THE UNIVERSITY OF CHICAGO

CROSS-LAYER APPROXIMATE COMPUTING: FROM CIRCUITS TO ALGORITHMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
SAEID BARATI

CHICAGO, ILLINOIS
MARCH 2020

TABLE OF CONTENTS

LIST OF FIGURES . . . v

LIST OF TABLES . . . vii

ACKNOWLEDGMENTS . . . viii

ABSTRACT . . . x

1 INTRODUCTION . . . 1
 1.1 Thesis Statement . . . 1
 1.2 Problem Description . . . 1
 1.3 Contributions . . . 2
  1.3.1 VIPER . . . 2
  1.3.2 BOA . . . 3
  1.3.3 MAMBA . . . 3
  1.3.4 COBRA . . . 4
  1.3.5 Reduced-precision FPU . . . 4
  1.3.6 Holistic Cross Layer Approximation Framework . . . 5

2 VISUALIZING APPROXIMATE COMPUTING FRAMEWORKS . . . 6
 2.1 Terminology . . . 6
 2.2 Motivation . . . 7
 2.3 Numerical Comparisons . . . 7
 2.4 Design . . . 10
 2.5 Evaluation . . . 12
  2.5.1 Experimental Setup . . . 12
  2.5.2 Comparison by Difference of Coverage . . . 14
  2.5.3 Comparison by Pareto-optimal Curves . . . 14
  2.5.4 Comparison by VIPER . . . 14

3 BOA AND SOFTWARE APPROXIMATION . . . 17
 3.1 Background and Motivation . . . 17
 3.2 Design . . . 19
  3.2.1 BOA-Simple . . . 20
  3.2.2 BOA-Flex . . . 20
  3.2.3 BOA-Prob . . . 21
 3.3 Evaluation . . . 22
  3.3.1 Experimental Setup . . . 23
  3.3.2 Points of Comparison . . . 23
  3.3.3 Comparison by Difference of Coverage . . . 24
  3.3.4 Comparison by VIPER . . . 25
  3.3.5 Exploration Time . . . 28

3.3.6 Input Sensitivity 29
3.3.7 Comparison to Exhaustive Exploration 30
3.3.8 Combination Distribution 32

4 MAMBA AND PROGRAMMING LANGUAGE APPROXIMATION 34
4.1 Background and Motivation 34
4.2 Design 37
4.2.1 MAMBA Inputs 38
4.2.2 MAMBA Outputs 42
4.3 Interface and Runtime Manual 43
4.4 Evaluation 45
4.4.1 Experimental Setup 46
4.4.2 Floating Point Precision Distribution 47
4.4.3 FPU Energy Saving 47
4.4.4 Memory Instructions 51
4.4.5 Flexible Precision Level 52
4.4.6 Function Call Stack 53
4.4.7 Input Sensitivity 53
4.4.8 Neural Network Integration 55

5 COBRA AND ARCHITECTURAL APPROXIMATION 59
5.1 Background and Motivation 59
5.2 Design 63
5.3 Interface and Runtime Manual 67
5.4 Evaluation 69
5.4.1 Experimental Setup 69
5.4.2 Tradeoff Spaces and Lower Convex Hull 70
5.4.3 Energy Efficiency and BFP 70
5.4.4 Energy Efficiency and PBQ 71
5.4.5 Accuracy Variation and Protected Bits 71

6 HARDWARE AND CIRCUIT APPROXIMATION 73
6.1 Background and Motivation 73
6.2 System Design 75
6.3 Evaluation 77
6.3.1 FPU Energy and Accuracy Loss 78

7 TOWARDS CROSS-LAYER APPROXIMATE COMPUTING 79
7.1 Background and Motivation 79
7.2 Design 81
7.3 Evaluation 82
7.3.1 Evaluation Setup 82
7.3.2 Comparison by Size of Search Space 83
7.3.3 Comparison by Configuration Space 83

7.3.4 Comparison in BOA Family 84
7.3.5 Combination Distribution 86

8 FUTURE WORK 87

LIST OF FIGURES

2.1 Dominance (a) and Coverage (b) Functions (from [71]). In (a) x_2 dominates x_3 as x_2 is both faster and more accurate. In (b), X covers 2/3 of Y because y_2 and y_3 are dominated by at least one point in X. 9
2.2 Performance/AccuracyLoss tradeoff spaces for PowerDial, Loop Perforation, and Approximate Math Library. 15
2.3 VIPER comparison of PowerDial, Loop Perforation, and Approximate Math Library. 16

3.1 Difference of coverage over NSGA-II. Higher bars are better. 24
3.2 Pareto-efficient curves found by each exploration technique. 25
3.3 VIPER comparison of MCKP, NSGA-II, and different versions of BOA. 26
3.4 Comparison of MCKP, NSGA-II, and variations of BOA with exhaustive exploration. (a) shows comparison by Pareto-efficient frontiers and (b) shows comparison by VIPER. 31

4.1 Energy Per Instruction for different classes of instructions. (From [64]) 36
4.2 MAMBA design. 37
4.3 FCS placement considers FFT function call stack before selecting the approximate FPI. 41
4.4 Floating-point type breakdown for benchmarks. While most benchmarks have a dominant FP type, some carry both. 48
4.5 Lower convex hulls of FPU energy and error rates for the WP and CIP. Values are normalized to the baseline. 49
4.6 FPU energy savings at different error rates, normalized to the baseline. Higher the bars, the more energy efficient is. 50
4.7 Memory transfer energy savings at different error rates, normalized to the baseline. 51
4.8 FPU energy savings with different optimization targets for MAMBA. 52
4.9 Comparison of CIP and FCS for the FPU energy savings in radar. 54
4.10 32-bit FLOP breakdown per layer in digit recognition CNN. 57
4.11 Comparison of PLC and PLC replacements for the CNN. (a) Lower convex hull curves of energy and error rate. (b) Quantized energy savings at different error rates. 57

5.1 SRAM cell failure probability under voltage overscaling in 28nm process technology. (From [33]) 62
5.2 Unreliable memory and quality of output in canny application. 62
5.3 COBRA design. 65
5.4 Sample code snippet of using COBRA for data partitioning. 68
5.5 Tradeoff space of inExact cache on benchmarks. 70
5.6 Energy efficiency at various supply voltages and bit protections. 71
5.7 Controlling unpredicted behavior of inExact cache with bit protection. 72

6.1 Applying bitwidth reduction in the source sode of openPiton. 76

6.2 Loaded OpenPiton processor on the Kintex FPGA. 77
6.3 FPU energy and accuracy loss of selected benchmarks on reduced-precision FPU. 78

7.1 Cross-layer AC with BOA. 82
7.2 Tradeoff space and Pareto-efficient curves of NSGA-II and BOA-simple in the cross-layer AC. 83
7.3 Lower convex hulls of BOA variations in cross-Layer AC. 84
7.4 VIPER plots of BOA variations in cross-Layer AC. 85

LIST OF TABLES

2.1 Benchmarks used for evaluation. 13

3.1 Number of explored configurations. 18
3.2 Correlation coefficients for accuracy loss. 30
3.3 Correlation coefficients for normalized runtime. 31
3.4 Number of used configurations from each approximation framework. 32

4.1 Built-in placement rules in MAMBA. 40
4.2 Benchmarks used for evaluation. 47
4.3 Correlation coefficients for error rates and FPU energy. 54
4.4 LeNet-5 architecture summary. 56
4.5 Mantissa bits For single precision FP recommended by MAMBA for each layer at different error rates. 56

5.1 Host system architectural parameters. 69

6.1 Power consumption of single precision FP multiplier (From [91]) 75

7.1 Number of explored configurations in cross-layer AC combination. 83
7.2 Unique configurations from each individual AC framework found on the Pareto-efficient curve. 86

ACKNOWLEDGMENTS

I wish to thank all the people whose assistance was a milestone in the completion of this dissertation.

My family... Your unconditional love, support, and encouragement from far far away made it possible for me to walk through all the hardships. I wish you were here to celebrate this together. I can only wish for a world without borders.

I could not have finished this PhD program if it wasn't for exceptional help and support of my advisor, Hank Hoffmann. Many times during the past 6 years, I wanted to quit, give up, and leave the program, but every single time, he assured me it's going to be alright. Not only an academic advisor, but a life coach, a true friend, and many more. Thanks for believing in me, teaching and mentoring me and making it possible to reach the end.

I would like to express my gratitude towards my partner in crime, Bahar. Crossing the finish line wouldnt have been this easy if it wasnt for your support and love.

I would like to thank faculty members at computer science department at the University of Chicago, specifically Professors Shan Lu, Fred Chong and Haryadi Gunawi, for being on my PhD and Masters committee.

My awesome friends in UChicago. Thanks for being there for me, learning together, laughing together, and making grad life enjoyable. Dr Connor Imes, for all those medici and grounds of being walks, for teaching me the softball (I will always remember to CHOP IT DOWWWWWN! when batting), for homebrews, volleyballs and many more. Dr Harper Zhang and all the Xbox NBA 2K games. Dr Gokalp Demirci for the basketball and soccer games. A special word of gratitude to my lab mates, Ivana Marincic, Will Kong, Anne Farrel, Ahsan Parviz, Yuliana Zamora, Bernard Dickens. I will always remember our tea and coffee breaks and long chats. Special thanks to Kavon Farvardin (and his adorable kitties, JiJi and Lil' Pumpkin) for being the best roommates ever. Also, Brian Hempel, Joe Wingerter, Suhail Rehman for all those PSD happy hour talks and board games.

I'm extremely grateful for my friends from humanities who reminded me that there is more to PhD life than just doing CS research. Thanks to Alexandra Hoffmann, Shaahin Pishbin, Xelef Botan, Allison Kanner, Arlen Wisental, Siavash Sabetrohani and many more for all those Kabobi therapy trips, delicious pastries and our weekly little Sofreh.

The department technical and administrative staff, especially Bob Bartlett, Margaret Jaffey, Nita Yack, and Sandy Quarlesnothing would ever get done with you, you are all awesome!

Finally, for those not mentioned here, I apologize if I miss your name. I appreciate each and every interaction with all of you and that I believe makes who I am today. So thank you! I hope, while you all making my life better, I have made a tiny positive changes in your life too and I am definitely looking forward to our path crossed in the future.

ABSTRACT

Approximate computing frameworks configure applications so they can operate at a range of points in an accuracy-performance tradeoff space. Approximations at layers of system stack provide computation and energy efficiency via lower resource usage by allowing tolerable errors.

Instead of overprovisioning the resources at each layer, higher efficiency is achievable if we consider the system stack holistically. This thesis evaluates cross-layer approximate computing. We explore three problems: 1) How to compare approximate computing frameworks in terms of accuracy-efficiency tradeoffs? 2) How to combine them without negating each others optimizations? 3) How to achieve significant end-to-end optimizations when the entire system stack is considered?

