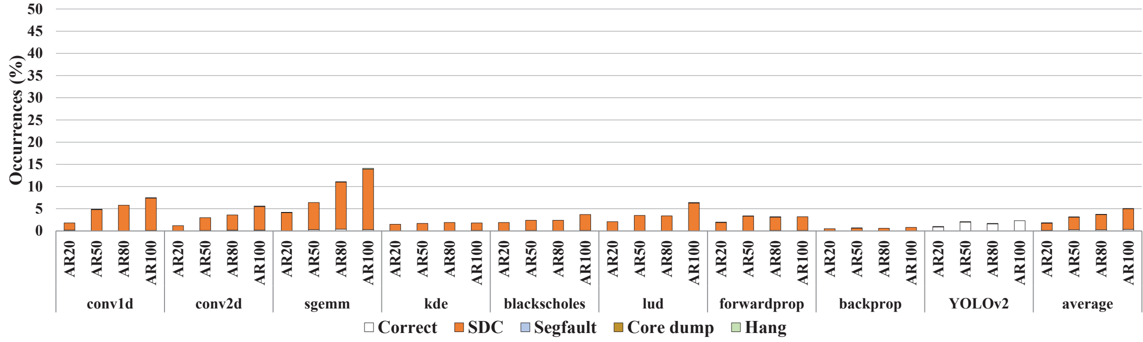
Figure 5.8: The detailed analysis for two selected benchmarks

SWIFT-R executes $3.48\times$ dynamic instructions while having only $1.47\times$ IPC improvement. As the increase of IPC is not enough to hide the additional instructions for protection, the technique suffers from slowdown. On the other hand, RSkip achieves performance improvements by inserting significantly fewer instructions while having a similar level of IPC with an unprotected program. AR20 inserts only $1.71\times$ additional instructions which can be decreased further by $1.49\times$ at AR100.

The performance of SWIFT-R is degraded by a large amount in *conv2d* as the value is calculated in nested loops with conditional statements inside them. Because of recurring synchronization points in the code with a complicated control flow, SWIFT-R cannot exploit the hardware parallelism well enough so that its IPC is almost the same with the unprotected program. At the same time, it executes $3.97\times$ extra instructions for repeating validations. In this case, the benefit of skipping re-computation is significant. By skipping 86.67% of



(a) Fault injection experiment : 1,000 faults are injected for each benchmark.



(b) Measurement of false negatives.

Figure 5.9: The result of the fault injection experiment. Each protection scheme is tagged under every application.

re-computation, AR20 shows the best-averaged performance improvement in *conv2d* among selected benchmarks. And the performance can be improved further with broader acceptable ranges.

blackscholes shows the highest skip rate, which is above 99%, with every acceptance rate due to the existence of approximate memoization as a second-level predictor. Interestingly, even without the significant change in skip rate, the performance improvement is still observed when the acceptable range grows. To understand this phenomenon, the detailed analysis is conducted for this application as shown in Figure 5.8a. The figure evaluates the presence of the fallback predictor by comparing both normalized execution time and skip rate. Without approximate memoization, AR20 shows $2.07\times$ runtime overhead with 11.47% skip rate and the gradual improvement is observed as the acceptable range broadens. With 100% of acceptable range, the skip rate of dynamic interpolation increases up to 67.03% resulting in $1.50\times$ slowdown. This can explain the gradual performance improvement of the RSkip with the presence of secondary predictor. When the re-computation is skipped at the secondary predictor, the estimation cost includes the overhead of the first prediction as well as the second predictor. Although, the cost is cheaper than re-computation, it is quite

expensive given the relatively high cost of approximate memoization. Thus, the contribution of the first-level predictor in the skip rate has a noteworthy impact. Consequently, as shown in Figure 5.8a, the performance of RSkip [7] can still improve even among the similar skip rates due to the increase of the contribution from the first predictor in the skip rate with broadened acceptance rate.