We address the first problem by providing an improved visualization algorithm to represent tradeoff spaces. For the second problem, we analyze the approximation framework in layers of system stack one by one to evaluate end-to-end approximation opportunities. First, we introduce a family of exploration techniques to combine software approximation frameworks. Next, at the programming language layer, an automated precision tuning framework is presented that helps users explore different levels of floating-point approximations in the application. At the architecture level, an unreliable memory framework is implemented and examined. At the hardware and circuit layer, precision tuning information is used to build a reduced-precision FPU.

For the third problem, we form the first holistic cross-layer approximate computing framework which integrates approximations at various layers and provides performance and energy improvements in return for minimal accuracy loss.

CHAPTER 1

INTRODUCTION

1.1 Thesis Statement

This dissertation explores cross-layer approximate computing, from software and algorithms to hardware and circuits. A visualization tool to compare approximation frameworks and a family of exploration algorithms to combine them in a single layer of computing stack is introduced. Moving forward, more frameworks from other layers are included to form a holistic platform to perform end-to-end optimizations.

1.2 Problem Description

Approximate Computing (AC) tradeoffs quality of output with diverse optimizations such as higher performance or lower resource usage. Due to the existence of noise and redundancy in real world inputs, perceptual limitations of human senses, and error attenuation characteristics of processing algorithms, approximate computing techniques are able to improve efficiency by trading precision, accuracy, or reliability.

Approximation opportunities exist across every layer of the system stack, focusing on energy efficiency, enhancing thermal profile, and performance improvement. We broadly classify AC techniques into four main categories.

1. **Operating Systems and Software** aims at reducing the computation by exploiting dynamic information about the runtime environment. These techniques primarily trade use of sensors, displays, and coarse-grained error behavior for improved efficiency.

2. **Programming Languages and Compilers** provide abstractions for the errors induced from lower layers to affect computations mostly. Floating-point optimization, code perforation, relaxed parallelization, and bitwidth reductions represent PL and compiler approximations.

3. **Architecture and micro-Architecture Designs** emphasize on reducing resource usage (chip area for processors or density for memories) through specialized instructions and instruction extensions. Imprecise instruction set architecture (ISA), neural processing units, and unreliable memory systems belong to architecture layer approximation.

4. **Circuits design and Accelerators** mainly target power efficiency, and to some lesser extent performance. At the hardware level, approximation happens via timing based on overscaling voltage or designing inExact functional components.

As AC proliferate, it is natural to ask their interaction. We address three of these questions.

- How to *compare* approximate computing frameworks with each other regardless of their target layer in the system stack?
- How to *combine* different techniques tradeoff spaces and locate Pareto-efficient points in the new space?
- How to *bridge the gap* in system stack layers for enabling cross-layer approximate computing?

1.3 Contributions

This dissertation addresses the cross-layer AC by introducing the following frameworks:

1.3.1 VIPER

We address the problem of comparing approximate frameworks by proposing VIPER: Visualizing Improved PErformance Ratios. While existing techniques use numerical comparisons [7, 63, 107] or simply display Pareto-optimal curves, VIPER produces a visual representation of the tradeoff space. VIPER produces charts showing normalized performance improvement ratios for different frameworks across all possible accuracy loss ranges. A chart is divided

into different, mathematically meaningful regions that show how much one framework outperforms another.

1.3.2 BOA

To merge complementing AC techniques, we propose BOA: Blending Optimal Approximations for combining approximate frameworks. BOA is a family of algorithms that locate Pareto-efficient points in the huge tradeoff space produced by multiple approximation frameworks. In its simplest version, BOA-simple searches the cross product of Pareto-optimal points from individual frameworks. More advanced versions of BOA extend the search space either deterministically or probabilisticallyto include more near-optimal points in the search space. BOA then returns the Pareto-efficient points from this search space.

1.3.3 MAMBA

The key to bridging the gap between layers is to provide a wide-range of accuracy vs efficiency configurations which is understandable by approximate components at different layers. For instance, the escalation of both different approximate functional units (hardware/circuit layer) and reduced-precision software methods (Operating System layer) creates tremendous opportunity [19, 31, 78], but it also creates a new problem. How do programmers decide which level of approximation to use at different points in their application?

We propose MAMBA: Mechanisms for Applying Multi Bit Approximations, tool that helps users explore different levels of approximation from underlying layers at software level without requiring implementing customized hardware or detailed instrumentation of the application. MAMBA does not require users to have domain knowledge specifics to meet the accuracy requirements. MAMBA accepts a user program, a set of approximate floating-point implementations, and a set of programmable placement rules for when to use a specific implementation. MAMBA then runs the program and dynamically replaces floating-point

operations (FLOPs) with the approximate version as specified by the rules. MAMBA can be used as a tool to explore the tradeoff space of floating-point implementations (FPI) without requiring deep numerical expertise.

1.3.4 COBRA

At the architecture level, the motivation of approximation is conducted towards power or memory efficiency. Memory system approximation trade energy efficiency with quality of output by changing individual memory cells [36, 37, 47] or modifying the memory hierarchy[66, 74, 90]. COBRA: Cache Optimization with Bit Reversal Approximation is an unreliable data cache providing energy efficiency at the memory system. COBRA injects the fault to noncritical data whose approximation would not result in application failures or crashes. Bit Flip Probability and higher order bit protection mechanism are two approximation knobs that COBRA exposes to users. COBRA then runs the application, enforces approximation to noncritical data, and measures data cache performance and energy consumption stats.

1.3.5 Reduced-precision FPU

To appraise cross-layer approximation, the proposed technique should be evaluated end-to-end on either actual state-of-the-art hardware or realistic simulations[89].

Upper layers of the computing stack have the unique advantage of the global view of the system hardware, and visibility into application behavior through the execution period. If we convoy runtime and compile the information into the hardware and architecture designs directly, a much higher efficiency should be expected while minimizing the quality loss.

To this end, we adopt floating-point analysis from MAMBA to build a reduced-precision FPU at the hardware level. We use OpenPiton [10], a general-purpose, multithreaded, many-core processor which is open source across the entire computing stack, from the hardware to the firmware and software. Hence, we profile the benchmarks using the MAMBA to find out

the most efficient floating-point implementation. Then, we synthesize the processor to deliver the most efficient approximation level. This is a case study for the actual implementation of an end-to-end approximation solution.

1.3.6 Holistic Cross Layer Approximation Framework

Most of the existing approximation techniques are designed in the context of a single layer in the system stack. Although, different AC frameworks could negate or complement the efficiency of each other. We believe that an end-to-end approximation could extend the accuracy-efficiency tradeoffs further. Hence, to enable the cross-layer approximate computing, we need to provide a systematical understanding of the relationship between ACs and system stack layers.

To this means, the efficiency-accuracy tradeoff spaces should be converted to comprehensible facts that can be translated to facilitating flexible designs at other layers. Also, the analysis of error propagation and data-driven resilience is necessary. While there has been an extensive effort for building approximate computing frameworks at various layers, no one has managed to build an end-to-end comprehensive evaluation on an actual platform for the cross-layer approximate computing. Our solution provides an inclusive platform of approximation opportunities for an application across the layers of the system stack.

CHAPTER 2

VISUALIZING APPROXIMATE COMPUTING FRAMEWORKS

This chapter describes VIPER (Visualizing Improved PErformance Ratios). Chapter 2 starts by explaining the terminology used throughout this thesis and then introduces current comparison techniques (numerical [7, 63, 107] and visual). Next, the backend structure of VIPER is discussed followed by a case study of comparing three approximate computing frameworks. VIPER is a generic scalable comparison tool that illustrates the most optimal approximation technique at first sight.

2.1 Terminology

AC frameworks allow applications to operate across a range of performance and accuracy tradeoffs. To produce this range, any approximation framework must have one or more tunable *parameters*. The values assigned to the parameter set represent a *configuration* and the range of possible parameter settings is a *configuration space*. Each configuration represents a *tradeoff* between performance and accuracy. The *tradeoff space* (or *design space*) is the set of all possible tradeoffs; *i.e.,* the range of achievable performance and accuracy. While it is easy to compare individual tradeoffs produced by individual configurations, it is less obvious how to compare tradeoff spaces induced by two separate configuration spaces.

This dissertation deals with large search spaces and many times we do not know the true optimal values for which we are searching. We therefore distinguish between the term *Pareto-optimal* – meaning we know that a point is on the true Pareto-optimal frontier – and *Pareto-efficient* – which means it is a point on the current best estimate of the unknown Pareto-optimal frontier. In other words, if we say a point is Pareto-efficient, we know it is better than all other points we have found so far, but we do not know that it is truly

Pareto-optimal.

2.2 Motivation

Approximate Computing Frameworks produce accuracy-efficiency tradeoff space. Each point in these tradeoff spaces corresponds to configuration of the approximation framework. Point-by-point comparison is unfeasible since tradeoff spaces include numerous configurations, many of which are not useful. Typically, only the Pareto-optimal points are used for comparison.

Comparison of approximation frameworks across their full range of accuracy is necessary, as not all users have the same accuracy requirements. We find, however, that looking at Pareto-optimal curves is unsatisfying and rarely makes it obvious which approximation method is better across a range of operating points. Moreover, while for a certain range of accuracy one framework might perform better, another framework might produce a higher performance at different accuracy ranges. Thus, we need to compare the approximation frameworks across their full range of accuracy. This motivates VIPER, a tool that allows users to tell—at a glance—which framework has the best performance for any range of accuracy loss. Knowing which framework is performing better at any accuracy range allows users to choose the most efficient framework if they want to use multiple frameworks at the same time.

2.3 Numerical Comparisons

For large tradeoff spaces, it is not possible to do a point-by-point comparison. Therefore, prior work has introduced analytical methods for comparing tradeoff spaces based on the number of Pareto-optimal—if the tradeoff space is known—or Pareto-efficient—if the tradeoff space is estimated—points found by each framework.

A point in our accuracy-performance tradeoff space is a 2D vector with *runtime* and *accuracyLoss*. Ideally, we would like to achieve zero run time and zero accuracy loss; *i.e.,* instantaneously get a perfect answer. With that goal in mind, we present:

Definition 1. *Objective Function: Given points x_1 and x_2 in the tradeoff space the objective function to be minimized is $f(x)$ where:*

$$\begin{aligned} f(x_1) < f(x_2) \iff &\ accuracyLoss(x_1) < accuracyLoss(x_2) \\ &\ \&\ runtime(x_1) < runtime(x_2) \end{aligned} \tag{2.1}$$

In our experiments, points closer to the origin represent more efficient configurations. Given the objective function $f(x)$, we determine if a point is more efficient than another by

Definition 2. *Dominance: Given points x_1 and x_2, we say:*

$$\begin{aligned} x_1 \succeq x_2 \text{ (weakly dominates) if } f(x_1) \leqslant f(x_2) \\ x_1 \succ x_2 \text{ (dominates) if } f(x_1) < f(x_2) \end{aligned} \tag{2.2}$$

A point is Pareto-optimal if it is not dominated by any other point. A point is Pareto-efficient if we do not know of another point that dominates it. Figure 2.1(a) illustrates an example of dominance where point x_3 is dominated by point x_2 in the tradeoff space.

Prior work defines *coverage* to compare the Pareto-efficient points produced by different design techniques [71]

Definition 3. *The* coverage function *is the dominance ratio of the Pareto-efficient curves induced by two separate frameworks. If X and Y are two Pareto-efficient curves, and x and y represent points on them respectively, then:*

$$C(X, Y) = \frac{|\{\, y \in Y \mid \exists x \in X : x \succeq y \,\}|}{|Y|} \tag{2.3}$$

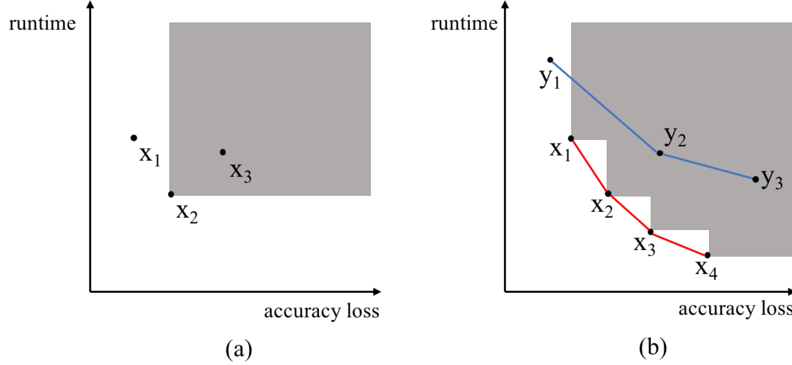


Figure 2.1: Dominance (a) and Coverage (b) Functions (from [71]). In (a) x_2 dominates x_3 as x_2 is both faster and more accurate. In (b), X covers 2/3 of Y because y_2 and y_3 are dominated by at least one point in X.

$C(X,Y) = 1$ means that all points in Y are weakly dominated by points in X; *i.e.,* all points of X provide lower runtime for the same accuracy loss than the points of Y.

Figure 2.1(b) illustrates the coverage of curve X with respect to curve Y. The point y_2 and y_3 on the Y curve are dominated by at least one point on the curve X—x_2, for example—therefore $C(X,Y) = \frac{2}{3}$. In contrast, no point on X is dominated by a point on curve Y which means $C(Y,X) = 0$. By this metric, we consider X more efficient, but note that the curve Y extends through a larger range within the tradeoff space;*i.e.,* y_1 is a useful point (has lowest accuracy loss) which neither dominates nor is dominated by any points in X. The coverage function is non-symmetric ($C(X,Y) \neq C(Y,X)$) and usually their sum does not equal 1 [44]. Hence, we need a metric that considers both coverage functions simultaneously.

Definition 4. Difference Of Coverage *compares coverage for two different Pareto-efficient curves.*

$$DOC(X,Y) = C(X,Y) - C(Y,X) \tag{2.4}$$

As a result, when $DOC(X,Y) \geq 0$, that fraction of Y points which are dominated by X is greater than X points that are dominated by Y. In other words, the DOC demonstrates how many more points in X dominate points in Y. Higher DOC implies one set is more

Algorithm 1 VIPER computes the Performance Improvement Ratio.

Require: M, B ▷ Lower convex hulls for framework M and baseline B

1: $MinX = Max(Min(M.x), Min(B.x))$ ▷ lower bound
2: $MaxX = Min(Max(M.x), Max(B.x))$ ▷ upper bound
3: $step = (MaxX - MinX)/1000$
4: **for** $accuLoss = MinX; accuLoss < MaxX; accuLoss+ = step$ **do**
5: $M_i \leftarrow$ find point on M where $M_i.x < accuLoss < M_{i+1}.x$
6: $B_j \leftarrow$ find point on B where $B_j.x < accuLoss < B_{j+1}.x$
7: $\hat{y}_M \leftarrow$ interpolate runtime between M_i and M_{i+1} where $x = accuLoss$
8: $\hat{y}_B \leftarrow$ interpolate runtime between B_j and B_{j+1} where $x = accuLoss$
9: $perfImprovRatio[accuLoss] = \hat{y}_M / \hat{y}_B$
10: **end for**
11: NORMALIZE($perfImprovRatio$) ▷ limit the ratio to [0,1]
return $perfImprovRatio$ ▷ array of points

efficient than the other. If $DOC(X, Y)$ is close to zero, it interprets as both may provide same level of efficiency. This metric is widely used by in Pareto-based genetic algorithms to determine convergence of genetic evolution [7, 62, 63].

2.4 Design

Prior studies used the Pareto-optimal curves to show how good an approximation framework can be, however comparing curves is complicated [35, 51]. While curves may look compact, they can be different by orders of magnitude. For example, when there is a steep slope on the curve and a large range covered, a small change in one dimension (*e.g.* accuracy loss) leads to a significant shift in the other (*e.g.* runtime). In addition, numerical metrics suffer from two major shortcomings. First, these metrics do not show the full range of accuracy loss induced by each approximation framework. Second, often the best approximation framework varies as accuracy loss range changes. Numerical metrics—like DOC—have limited expressiveness; Figure 2.1(b) shows that y_1 is a useful point, but DOC makes X look uniformly better than Y. To provide a simple and convenient analysis of approximation frameworks, we introduce VIPER that illustrates the speedup over a baseline framework for any range of accuracy loss.

Algorithm 1 explains how VIPER calculates the *performance improvement ratio* (PIR) of one framework M over a baseline B. The lower convex hull includes the Pareto-optimal

points that are a part of the convex hull. VIPER receives the lower convex hulls of the frameworks M and B as the input. By definition, the straight line between any two points on the convex hull stays in the convex hull. Therefore, we use the lower convex hull rather than Pareto-optimal points to be able to interpolate runtime for the specific accuracy loss that is achievable by the approximation framework.

First, VIPER finds the lower and upper bounds for the accuracy loss which defines the range of comparison. Then, this range gets divided by a parameterized granularity. We use a granularity of 1000 for all results in this paper as larger values produced no benefit and smaller values make the charts less clear. For each *accuLoss* value, VIPER estimates the corresponding runtime in both frameworks. Afterwards, we search for the nearest points on each lower convex hull where their accuracy loss is smaller than *accuLoss* (identified with M_i and B_j points respectively). Then, we interpolate the runtime of the specified *accuLoss* for both frameworks (named as $\hat{y}_M$ and $\hat{y}_B$), and divide these interpolated *runtime* values to compute the performance improvement ratio. Finally, we normalize the ratio to the lowest and highest calculated performance improvement ratios to limit the ratio to [0,1]. When we compare multiple lower convex hulls at the same time, the performance improvement ratio is normalized to the lowest and highest ratio among all hulls.

If the line for M stays above that for B for a greater range of accuracy loss, it means M has found more efficient configurations, on average. The color shading on the plot background indicates the highest performance method for that accuracy loss from the multiple frameworks. Therefore, if the plots background is dominated by a single color, the corresponding method provides the more efficient (higher performance at the same accuracy loss) configuration. Thus, VIPER allows users to see at a (literal) glance, whether one approximation framework is clearly superior to another.

2.5 Evaluation

As an example, we compare three Approximate Computing Frameworks: PowerDial [43], Loop Perforation [87], and the Approximate Math Library [54]. We pick these three frameworks because (1) they are either easily recreated or publicly available, requiring no specialized language or hardware support, and yet (2) they are representative of approaches applied at different levels of the system stack. PowerDial is an application level approach that exploits existing tradeoffs envisioned by the application developers. Loop Perforation creates approximate applications by applying a compiler transformation to selectively skip loop iterations. The Approximate Math Library changes computation, and while implemented in software, it is a good proxy for approximation techniques that change hardware arithmetic units.

2.5.1 Experimental Setup

We use a dual socket Intel Xeon E5 server system with 20 physical cores at 2.9 GHz, hyperthreading, and 32 GB memory. Table 2.1 lists the used benchmarks, from Parsec 3.0 [13] and Rodinia 3.1 [18]. Table 2.1 also contains the description, type, application accuracy metric, and default runtime for each benchmark. Accuracy loss is the error relative to the most accurate configuration. Shorter runtime interprets as higher performance. The `blackscholes`'s only tunable parameter is the number of prices to estimate and modifying it does not affect accuracy. Thus, PowerDial has no effect on `blackscholes`. Similarly, `canneal`, `heartwall`, `kmeans`, and `x264` use math functions infrequently; the Approximate Math Library is not applicable to them.

We evaluate each suite across multiple inputs and compare the median across the inputs for this evaluation. In this section, we are evaluating known frameworks, thus we use the training inputs from Table 2.1. In subsequent sections—where we present new techniques—we divide inputs into training and test and build combinations of frameworks using the

Table 2.1: Benchmarks used for evaluation.

Benchmarks	Accuracy Metric	Training inputs	Test Inputs	Runtime (sec)
Blackscholes	Average Relative Error of Prices	30 lists with 1M initial prices	90 lists with 1M initial prices	3.2
Bodytrack	Average Distance of Poses	sequence of 100 frames	sequence of 261 frames	3.1
Canneal	Average Relative Error of Routing Cost	30 netlists with 400K+ elements	90 netlists with 400K+ elements	6.88
Fluidanimate	Distance between Particles	5 fluids with 100K+ particles	15 fluids with 500K+ particles	17.2
Heartwall	Average Relative Error of heart frames	sequence of 30 ultrasound images	sequence of 100 ultrasound images	11.6
Kmeans	Distance between Cluster Centers	30 vectors with 256K data points	90 vectors with 256K data points	3.1
Particlefilter	Distance between Particles Coordinates	sequence of 60 frames	sequence of 240 frames	12.9
Srad	Image Diff (RMSE)	3 images with 2560*1920 pixels	9 images with 2560*1920 pixels	22.6
Streamcluster	Distance between Cluster Centers	3 streams of 19k-100K data points	9 streams of 100K data points	30
Swaptions	Average Relative Error of Prices	40 swaptions	160 swaptions	6.2
x264	Average Relative Error of PSNR+Bitrate	4 HD videos of 200+ frames	12 HD videos of 200+ frames	7.7

training data and then use the test data to ensure our selected combination works well on previously unseen data.