To assess the impact of input diversity, the variance in performance and skip rate towards different test inputs are observed with AR20. Figure 5.8b presents a representative result with *lud*. Mostly, performance and skip rate are measured close to $1.15\times$ with 90.00% respectively. The worst case is measured as $1.59\times$ slowdown with 55.00% skip rate. Yet, significant enhancement from SWIFT-R is observed. The best case is measured as $1.07\times$ slowdown with 97.15% skip rate. In this case, the overall run-time overhead for full protection scheme (detection + recovery) is only 7% of unprotected program execution.

5.7.2 Reliability

The reliability aspect is evaluated using *Statistical Fault Injection* (SFI) [65, 88] in Gem5 simulator [41]. Each of nine benchmarks is executed 1,000 times on the out-of-order cores with a configuration of ARMv7-A. By following the previous work [88], a single bit flip is injected randomly into the simulated processor during each run. **In our reliability evaluation, faults are only injected into the detected loops to strictly evaluate the reliability of prediction-based protection approach. Note that, in real-life execution, a fault can occur at regions outside of detected loops that are protected by the traditional approach.**

The simulation result is categorized into five classes:

- Correct: The execution generates correct output without any data corruption. Conventional approximation methods often ignore certain amount of error [29, 140, 145] in the output quality. **However, our evaluation considers even small output errors as bad quality and only 100% of output quality as ”Correct”.**
- Silent Data Corruption (SDC): An influence of single bit flip silently remains until the program termination and creates corrupted output.
- Segfault: Failure due to illegal memory access.
- Core dump: The injected fault results in a system crash or an abnormal termination of the program.
- Hang: The program cannot terminate its execution.

[7] RSkip employs both dynamic interpolation and approximate memoization

Figure 5.9 shows the result of fault injection experiment. Fault protection rate of each protection scheme is provided in Figure 5.9a. For comparison, the unprotected program is included and labeled as UNSAFE. On average, 76.68% of injected faults are masked in UNSAFE. Additionally, they suffer from 20.72% SDCs and 2.13% Segfaults. SWIFT-R shows a 97.24% protection rate, with the decreased occurrences of SDCs and Segfaults to 1.08% and 1.40%, respectively. Such failures are caused by limitations of the software-only protection scheme [135]. Since there is no dedicated mechanism to protect special registers, detection is not possible when a transient fault strikes a single bit in the opcode field and changes the instruction into operations like *branch* or *store*. Also, a transient fault may occur at the examined register before its actual usage. In this stage, as validation is already done, the program will proceed rest of the execution with erroneous value. Any compiler optimization to reduce register lifetime will be helpful in this scenario.

AR20 demonstrates a comparable level of protection rate to SWIFT-R by exhibiting a 95.67% protection rate with 2.23% of SDCs and 1.63% of Segfaults. Protection rate decreases when acceptable range broadens: AR20 (95.67%), AR50 (94.51%), AR80 (93.42%), AR100 (92.52%). The differences in protection rate are mainly originated by SDCs because we protect address calculation of memory instruction with the conventional strategy: AR20 (2.23% SDCs), AR50 (3.37% SDCs), AR80 (4.30% SDCs), AR100 (5.29% SDCs). Throughout the experiment, the occurrence for Core dump and Hang in every protection scheme, including UNSAFE and SWIFT-R, is measured less than 0.3%.

Figure 5.9b presents the statistics of false negative. As expected, its occurrence increases with widened acceptable range: AR20 (1.80%), AR50 (3.12%), AR80 (3.74%), AR100 (5.04%). As fuzzy validation is applied to data validation, false negatives mostly produce SDCs. Interestingly, due to *YOLOv2*'s program characteristics, false negatives are generally benign in this application. After extensive computations through multiple layers, only a label with the highest probability for each detected object is produced as the output. Naturally, small errors that eschewed fuzzy validations tend to be logically masked in later part of the execution.