We evaluate three approximation frameworks. PowerDial (PD) transforms an application's command line parameters into *software knobs* that are automatically manipulated to trade accuracy for performance [43]. Each application has tunable *knobs*, which can take different values, and an assignment of values to knobs is a configuration. Loop Perforation (LP) identifies *perforatable loops* whose iterations can be skipped to produce faster, but less accurate results [87]. A set of loops and perforation rates is a configuration. The Approximate Math Library (AML) substitutes math functions with a variable Taylor series expansion. A set of functions and their number of terms is the configuration.

2.5.2 Comparison by Difference of Coverage

To compare Loop Perforation and PowerDial, we calculate the average coverage function ($C(X, Y)$ from Eq. 3) across all benchmarks for both. Higher values of $C(X, Y)$ implies the greater coverage of framework X over framework Y. On average, Loop Perforation covers only 36.64% of Pareto-optimal points of PowerDial, while PowerDial covers 44.16% of Pareto-optimal points of Loop Perforation. Thus, $DOC(LoopPerforation, PowerDial) = -0.075$ which shows the slight superiority of PowerDial over Loop Perforation, on average. This means employing PowerDial provides higher performance and higher accuracy on average across all benchmarks. On the other hand, negative values of $DOC(AML, PD) = -0.9135$ and $DOC(AML, LP) = -0.929$ shows the significant inferiority of Approximate Math Library against other frameworks, on average.

2.5.3 Comparison by Pareto-optimal Curves

Figure 2.2 illustrates the frameworks' tradeoff spaces. Each point represents a configuration. The y-axis is runtime normalized to the default configuration and the x-axis is the accuracy loss. Each framework's Pareto-optimal curve is shown in the same color as the tradeoff space. These plots highlight how configurations cover wide ranges of runtime and accuracy loss. While in some cases—*e.g.* `canneal`, `kmeans`, and `srad`—Pareto-optimal curves are easily comparable, in other benchmarks—*e.g.* `particlefilter` and `swaptions`—comparison is unfeasible. For `fluidanimate`, the Pareto-optimal curves intersect multiple times; the best approximation framework differs across the range of accuracy loss.

2.5.4 Comparison by VIPER

Just viewing the Pareto-optimal curves in Figure 2.2 provides limited intuition on which framework is better, because differences are not always visible. Hence, we use VIPER to compare these frameworks in Figure 2.3. The y-axis represents the performance improvement

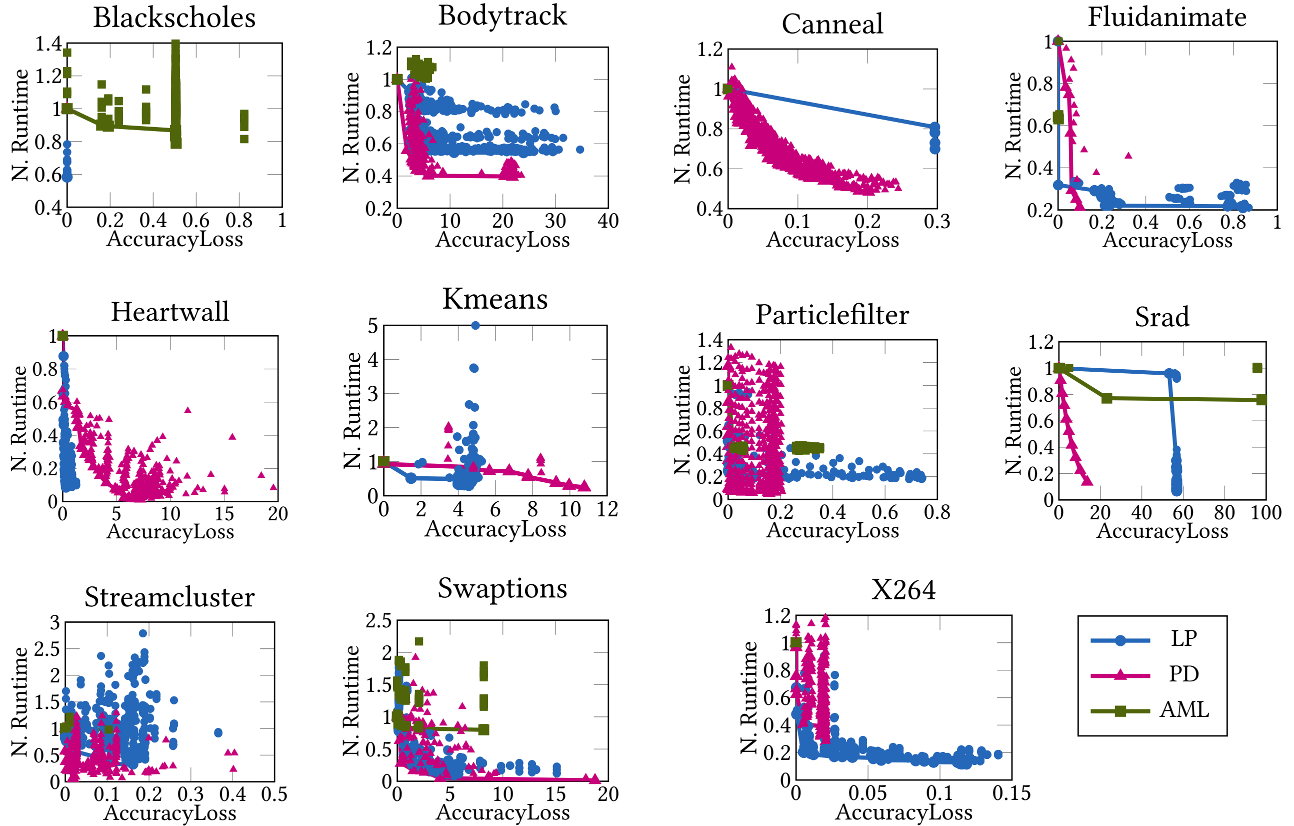


Figure 2.2: Performance/AccuracyLoss tradeoff spaces for PowerDial, Loop Perforation, and Approximate Math Library.

ratio (PIR), while the x-axis illustrates accuracy loss. The horizontal line represents Loop Perforation: points above that line mean the corresponding technique is faster than Loop Perforation. The backdrop color indicates the best method for any range of accuracy loss.

Figure 2.3 provides limited intuition on the superiority of one AC over others while VIPER plots with backdrop color indicate the best method for any range of accuracy loss.

Since VIPER only requires tradeoff spaces to compare, it can be applied to any type of approximation framework regardless of system level. VIPER provides the following insights:

- It illustrates how approximation frameworks perform within a specific accuracy loss range. For instance, while PowerDial finds a higher performance configuration than Loop Perforation for `bodytrack` and `canneal`, its performance is worse for `kmeans` and

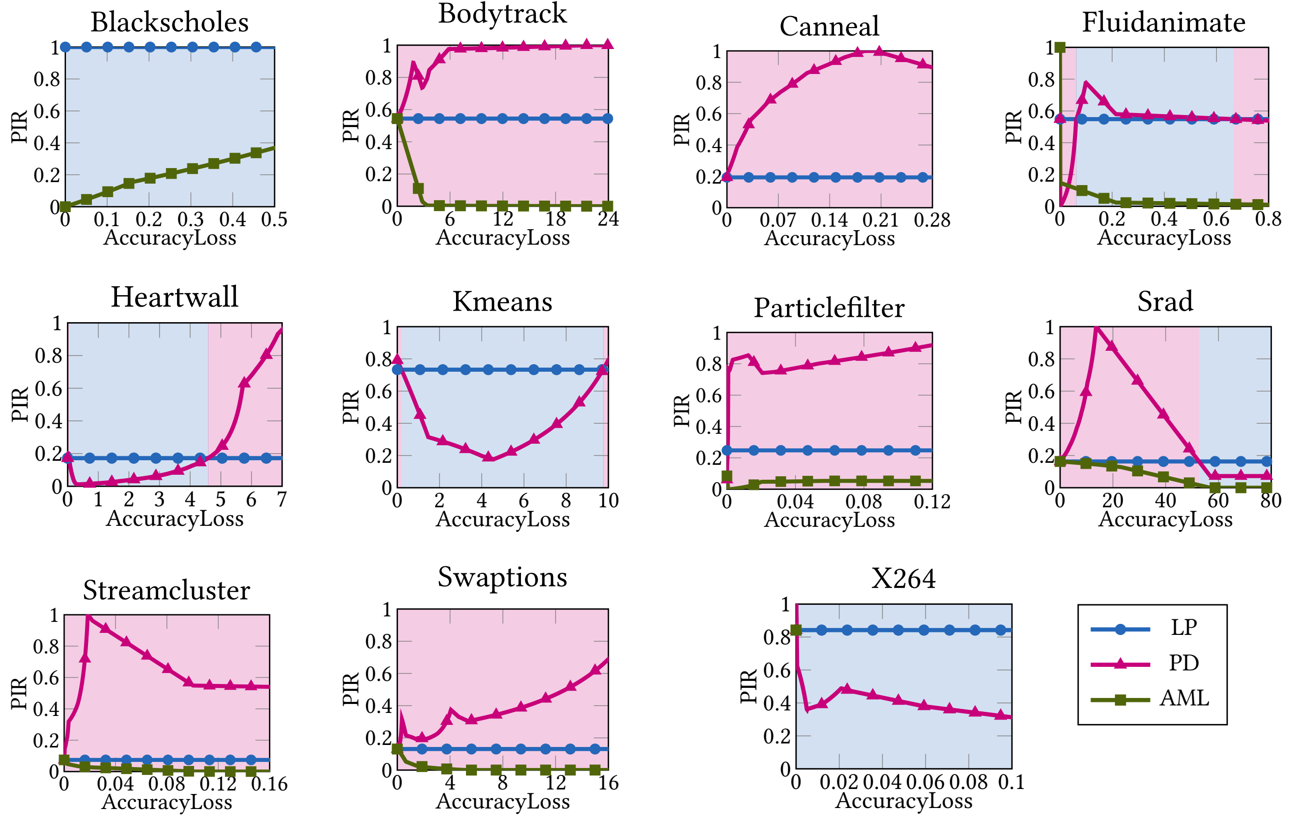


Figure 2.3: VIPER comparison of PowerDial, Loop Perforation, and Approximate Math Library.

`x264` for most accuracy losses.

- While the distinction between approximation frameworks is not possible in Figure 2.2 for `streamcluster` and `swaptions`, VIPER allows quick comparison between those frameworks according to the line positions and the background shade.
- In benchmarks like `fluidanimate` and `srad` where Pareto-optimal curves intersect multiple times, VIPER illustrates intersection by showing fluctuation around the baseline.

CHAPTER 3

BOA AND SOFTWARE APPROXIMATION

This chapter explains software level approximation and BOA (Blending Optimal Approximations) in detail. First, an overview of software approximation techniques is presented. Next, it is followed by the introduction of BOA family and presenting the design structure of various versions of BOA. Then, two techniques related to immense tradeoff exploration is discussed and finally, a case study to illustrate the superiority of BOA in combination of approximate computing techniques is included.

3.1 Background and Motivation

Software approximations target performance improvements by reducing computation operations. While these techniques are easy to combine as they do not require modifications to underlying stack, they have a unique advantage of having a global view of the stack. Software and runtime AC frameworks have access to the whole execution environment, often allows them to make decisions more intuitively.

In the prior chapter, we showed that none of Loop Perforation, PowerDial, or the Approximate Math Library is uniformly best. This motivates us to combine frameworks to take advantages of multiple approximations simultaneously. Although, combining approximation frameworks is actually quite easy—just create a new tradeoff space that is the cross product of all configurations in the original frameworks. The challenge, of course, is quickly locating the Pareto-efficient points in the resulting massive combined tradeoff space.

Table 3.1 lists the number of points in the tradeoff spaces of Loop Perforation, PowerDial, and Approximate Math Library for sample benchmarks. The cross product of all configurations is an intractable tradeoff space. For example, the `x264` benchmark takes up to 4 weeks to test all combined configurations with a single input. Considering multiple inputs

Table 3.1: Number of explored configurations.

Benchmarks	LP	PD	AML	MCKP	NSGA-II	BOA-simple	BOA-flex Th=0.05	BOA-flex Th=0.1	BOA-prob	Combined Configs
Blackscholes	20	-	216	5	240	3	12	18	27	4,320
Bodytrack	768	200	36	15	800	30	224	256	144	5,529,600
Canneal	7	525	-	23	480	33	102	165	110	3,675
Fluidanimate	144	20	6	11	180	24	216	672	168	17,280
Heartwall	256	320	-	19	400	98	665	722	777	81,920
Kmeans	120	100	-	11	200	32	216	216	192	12,000
Particlefilter	200	380	216	12	1000	240	1760	7200	1728	16,416,000
Srad	256	10	36	11	320	48	288	396	160	92,160
Streamcluster	384	256	6	9	480	80	392	1380	640	589,824
Swaptions	768	100	36	15	1000	330	2310	11025	1584	2,764,800
X264	768	400	-	17	1000	45	345	400	405	307,200

should be tested for statistically sound results, the unfeasibility of exhaustive exploration is obvious.

Exploring large tradeoff spaces is well-studied in the context of application specific processor design, which has produced two classes of approaches. The first is carefully selecting a subset of the combined tradeoff space and exhaustively searching that subset [35, 100]. The second class intelligently traverses the entire combined tradeoff space to find more efficient configurations on each iteration. This class does not limit the initial set of configurations but explores a small number of the total configurations [7, 71, 109]. Among these intelligent search techniques, NSGA-II (a genetic algorithm-based approach) has repeatedly outperformed the other exploration techniques [23, 109].

While prior work has proven effective for application-specific processor design, we find that it is not the best match for combining approximation frameworks. Specifically, heuristic exploration of genetic algorithms appears to cause two issues: (1) in an effort to avoid local minima, they produce less efficient combinations and (2) they add too much randomization that leads to a lower correlation between training and test inputs.

3.2 Design

To address the exploration issue of intractable tradeoff spaces, we propose BOA which is a family of exploration techniques that quickly locate Pareto-efficient points in the immense tradeoff space produced by the combination of two or more AC frameworks. All BOAs select a subset of the combined tradeoff space and exhaustively search that space for Pareto-efficient configurations. The first algorithm, BOA-simple, only considers configurations in the cross-product of individual frameworks Pareto-optimal configurations.

Unfortunately, most approximation frameworks are not independent; i.e., their configurations combine in non-linear and unpredictable ways. For example, Loop Perforation changes the number of loop iterations within an application; PowerDial may change convergence criteria. When we combine configurations from these frameworks, we find that some configurations that were Pareto- optimal when considering only the original frameworks are now far from optimal. Conversely, we empirically find that some configurations that were not Pareto-optimal in the original frameworks combine to be Pareto-optimal when we consider multiple frameworks together. This motivates us to expand the BOA-simple to include more non Pareto-optimal configurations in combination.

Thus, we propose BOA-flex that expands the combined search space to consider the configurations that produce a tradeoff within a user-defined threshold of Pareto-optimal. This technique searches more points and tends to find more efficient combinations, but it is deterministic. To avoid local minima in large search spaces, add randomization to build the last member of BOA family. BOA-prob, which uses a probabilistic algorithm to select configurations from each individual framework to combine. Specifically, it uses a sigmoid probability function, so that the closer points are to Pareto-optimal, the more likely they are to be included in the combined tradeoff space. BOA-prob includes most of the same points as the other frameworks, but includes some outliers with small probability, making it more robust in the presence of local optima.

3.2.1 BOA-Simple

The simplest version of BOA forms the cross-product of all Pareto-optimal configurations from the individual frameworks. After executing on evaluation platform, BOA-simple returns the Pareto-efficient configurations found in this combined tradeoff space.

The worst case complexity of BOA-simple is bounded by $O(m + log(m)) * 2^{\frac{n}{m}}$ where m is the number of frameworks to be merged and n contains the total number of parameters of all approximation frameworks[35].[1] In our experiments, input parameters are the sum of the number of loop rates, software knobs, and Taylor series bounds that represent Loop Perforation, PowerDial, and the Approximate Math Library respectively. While the algorithm has exponential complexity, it is practical because so few configurations lie on the Pareto-optimal frontiers produced by individual frameworks (see Table 3.4 for more examples).

3.2.2 BOA-Flex

BOA-flex augments BOA-simple with a user-specified selection threshold, as shown in Algorithm 2. This threshold also removes some inconsistency that may arise due to experimental noise; *i.e.,* it is possible that for high variance applications, the true Pareto-optimal configurations cannot be found with confidence, so adding the threshold makes the search more robust. Specifically, BOA-flex considers all configurations whose tradeoff is within the user-specified threshold of a Pareto-optimal tradeoff.

This threshold is specified in terms of *normalized Euclidean distance.* This simply means that all tradeoffs are normalized so that accuracy loss and runtime range from 0 to 1. Accuracy loss of 1 means the lowest quality. A normalized runtime of 1 is the slowest execution time. A tradeoff point is the output of executing a configuration and, is thus, a pair of accuracy loss and runtime. Having normalized all configurations accuracy loss and runtime,

1. For the purpose of time complexity analysis, we assume each approximation knob can take on two values only, however, in reality, parameters may be assigned a larger number of values.

Algorithm 2 BOA-flex: expands search space by *threshold*.

```
Require: frameworks                                      ▷ tradeoff spaces of frameworks
Require: threshold                                       ▷ User-defined threshold
 1: Combination = []                                     ▷ configurations to explore
 2: for f in frameworks do
 3:     Pareto − opt_f ← Get-Pareto-Opt(f)
 4:     for Config C_i in Pareto − opt do
 5:         for Config C_j in f do
 6:             if NormalizedEuclideanDistance(C_i, C_j) ⩽ threshold then
 7:                 Combination.append(C_j)
 8:             end if
 9:         end for
10:     end for
11: end for
    return Combination                                   ▷ Set of points to explore
```

we can then compute the Euclidean distance between the tradeoffs of two separate configurations. Given this definition, the threshold specifies how close to Pareto-optimal a tradeoff must be for it to be included in the search. For example, the threshold is 0.05, and then the algorithm will include any configuration whose accuracy loss/runtime tradeoff is within 5% of a Pareto-optimal point. If the threshold is zero, this algorithm is equivalent to BOA-simple.

3.2.3 BOA-Prob

While BOA-flex expands the combined search space, it only considers additional configurations that are close to an individual framework's Pareto-optimal curve. To make BOA even more robust to local minima, BOA-prob employs a sigmoid probability function that allows a few points that are further from the individual frameworks' Pareto-optimal curves to be considered in the combined search space. Equation 3.1 presents the sigmoid function:

$$S(C_j) = \frac{1}{1 + exp(\frac{-\Delta+\beta}{\gamma})} \tag{3.1}$$

where Δ is the normalized Euclidean distance between configuration C_j and the nearest Pareto-optimal configuration. β is the horizontal shift and γ decides the smoothness of the

Algorithm 3 BOA-prob: probabilistic search space expansion.

Require: $frameworks$ ▷ tradeoff spaces of frameworks
1: $Combination = []$ ▷ Pareto-efficient configurations
2: **for** f in $frameworks$ **do**
3: $Pareto - opt_f \leftarrow$ Get-Pareto-Opt(f)
4: **for** Config C_i in $Pareto - opt$ **do**
5: **for** Config C_j in f **do**
6: $\Delta = NormalizedEuclideanDistance(C_i,C_j)$
7: $S(C_j) = \frac{1}{1+exp(\frac{-\Delta+\beta}{\gamma})}$
8: r=rand() ▷ Random number between 0 and 1
9: **if** $(r \leqslant S(C_j))$ or $(\Delta d = 0)$ **then**
10: $Combination$.append(C_j)
11: **end if**
12: **end for**
13: **end for**
14: **end for**
return $Combination$ ▷ Set of points to explore

sigmoid curve. Algorithm 3 shows how BOA-prob uses Equation 3.1 to include more points in the combined search space.

We use constants $\beta = 0.2$ and $\gamma = 0.01$ so there is a 92% chance of including the point that has $\Delta < 0.05$, and 50% chance of selecting the point with $\Delta < 0.1$. At $\Delta = 0.2$, there is less than 1% chance of including the point in the combination. If $\Delta = 0$, it means that C_j is actually Pareto-optimal and BOA-prob includes it. The interdependent parameters β and γ control the size of combined tradeoff space and the exploration time should be considered accordingly.

3.3 Evaluation

This section evaluates BOA versus state-of-the-art exploration techniques. First, we examine BOA, MCKP, and NSGA-II using numerical methods. Then, VIPER allows us to compare Pareto-efficient configurations found by each method. We then clarify how the BOA brings more optimality and stability through unseen datasets. Next, we illustrate how optimal are the configurations that BOA locates according to the Pareto-optimal points. Finally, we wrap up the evaluation by illustrating how BOA involves all of the approximation frameworks

to locate more Pareto-efficient configurations.

3.3.1 Experimental Setup

We compare variations of BOA to prior exploration techniques using the same experimental setup from the prior case study. We now split our inputs into training and test data sets. Table 2.1 summarizes the training and test inputs for each benchmark. For each exploration technique, we first use the training inputs to find Pareto-efficient configurations, then we evaluate those points using the test data.

3.3.2 Points of Comparison

We compare BOA to state-of-the-art approaches for locating Pareto-efficient points in large tradeoff spaces:

- **MCKP**: The *multiple choice knapsack problem* variant of the classic knapsack problem has classes of items and must choose one item from each class. MCKP has been used to find Pareto-efficient processor designs in the performance-power space for application specific embedded processors [100]. We declare each framework to be a class. MCKP then selects the Pareto-optimal configurations of each class while keeping the default values for other classes. This creates a new, small tradeoff space which can be searched with brute force to find new Pareto-efficient configurations.