5.7.3 The Rationality of Acceptable Range

Before the deployment of RSkip, the acceptable range of fuzzy validation should be determined with consideration of both protection rate and performance. On average, the representative conventional scheme, SWIFT-R, can provide 97.24% of protection rate with $2.33\times$ slowdown. As for widening acceptable range, RSkip achieves significant performance benefits in exchange of certain loss for the protection rate : AR20 (95.67% of protection rate with $1.42\times$ slowdown), AR50 (94.51% of protection rate with $1.33\times$ slowdown), AR80

(93.42% of protection rate with $1.30\times$ slowdown), AR100 (92.52% of protection rate with $1.27\times$ slowdown). With AR20, RSkip can significantly reduce protection overhead while presenting comparable level with the conventional scheme. Also, RSkip can reduce the overhead further to $1.27\times$ with the same amount of loss (5 percentage point) that previous works [87, 88] leveraged.

5.8 Related Works

Hardware techniques insert additional hardware module to protect their execution. For example, Austin proposed DIVA [35, 157] checker which is a small core designed to validate the computation on the fly. Despite their expensive cost, hardware techniques have been adopted in real systems for the high fidelity [81, 146, 162]. Additionally, Racunas et al. [126] proposed a perturbation-based screening hardware technique considering an erratic value as a possible fault. Legitimate sets of values are defined to detect inconsistency in values.

After Saxena et al. [144] noticed the SMT redundancy, many techniques were proposed to reduce hardware cost [72, 112, 147]. To improve performance of thread-level approach, DAFT [166] exploits speculation to minimize inter-thread communications for memory operations. Yet, utilizing redundant threads generally suffers from high energy consumption [106] due to an increased number of dynamic instructions for redundant threads and validation.

Additionally, instruction duplication based protection was first introduced by Oh et al. [118] where all instructions including memory operations were duplicated and validated. Reis et al. proposed SWIFT [135] to optimize instruction duplication scheme by removing unnecessary memory redundancies based on the assumption of ECC. As SWIFT did not claim any recovery method, Reis et al. expanded SWIFT further with TMR-based instruction level recovery technique [134]. In general, fault detection and fault recovery mechanism can be investigated independently [135, 88]. Encore [66] and checkpoint-based methods [65, 155] are proposed as the independent recovery scheme. At wrong predictions, our prediction-based protection scheme executes re-computation for the exact validation. In this case, the mechanism generating re-computation for the redundant copy resembles a form of recovery mechanism. Also, unlike previous works [135, 134, 88, 63] that trigger recovery routine only on the actual fault detection, our approach additionally triggers recovery mechanism on mispredictions. To enhance performance of both detection and recovery mechanisms in RSkip, the integration of advanced recovery mechanisms [66, 65, 155] can be studied.

To reduce the protection cost, Khudia et al. [88] leveraged certain loss in protection rate by protecting only critical variables. Similarly, Venkatagiri et al. proposed Approxilyzer

[153], a framework that suggests protecting critical dynamic instructions by quantifying quality impact of a single bit flip to avoid excessive protection cost. In contrast to our work, they reduce protection cost by narrowing down protection targets. By incorporating this technique, a better performance can be achieved.

5.9 Conclusion

The unique properties of a transient fault force protection strategies to be fast and cost-efficient. RSkip demonstrates that the performance of the software-only protection strategy can be significantly improved with a newly proposed prediction-based protection scheme. By predicting the value of computation with approximation techniques, the expensive re-computation can be bypassed. As a result, 81.10% of re-computation can be skipped on average with a remarkable performance improvement. While conventional technique suffers from 2.33x slowdown compared to the unreliable execution, RSkip only suffers 1.27x slowdown with a 5% loss in protection rate.

Acknowledgement

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), under Award Number DE-SC0014134.