- **NSGA-II**: The non-dominated sorting-based multi-objective evolutionary algorithm (NSGA-II) explores large tradeoff spaces to find non-dominated configurations using an evolutionary genetic algorithm [23]. NSGA-II is the state-of-the-art for multiobjective optimization of embedded processors that navigate performance-power tradeoffs, having been cited over 20,000 times. It finds more efficient configurations in less time compared to other evolutionary algorithms such as the Strength Pareto Evolutionary

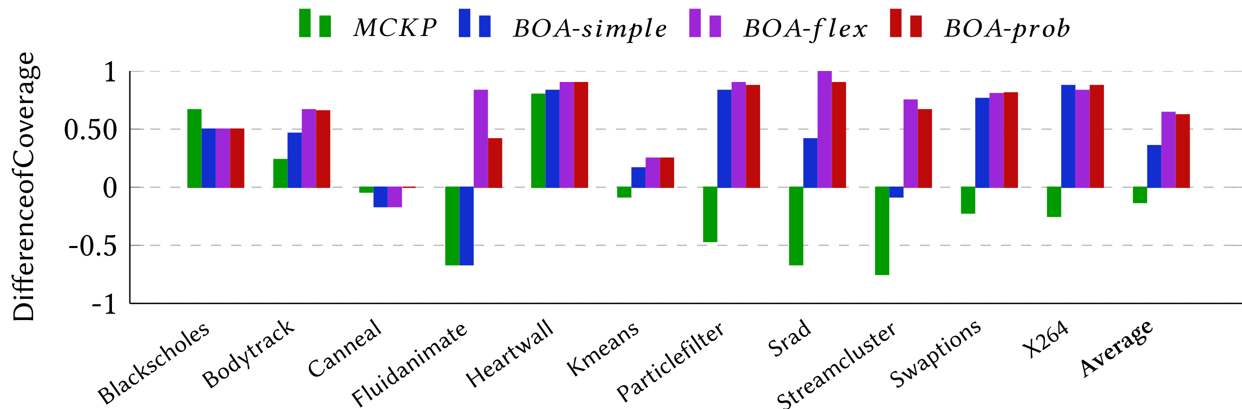


Figure 3.1: Difference of coverage over NSGA-II. Higher bars are better.

Algorithm (SPEA-II) [109] and the Multiobjective Evolutionary Algorithm (MOEA) [51].

3.3.3 Comparison by Difference of Coverage

Recall from Section 2.3 that difference of coverage (DOC) implies the efficiency of one curve over another. Figure 3.1 displays the difference of coverage for various techniques over NSGA-II per benchmark and on average. The y-axis shows $DOC(X, NSGA)$ which is DOC of exploration technique X over NSGA-II. Negative values of $DOC(X, NSGA)$ indicate that X does not find as many Pareto-efficient points as NSGA-II. Conversely, positive values imply that technique X provides that many more values that dominate NSGA-II.

BOA-flex and BOA-prob on average locate 52.8-65.6% more Pareto-efficient configurations than NSGA-II. BOA's superiority is due to its focus on the configurations that have been shown to be Pareto-optimal on individual frameworks. NSGA-II starts the exploration from a random set of points in the combined tradeoff space and iteratively looks for more efficient points. MCKP uses the individual Pareto-optimal curves but keeps the rest of frameworks at default configurations. Since the frameworks are not fully independent, we empirically find some configurations that were not Pareto-optimal in the original frameworks become part of a Pareto-efficient curve of combined tradeoff space when we consider multiple

frameworks together.

Moreover, by expanding BOA with threshold and probabilistic exploration, we search more points which results in finding more efficient configurations. In short, MCKP does not search enough combinations, while NSGA-II searches too many. By restricting the search to points likely to be near the Pareto-optimal frontier for individual frameworks, BOA achieves the right balance and the best empirical results. *This data shows that, for approximate computations, BOA produces many more efficient configurations than prior state-of-the-art search techniques.*

3.3.4 Comparison by VIPER

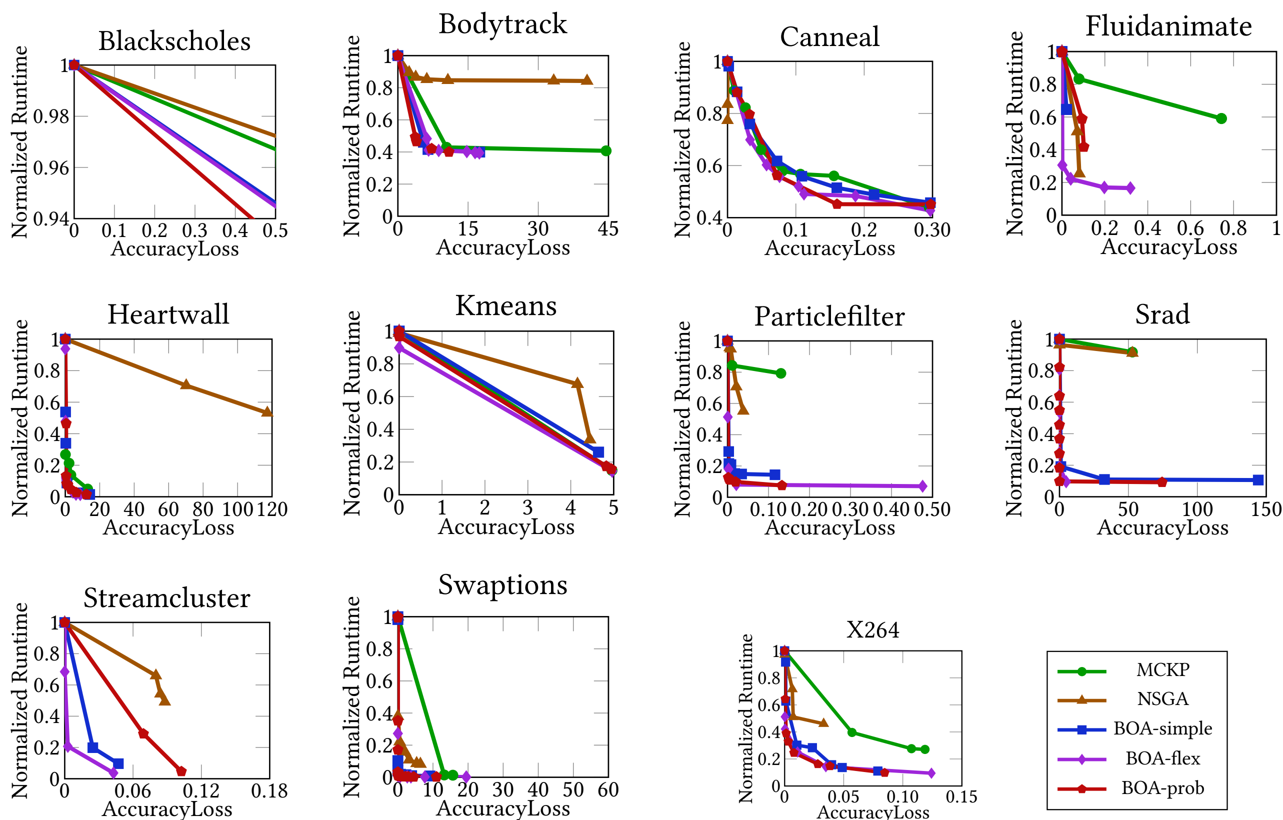


Figure 3.2: Pareto-efficient curves found by each exploration technique.

Figure 3.2 shows the Pareto-efficient points for each benchmark and search method. The

y-axis shows runtime (normalized to the default configuration) and the x-axis represents accuracy loss. We use median runtime across numerous inputs to produce the runtime/accuracy loss tradeoff points. These figures display the range of runtime and accuracy loss that a method can achieve. For instance, NSGA-II and MCKP cannot provide normalized runtime less than 78.1% and 55.5% of the default configuration respectively for the `particlefilter` benchmark. If the Pareto-efficient curves do not intersect, the best approach for a specific benchmark can be easily chosen. When the curves intersect or have steep slopes, however, it can be difficult to differentiate them.

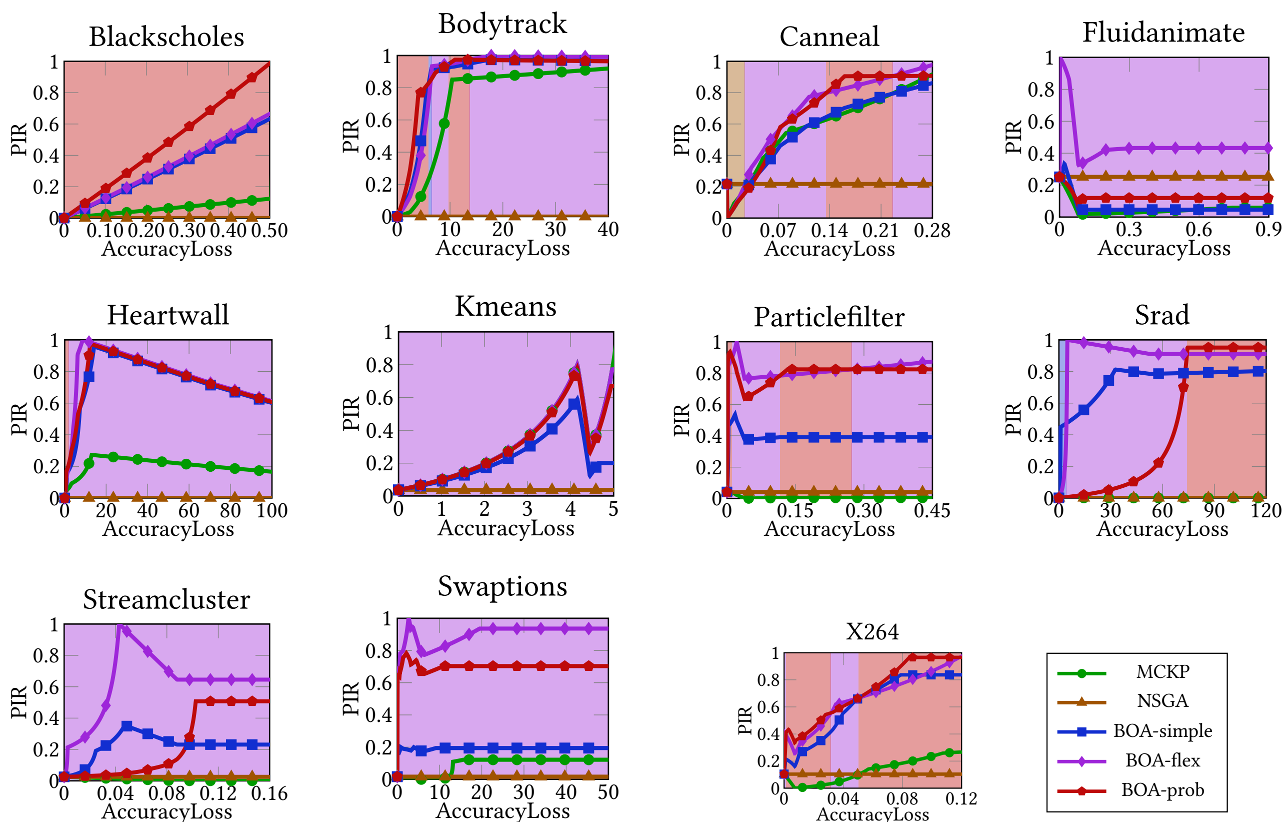


Figure 3.3: VIPER comparison of MCKP, NSGA-II, and different versions of BOA.