CHAPTER 6

Conclusion and Future Works

6.1 Conclusion

Due to the increasing complexity of both hardware and software systems, it becomes a highly challenging environment for compilers to deliver the optimal executable of a program to the users. To deal with this complexity, most of state-of-the-arts compilers depend on the manual approach where the optimization decision heuristic is hand-written by the compiler developers. Although this approach had been successful by leveraging the compiler developers' expertise, it cannot handle the recent computing environment effectively anymore by frequently applying sub-optimal optimization decisions for the given workload and underlying hardware. In addition, these manual approaches are naturally labour-intensive and error-prone for the fast evolving landscape, such as deep learning domain.

To attack this challenge, this thesis proposed diverse automatic techniques targeting representative state-of-the-art compilers targeting various hardware architectures in diverse domains:

Chapter 2 examined one of the most representative compiler optimizations, *loop unrolling*, and study the feasibility of automatic construction of its decision model in the mobile dynamic binary translation system. By effectively learning from the pattern between optimal decisions and loop features, this approach could save extra 50% in the dynamic instruction counts compared to the baseline heuristics handmade by industry experts while satisfying the tough constraints in memory/runtime from the mobile dynamic translation system.

As the potential of automatic approach for single optimization decision is well demonstrated, Chapter 3 expanded the scope of interests and endeavored to address the problem of finding the best use of a group of optimizations in the presence of interaction between optimizations. This work proposed an auto-tuning technique, called *SRTuner*, that customizes

the effective optimization setting for the given workload, compiler, and underlying hardware by fully utilizing its tuning budget. With a multi-stage structure and novel impact estimation method, SRTuner could reveal the important synergistic relations between optimizations and use it directly to focus on the promising subspace. In addition, to maximize the tuning quality while efficiently handling numerous local optima, the concept of multi-armed bandit problem is adapted to compiler optimization problem. This technique is evaluated on three representative compiler stacks from different domains: GCC (traditional C/C++ compiler) targeting CPU, TVM (domain-specific machine learning compiler) targeting GPU and OpenCL compiler (kernel compiler for heterogeneous computing) targeting CPU/GPU. The evaluation showed that SRTuner can extract up to $34.4\times$ speedup from the best optimization setting from each state-of-the-arts compiler stack while outperforming prior tuning approaches. Due to the unique tuning strategy of SRTuner, it can offer the analytic information regarding important synergistic optimizations back to the users. This work also discussed how this information can be used for future compiler design and studies.

In addition, Chapter 4 investigated the best uses of various software backends and suggested auto-tuning system, *Collage*, that produces the execution plan with multiple backends customized for the deep learning workload and the underlying hardware. To utilize the full capability of diverse backends, Collage offers a seamless user interface that allows the flexible description of operator patterns supported in each backend. For the effective navigation of the vast search space, Collage introduces two-level auto-tuning strategies to cope with different characteristics of backends with different approaches. For operator-level tuning, this work suggests a dynamic programming method with the consideration of the graph topology and backend capability. Without taking into account the effect from cross-kernel optimizations, this tuner efficiently customizes a hybrid execution plan within a few seconds. Optionally, to make up for the potential loss from the relaxation, second tuner may fine-tune the execution plan on the top of the plan customized from the first tuner. Results demonstrate that could extract $1.18\times$ speedup compared to the best backend choice for each workload while outperforming the hand-written approach in the existing deep learning framework by $1.31\times$.

Lastly, Chapter 5 examined the reliability field to evaluate if a compiler can provide a cost-efficient pure software protection scheme. This chapter proposed *RSkip*, which is a novel prediction-based fault protection technology. By employing two approximation methods, this technique could create a cheap approximation copy of a program to verify the integrity of a program execution. If both approximation copy and original copy agree within the acceptance range, expensive re-computation could be bypassed. To maximize the performance while minimizing missed faults, RSkip offered an automatic method to

adjust the aggressiveness of approximation methods according to the run-time context. In evaluation, prior instruction replication work showed $2.33\times$ execution time compared to the unreliable execution over nine compute-intensive benchmarks. However, with the proper use of approximation methods, RSkip could successfully reduce the protection overhead to $1.27\times$ by skipping redundant computation in our target loops at a rate of 81.10%.

6.2 Future Works

Although three important problems are addressed in this thesis, there still remain important challenges to attack:

The first work of this thesis demonstrated the effective optimization decision model can be automatically constructed by learning from the training dataset. Even though its performance significantly outperforms the hand-written heuristics, there are some rooms for improvement. This gap may be caused by either the incomplete feature set or limitation of the current learning approach. Future works may investigate each aspect and endeavor to reduce the gap.

Despite the decent speedup from SRTuner in the second work, it inevitably requires expensive tuning time. To make the holistic tuning approach more practical, the generalization method [70, 34] is necessary to learn the pattern between workload features and good optimization settings and make a flexible recommendation of optimization setting for the unseen workload. Chapter 3.6 discussed the future direction of how this generalization method can be effectively built. By accumulating a number of tuning experiences with SRTuner in the tuning database, the future work will examine what features are necessary to make the right decisions for a group of optimizations and train the recommendation model accordingly to assess the idea.

As the extension of Collage in the third work, heterogeneous hardware devices can be considered. For the fast execution of deep learning models, the state-of-the-arts frameworks often support the heterogeneous executions by employing the multiple hardware devices with different architectural characteristics [9]. This requires accurate decisions of (1) whether it is beneficial to offload the task to other device given the data transfer and synchronization overhead (2) where to offload the computation (3) which software backend should be used for the chosen device. All these three problems are deeply intertwined and form a large and complex search space to explore. Collage can be extended to these challenges. To determine whether to offload the computation, the accurate and efficient cost model is essential and it should be able to consider the data transfer overhead as well. As this can be difficult due to the complexity of modern computing system, learning-based approach may be considered [55]. Also, dynamic programming approach may need to be improved as the

additive relation between the cost of subgraphs may not hold.

Finally, for the fourth part of this dissertation, the wide adoption of various approximation computing methods can be investigated for RSkip. As the idea of prediction-based protection is orthogonal to the approximation techniques, less restricted prediction techniques would help expanding the applicability of RSKip beyond the compute-intensive loops.

BIBLIOGRAPHY

[1] Apple neural engine (ane). https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/. Accessed: 2021-08-25.

[2] Google tensor processing unit (tpu). https://cloud.google.com/tpu. Accessed: 2021-08-25.

[3] Intel onednn. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html#gs.9zgc55. Accessed: 2021-08-27.

[4] Intel openvino. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html. Accessed: 2021-08-27.

[5] Nvidia cublas. https://developer.nvidia.com/cublas. Accessed: 2021-08-05.

[6] Nvidia cudnn. https://developer.nvidia.com/cudnn. Accessed: 2021-08-05.

[7] Nvidia deep learning accelerator (nvdla). http://nvdla.org/. Accessed: 2021-08-25.

[8] Nvidia tensorrt. https://developer.nvidia.com/tensorrt. Accessed: 2021-08-05.

[9] Pytorch. https://pytorch.org/. Accessed: 2021-08-05.

[10] Tensorflow. https://www.tensorflow.org/. Accessed: 2021-08-05.

[11] Tensorflow xla. https://www.tensorflow.org/xla. Accessed: 2021-08-27.

[12] *Intel core i7 embedded processor*, 2019-02-08.

[13] *3DMark*, 2019-06-02.

[14] *FPMark*, 2019-06-02.

[15] *Geekbench*, 2019-06-02.

[16] *SYSmark*, 2019-06-02.

[17] *TabletMark*, 2019-06-02.

[18] *cTuning Foundation*, 2019-11-17.

[19] *GCC online documentation*, 2021-04-11.

[20] *NVIDIA Multi-Process Service (MPS) online documentation*, 2021-04-11.