We use VIPER to identify the best exploration technique; *i.e.* the one that provides more efficient solutions across a range of accuracy loss. Figure 3.3 illustrates the performance improvement ratio over NSGA-II for MCKP and different variations of BOA. The y-axis

shows the performance improvement ratio while the x-axis shows accuracy loss. We use NSGA-II as the baseline, so it is represented by a horizontal line. For the same accuracy loss, points above that horizontal represent better (more efficient) configurations, and points below demonstrate configurations worse than those found by NSGA-II. For most applications, MCKP stays below the horizontal, meaning it achieves lower Pareto-efficiency than NSGA-II. Moreover, by comparing BOA-simple and MCKP performance improvement ratio lines, we see that BOA-simple outperforms MCKP.

From the VIPER plots, we also find the maximum and minimum performance improvement over NSGA-II. Looking at `fluidanimate`, the NSGA-II line is at 0.25 indicating that the maximum performance is $4\times$ better than NSGA-II, and minimum performance is 25% worse. In fact, for every benchmark BOA-flex finds at least one configuration with higher accuracy for the same performance.

The backdrop color allows quick identification of the most efficient technique at a glance. While for benchmarks like `heartwall`, `kmeans`, and `fluidanimate` BOA-flex is always the best, for others such as `bodytrack`, `canneal`, and `x264` the best technique differs depending on the desired accuracy loss. For benchmarks such as `blackscholes` and `srad`, BOA-prob is more efficient than BOA-flex because it includes some outlier points. Additionally, looking across the x-axis, we see that the BOA-simple lines have more points above the NSGA-II line than below, meaning that on average BOA-simple locates a better set of points. In fact, we find that for 10 of 11 benchmarks, BOA-simple's lines are above the NSGA-II line more than 80% of the time, meaning that BOA-simple's generally finds points that are better than NSGA-II even when selection is not expanded.

Whenever NSGA-II locates more Pareto-efficient points than BOA-simple, by expanding the Pareto-efficient configurations we reduce the performance improvement ratio gap. Benchmarks such as `heartwall`, `kmeans`, and `particlefilter` demonstrate how expanding the combined configurations provides higher Pareto-efficiency. In total, we find by increasing

the threshold, lines of BOA-flex are above the NSGA-II line more than 95% of the time for all of the benchmarks.

Essentially, multiple-choice knapsack problem (MCKP) does not consider points that are not optimal in individual frameworks, while the NSGA-II considers too many points that are not a part of Pareto-optimal curves in individual frameworks. Moreover, MCKP gets stuck in local minima almost immediately. On the other hand, NSGA-II, in its attempt to avoid local minima, wastes too much of its search time exploring random configurations that end up being useless. BOA's method for expanding the search space is simple yet efficient and keeps the exploration close to points that are optimal for individual frameworks. BOA's method of expanding the search space—with a threshold or probabilistically—carefully controls the search space to provide higher efficiency from more intelligent searching rather than exploring more points in the combined tradeoff space. *These results provide visual confirmation that BOA not only finds a greater number of efficient points than prior techniques, but BOA's points are also significantly better, representing much more efficient tradeoffs.*

3.3.5 Exploration Time

The Pareto-efficiency of the located points depends on exploration time. While Figure 3.1 show that BOA produces more and better configurations than other exploration techniques, it is important to know if that gain comes from exploring more points or from a better exploration strategy. Table 3.1 presents the number of configurations explored for each benchmark for different methods, including different thresholds for BOA-flex. To get an estimate of the time spent exploring the combined tradeoff space for a specific benchmark, we can multiply the number of explored configurations by the average runtime (from the last row in Table 1). Comparing NSGA-II and BOA-simple across all benchmarks, NSGA-II explores 2.05% of all possible configurations, while BOA-simple explores about 14× less. BOA-flex and BOA-prob only searches the 0.682% and 0.692% of all possible configurations, respectively.

Thus, even with expansion, BOA explores fewer configurations than NSGA-II. BOA already considers all points on the Pareto-optimal frontier for individual approximation frameworks. The threshold in BOA-flex adds points that are not optimal for an individual framework, but may be combined with other techniques to produce a better configuration. The probabilistic expansion of BOA-prob admits (with low probability) some outlier points that can avoid local minima. *These results indicate that BOA not only finds better combinations of approximate frameworks, it does so with less searching.*

Since MCKP only chooses configurations from individual Pareto-optimal curves rather than merging the configurations, the number of explored configurations stays very low. In the worst case, MCKP explores up to the sum of the Pareto-optimal points of PowerDial, Loop Perforation, and the Approximate Math Library. Unfortunately, while MCKP searches a small space, it is too small to find many useful points.

3.3.6 Input Sensitivity

Since exhaustive exploration is not feasible, we use training and test data to ensure the robustness of BOA on unseen data. We show how well the behavior on training inputs predicts that on test inputs. For each search method, we take the normalized runtime and accuracy loss, compute a linear least squares fit of training data to test data, and compute the correlation coefficient of each fit. Higher correlation coefficients imply less input sensitivity; *i.e.* the behavior of configurations found during training data is a good predictor of test behavior.

Table 3.2 shows the correlation coefficient (R-values) for accuracy loss for each exploration method per benchmark. Table 3.3 shows the R-values for runtime. By harmonic mean, BOA has higher consistency of accuracy loss and normalized runtime comparing to NSGA-II by 17% and 64% respectively. Since MCKP evaluates a few configurations, its correlation is calculated based on just a few points and stays at a high value. More configurations escalate

Table 3.2: Correlation coefficients for accuracy loss.

Benchmark	MCKP	NSGA-II	BOA-simple	BOA-flex	BOA-prob
Blackscholes	1	1	1	1	1
Bodytrack	0.992	0.972	0.990	0.989	0.949
Canneal	0.999	0.999	1	1	1
Fluidanimate	0.539	0.463	0.594	0.943	0.772
Heartwall	0.951	0.341	0.964	0.999	0.959
Kmeans	0.999	0.996	0.999	1	0.999
Particlefilter	0.933	0.999	0.822	0.997	0.997
Srad	1	1	0.958	0.927	0.999
Streamcluster	1	0.403	0.573	0.873	0.562
Swaptions	1	0.999	0.999	0.999	0.999
X264	0.888	0.841	0.992	0.938	0.983
Average	0.908	0.696	0.863	0.968	0.928

the probability of a sparse tradeoff space with test inputs. Thus, MCKP presents higher consistency with R-values close to 1.

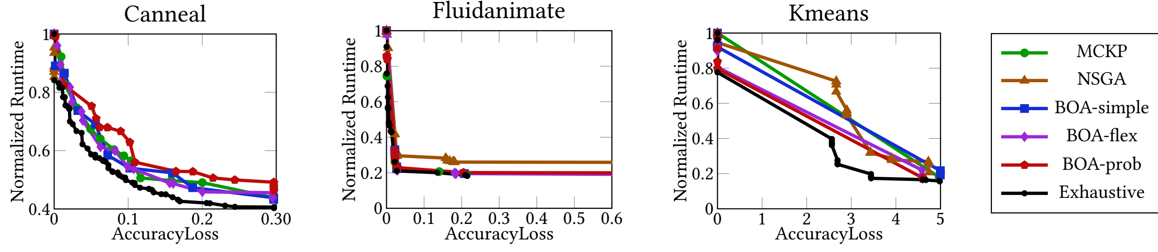
Benchmarks demonstrate different behaviors on test data. While some benchmarks such as `fluidanimate` and `streamcluster` clearly stress the difference between training and test inputs, BOA-flex has uniformly high R-values—all are at least 0.87. Due to the heuristic behavior of NSGA-II, it can select configurations on the training data that often produce bad results on the test data. In contrast, BOA not only finds more efficient configurations, its results are also much more robust when applied to new inputs. *These results indicate that BOA is a statistically sound method for combining approximation frameworks.*

3.3.7 Comparison to Exhaustive Exploration

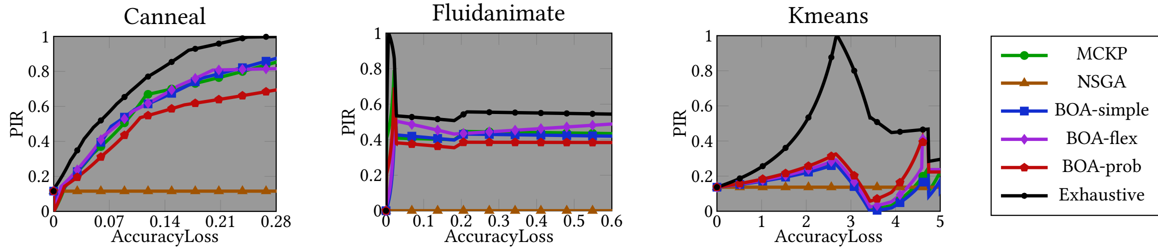
While exhaustive exploration is infeasible for all of the benchmarks using the complete set of inputs, we selected those benchmarks that have smaller tradeoff spaces and evaluate the whole combined tradeoff space for a single input. Figure 3.4 displays the Pareto-efficient curves located by different explorations techniques compared to exhaustive exploration. BOA-simple is closer to exhaustive search than MCKP and NSGA-II. Expanding

Table 3.3: Correlation coefficients for normalized runtime.

Benchmark	MCKP	NSGA-II	BOA-simple	BOA-flex	BOA-prob
Blackscholes	0.993	0.725	1	0.988	0.982
Bodytrack	0.999	0.999	0.999	0.999	0.999
Canneal	0.985	0.319	0.988	0.989	0.998
Fluidanimate	0.999	0.999	0.999	0.953	0.985
Heartwall	0.999	0.999	0.999	0.999	0.999
Kmeans	0.981	0.935	0.992	1	0.997
Particlefilter	0.992	0.987	0.998	0.999	0.998
Srad	0.998	0.746	0.999	0.999	0.999
Streamcluster	1	0.056	0.998	0.999	0.999
Swaptions	1	0.991	0.999	0.973	0.971
X264	0.998	0.995	0.983	0.980	0.998
Average	0.995	0.358	0.996	0.989	0.993



(a) Pareto-optimal curves of located configurations by each exploration technique.