[21] *GCC*, 2021-08-29.

[22] *LLVM*, 2021-08-29.

[23] *LLVM AMD GPU as a backend*, 2021-08-29.

[24] *LLVM Compiler Passes*, 2021-08-29.

[25] *LLVM NVIDIA GPU as a backend*, 2021-08-29.

[26] *SpiderMonkey Javascript Engine*, 2021-08-29.

[27] *V8 Javascript Engine*, 2021-08-29.

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

[29] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, Technical report, MIT, 2009.

[30] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922–927, 2005.

[31] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Dynamic tolerance region computing for multimedia. *IEEE Transactions on Computers*, 61(5):650–665, 2012.

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

[33] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[34] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):21, 2016.

[35] Todd M Austin. Diva: A reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207. IEEE, 1999.

[36] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[37] Felice Balarin, Paolo Giusto, Attila Jurecska, Michael Chiodo, Harry Hsieh, Claudio Passerone, Ellen Sentovich, Luciano Lavagno, Bassam Tabbara, Alberto Sangiovanni-Vincentelli, et al. *Hardware-software co-design of embedded systems: the POLIS approach*. Springer Science & Business Media, 1997.

[38] Shumeet Baluja. Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, 1994.

[39] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[40] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[41] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[42] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on profile and feedback-directed compilation*, 1998.

[43] Edson Borin, Youfeng Wu, Cheng Wang, Wei Liu, Mauricio Breternitz Jr, Shiliang Hu, Esfir Natanzon, Shai Rotem, and Roni Rosner. Tao: two-level atomicity for dynamic binary optimizations. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 12–21. ACM, 2010.

[44] Shekhar Borkar et al. Microarchitecture and design challenges for gigascale integration. In *MICRO*, volume 37, pages 3–3, 2004.

[45] Greg Bronevetsky, B de Supinski, and Martin Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2009.

[46] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[47] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[48] James Bucek, Klaus-Dieter Lange, et al. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42. ACM, 2018.

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

[50] John Cavazos and J Eliot B Moss. Inducing heuristics to decide whether to schedule. In *ACM SIGPLAN Notices*, volume 39, pages 183–194. ACM, 2004.

[51] John Cavazos and Michael FP O'Boyle. Automatic tuning of inlining heuristics. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 14–14. IEEE, 2005.

[52] John Cavazos and Michael FP O'boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, 41(10):229–240, 2006.

[53] Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, 2005.

[54] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

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

[56] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[57] Kingsum Chow and Youfeng Wu. Feedback-directed selection and characterization of compiler optimizations. In *Proc. 2nd Workshop on Feedback Directed Optimization*, 1999.

[58] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[59] Jack W Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In *International Conference on Compiler Construction*, pages 59–73. Springer, 1996.

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

[61] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing/spl trade/software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 15–24. IEEE, 2003.

[62] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[63] Moslem Didehban and Aviral Shrivastava. nzdc: A compiler technique for near zero silent data corruption. In *Proceedings of the 53rd Annual Design Automation Conference*, page 48. ACM, 2016.

[64] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on computers*, 50(6):529–548, 2001.

[65] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 385–396. ACM, 2010.

[66] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 398–409. ACM, 2011.

[67] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.

[68] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. *ACM SIGPLAN Notices*, 40(7):78–86, 2005.

[69] GG Fursin, Michael FP O'Boyle, and Peter MW Knijnenburg. Evaluating iterative compilation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 362–376. Springer, 2002.

[70] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[71] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, Allen D Malony, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(4):309–329, 2014.

[72] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 98–109. IEEE, 2003.

[73] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[74] Ramaswamy Govindarajan, Erik R Altman, and Guang R Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 85–94. ACM, 1994.

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

[76] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet? In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6546–6555, 2018.

[77] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[78] John Hennessy and David Patterson. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced.

[79] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[80] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[81] Robert W Horst, Richard L Harris, and Robert L Jardine. Multiple instruction issue in the nonstop cyclone processor. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 216–226. ACM, 1990.

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

[83] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 62–72. ACM, 2010.

[84] Michael R Jantz and Prasad A Kulkarni. Exploring single and multilevel jit compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):1–29, 2013.

[85] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[86] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, et al. Miopen: An open source library for deep learning primitives. *arXiv preprint arXiv:1910.00078*, 2019.

[87] Daya Shanker Khudia and Scott Mahlke. Low cost control flow protection using abstract control signatures. In *ACM SIGPLAN Notices*, volume 48, pages 3–12. ACM, 2013.

[88] Daya Shanker Khudia and Scott Mahlke. Harnessing soft computations for low-budget fault tolerance. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 319–330. IEEE, 2014.

[89] Toru Kisuki, P Knijnenburg, M O'Boyle, and H Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44. Citeseer, 2000.

[90] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 237–246. IEEE, 2000.

[91] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[92] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. *ACM Sigplan Notices*, 36(5):156–167, 2001.

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

[94] Prasad A Kulkarni. Jit compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 773–788, 2011.

[95] Prasad A Kulkarni, Michael R Jantz, and David B Whalley. Improving both the performance benefits and speed of optimization phase sequence searches. *ACM Sigplan Notices*, 45(4):95–104, 2010.

[96] Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(1):1–36, 2009.

[97] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91. IEEE Computer Society, 2009.

[98] Hugh Leather, Edwin Bonilla, and Michael O'boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):1–32, 2014.

[99] Shikai Li, Sunghyun Park, and Scott Mahlke. Sculptor: Flexible approximation with selective dynamic loop perforation. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 341–351. ACM, 2018.

[100] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[101] Robert Lim, Boyana Norris, and Allen Malony. Autotuning gpu kernels via static and predictive analysis. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 523–532. IEEE, 2017.

[102] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*, pages 149–158. IEEE, 2018.

[103] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. Mlbench: benchmarking machine learning services against human experts. *Proceedings of the VLDB Endowment*, 11(10):1220–1232, 2018.

[104] Josep Llosa, Mateo Valero, E Agyuade, and Antonio González. Modulo scheduling with reduced register pressure. *IEEE Transactions on computers*, (6):625–638, 1998.

[105] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[106] Niti Madan and Rajeev Balasubramonian. Power efficient approaches to redundant multithreading. *IEEE Transactions on Parallel and Distributed Systems*, 18(8), 2007.

[107] Uma Mahadevan and Lacky Shah. Intelligent loop unrolling, August 18 1998. US Patent 5,797,013.

[108] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. Autotuning stencil-based computations on gpus. In *2012 IEEE international conference on cluster computing*, pages 266–274. IEEE, 2012.

[109] Panagiotis D Michailidis and Konstantinos G Margaritis. Accelerating kernel density estimation on the gpu using the cuda framework. *Applied Mathematical Sciences*, 7(30):1447–1476, 2013.

[110] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *International conference on artificial intelligence: methodology, systems, and applications*, pages 41–50. Springer, 2002.

[111] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*, volume 821. John Wiley & Sons, 2012.

[112] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 99–110. IEEE, 2002.

[113] Shubhendu S Mukherjee, Christopher T Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003.

[114] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[115] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[116] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, pages 1–10, 2018.

[117] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202. IEEE, 2015.

[118] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

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

[120] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

[121] Sunghyun Park, Youfeng Wu, Janghaeng Lee, Amir Aupov, and Scott Mahlke. Multi-objective exploration for practical optimization decisions in binary translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):57, 2019.

[122] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[123] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

[124] Rogier PJ Pinkers, Peter MW Knijnenburg, Masayo Haneda, and Harry AG Wijshoff. Statistical selection of compiler options. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings.*, pages 494–501. IEEE, 2004.

[125] James Price and Simon McIntosh-Smith. Exploiting auto-tuning to analyze and improve performance portability on many-core architectures. In *International Conference on High Performance Computing*, pages 538–556. Springer, 2017.

[126] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S Mukherjee. Perturbation-based fault screening. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 169–180. IEEE, 2007.

[127] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

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

[129] Ari Rasch, Michael Haidl, and Sergei Gorlatch. Atf: A generic auto-tuning framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 64–71. IEEE, 2017.

[130] Archana Ravindar and YN Srikant. Relative roles of instruction count and cycles per instruction in wcet estimation. In *ACM SIGSOFT Software Engineering Notes*, volume 36, pages 55–60. ACM, 2011.

[131] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[132] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*, 2016.

[133] Steven K Reinhardt and Shubhendu S Mukherjee. *Transient fault detection via simultaneous multithreading*, volume 28. ACM, 2000.

[134] George A Reis, Jonathan Chang, and David I August. Automatic instruction-level software-only recovery. *IEEE micro*, 27(1), 2007.

[135] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[136] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68, 2018.

[137] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al.

Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[138] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[139] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79. IEEE, 2008.

[140] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 35–50. ACM, 2014.

[141] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. ACM, 2013.

[142] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The bunker cache for spatio-value approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[143] Vivek Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*, pages 153–166. ACM, 2000.

[144] Nirmal R Saxena and Edward J McCluskey. Dependable adaptive computing systems-the roar project. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 3, pages 2172–2177. IEEE, 1998.

[145] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[146] Timothy J Slegel, Robert MIII Averill, Mark A Check, Bruce C Giamei, Barry W Krumm, Christopher A Krygowski, Wen H Li, John S Liptay, John D MacDougall, Thomas J McPherson, et al. Ibm's s/390 g5 microprocessor design. *IEEE micro*, 19(2):12–23, 1999.

[147] Jared C Smolens, Jangwoo Kim, James C Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 257–268. IEEE Computer Society, 2004.

[148] Tyler Sorensen, Sreepathi Pai, and Alastair F Donaldson. One size doesn't fit all: Quantifying performance portability of graph applications on gpus. In *2019 IEEE*

International Symposium on Workload Characterization (IISWC), pages 155–166. IEEE, 2019.

[149] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, pages 123–134. IEEE Computer Society, 2005.

[150] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM, 2003.

[151] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[152] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[153] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–14. IEEE, 2016.

[154] Cheng Wang and Youfeng Wu. Tso_atomicity: efficient hardware primitive for tso-preserving region optimizations. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 509–520. ACM, 2013.

[155] Nicholas J Wang and Sanjay J Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

[156] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 2018.

[157] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 411–420. IEEE, 2001.

[158] Vincent M Weaver, Dan Terpstra, Heike McCraw, Matt Johnson, Kiran Kasichayanula, James Ralph, John Nelson, Phil Mucci, Tushar Mohan, and Shirley Moore. Papi 5: Measuring power, energy, and the cloud. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 124–125. IEEE, 2013.

[159] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrich Meyr. Fridge: An interactive code generation environment for hw/sw codesign. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 287–290. IEEE, 1997.

[160] Wayne H Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, 1994.

[161] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[162] Ying C Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293–307. IEEE, 1996.

[163] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. *ACM SIGPLAN Notices*, 38(5):63–76, 2003.

[164] Xinchuan Zeng and Tony R Martinez. Distribution-balanced stratified cross-validation for accuracy estimation. *Journal of Experimental & Theoretical Artificial Intelligence*, 12(1):1–12, 2000.

[165] Guoqiang Peter Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, 2000.

[166] Yun Zhang, Jae W Lee, Nick P Johnson, and David I August. Daft: decoupled acyclic fault tolerance. *International Journal of Parallel Programming*, 40(1):118–140, 2012.

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

[168] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924*, 2020.

[169] James F Ziegler and Helmut Puchner. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress, 2004.

[170] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. 2018.