(b) VIPER comparison of MCKP, NSGA-II, and different versions of BOA

Figure 3.4: Comparison of MCKP, NSGA-II, and variations of BOA with exhaustive exploration. (a) shows comparison by Pareto-efficient frontiers and (b) shows comparison by VIPER.

the combined tradeoff space with either BOA-flex or BOA-prob reduces the gap between Pareto-efficient and Pareto-optimal further.

Looking at subplots of Figure 3.4, we note that VIPER is designed to compare approxi-

Table 3.4: Number of used configurations from each approximation framework.

Benchmark	LP	PD	AML	BOA-simple
Blackscholes	1	-	3	3
Bodytrack	6	5	1	30
Canneal	3	11	-	33
Fluidanimate	4	3	2	24
Heartwall	7	14	-	98
Kmeans	4	8	-	32
Particlefilter	6	8	5	240
Srad	2	8	3	48
Streamcluster	8	5	2	80
Swaptions	11	5	6	330
X264	9	5	-	45

mation techniques to each other, and not to certain fixed standard. Therefore, the shapes of the curves and the ranges of axes can change significantly as different techniques are added or removed from the comparison. In addition, the exhaustive search for `kmeans` finds configurations whose individual components are extremely far from optimal. Neither BOA, nor the existing search techniques can pick up on these points, but that is just the reality of dealing with an exponentially large search space with many local optima. *These results demonstrate that BOA achieves near-optimal performance for those applications where exhaustive search is feasible.*

3.3.8 Combination Distribution

When BOA combines approximation frameworks, it considers multiple configurations from each framework rather than choosing from one or two of frameworks only. Table 3.4 includes the number of configurations BOA-simple selects from each approximation framework to generate the new, combined tradeoff space. Expanding the configurations in combination may change the distribution. As we mentioned in the Section 3.3.4, the Approximate Math Library is never better than neither Loop Perforation nor PowerDial in any range of accuracy loss. However, BOA uses the Approximate Math Library in combination with Loop

Perforation and PowerDial for 7 out of the 11 applications studied. The only reason the Approximate Math Library is not used for those four applications is that they do not make any calls to the math library. *These results, however, show that there is real benefit to combining frameworks, as even the Approximate Math Library—which is uniformly the worst of the three techniques by themselves—contributes to Pareto-efficient points in the combined space found by BOA-simple.*

CHAPTER 4

MAMBA AND PROGRAMMING LANGUAGE APPROXIMATION

This chapter describes approximation and programming language level, reviews current state-of-the-art techniques, introduces the MAMBA (Mechanisms for Applying Multi Bit Approximations), and evaluates MAMBA with a case study on convolutional neural networks.

4.1 Background and Motivation

Languages support approximation allowing specification of variants for key functionality and formal analysis of their effects. They deal with the errors induced at lower layers or generate errors to higher level approximations directly. These set of techniques include precision tuning at fine grain [19, 78, 80], programming language supports [4, 9, 17, 95, 101, 102], or introducing compiler pragmas for indicating the approximable sections [73, 92].

Approximation Knobs provide a way to lend performance and energy gains to existing power knobs [49]. Quora is a quality programmable processor where the notion of quality is codified in the instruction set of the processor [96]. Another example of user-defined approximation is Green, which is a system that allows programmers to supply approximate versions of loops and while-blocks that terminate early [9].

Performing precision tuning at fine grain is available through software libraries. EnerJ proposes to declare approximate data via type qualifiers [80]. MPFR adds to its arbitrary-precision representation the support for rounding modes, exceptions and special values as defined in the IEEE 754 standard [31]. FlexFloat reduces floating point emulation time by providing a C/C++ interface for supporting multiple FP formats. These techniques require source code instrumentation (changing $float$ and $double$ variables definition to custom

parameters) or intending to yield more precise computation (for instance floating points numbers with more than 128 bits). MAMBA focuses on energy efficiency by reducing precision while only requiring the program binary.

A body of literature has focused on providing tool supports that allow users to define several approximations for different components of the application [29, 52, 68, 81]. Petabricks provides language extensions that expose tradeoffs between time and accuracy to the compiler [4]. The compiler then runs dynamic autotuning to generate optimized elements to achieve the target accuracy. However, autotuners need to be determined on a per-application basis by the user. OpenTuner provides fully-customizable configuration representation and ensembles of search techniques to find an optimal solution [5]. Both autotuning techniques are supposed to help programmers but Petabricks requires a separate language and both require users to implement all alternatives before the search can be conducted. MAMBA also helps users deal with approximation, but instead of requiring users to implement all possible alternatives, users simply describe programmable rules that are then used to automatically generate the alternatives.

While prior work mainly develops *mechanisms* that enable approximation to provide energy and runtime savings at different domains, they do not help users make more informed decisions about approximation. These techniques mostly are not flexible about how much, where, and when to approximate, and only provide discrete approximation knobs which leads to more conservative design choices. MAMBA does not propose new mechanisms but helps users answer the questions above.

Current inexact functional units in addition to approximate software libraries create an opportunity to exploit quality-energy tradeoffs. While an FPU accounts for 2-5% area on the chip, the floating point instructions consume significantly more energy compared to other classes of instructions such as integer, memory, and control [10, 64]. Figure 4.1 illustrates the energy per instruction (EPI) results for different classes of instructions of 64-bit 32nm

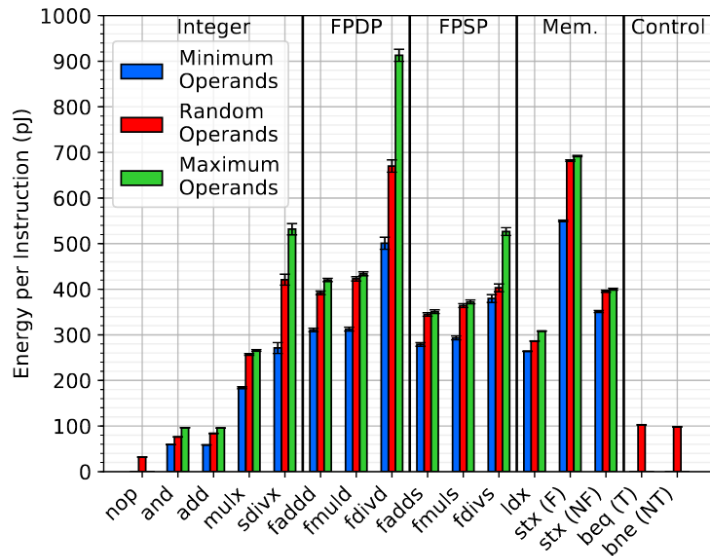


Figure 4.1: Energy Per Instruction for different classes of instructions. (From [64])

processor. With random operands, a 64-bit floating point add consumes 400 pJ, and a division operation could go as high as 680 pJ. For a 32-bit version, the energy consumption is 350 and 420 pJ respectively.

As expected with regards to the type of operations, executing the floating point instructions emerges as a major contributor to the total energy consumption. Recent empirical studies have shown up to 50% of the energy consumed in a core and memory is related to floating point instructions [64]. Thus, exploiting reduced bitwidth at instruction level (bit truncation) to generate Floating Point Implementations (FPI) could facilitate higher energy efficiency. Another useful insight from Figure 4.1 is the relationship between computation and memory accesses. For example, three `add` operations consume the same amount of energy as a `ldx` instruction. Hence, looking from an energy efficiency point of view, reducing the memory traffic could be as efficient as optimizing the floating point arithmetic operations [64].

Hence, there is a need for a generic framework that provides multiple precision levels, accommodates custom user-defined floating point implementation, and does not require code

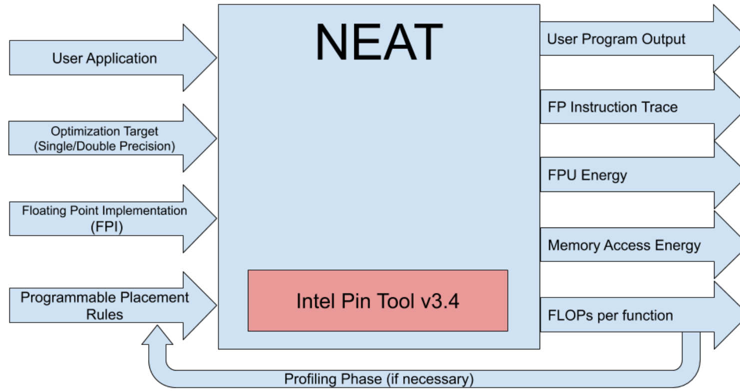


Figure 4.2: MAMBA design.

refactoring. MAMBA provides such a solution. MAMBA generates insightful information for precision tuning at different granularity for floating point intensive programs.

4.2 Design

The main challenge of precision tuning is constructing the right configuration of floating point precisions for the application. This configuration space might be extremely large to fairly small, ranging in complexity from using a different floating point implementation for each dynamic floating point instruction to using a different implementation for different function calls, or just picking a single floating point implementation for the entire application. MAMBA provides such flexibility in the granularity of enforcing floating point approximations by introducing the programmable placement rules and then automatically searching the accuracy and energy tradeoff space to find the optimal frontier.

Figure 4.2 illustrates the MAMBA system from the user perspective. Users specify: (1) the application that they want to understand (this could be just a binary and requires no special changes), (2) whether MAMBA should consider double or single precision (or both), (3) a set of alternative implementations for floating point arithmetic, and (4) the programmable placement rules that describe when, where, and how in the program to replace the standard floating point operations with one of the alternative implementations. MAMBA

then runs the program as a pin tool and intercepts all floating point operations of the specified type and replacing them according to the rules. MAMBA will perform multiple runs of the application, collect statistics on floating point usage, accuracy, and estimated energy. MAMBA offers a profiling mode where the user collects precision analysis such as quantity and frequency of FLOPs for the application before applying any FPIs. Ultimately, MAMBA can repeatedly test different assignments of floating point operations to find the frontier of optimal configurations; i.e., assignments of floating point operations to different regions of the code.

This section describes MAMBA's inputs, internals, and outputs.

4.2.1 MAMBA Inputs

User inputs of MAMBA includes: a user application to instrument, a precision level as the optimization target, the desired FP arithmetic implementations, and a set of FPI to function mappings (programmable placement rules).

MAMBA receives the binary of the program and instruments the floating point instructions. Unlike other precision tuning tools, MAMBA does not require the source code of the program. Then, MAMBA expects the optimization target which can be either single or double precision. There are two reasons behind including optimization objective. First, for most of the programs, the same precision level is held across the code base for the data structures and the functions. Second, if we consider both $float$ and *double* FLOPs to optimize, the configuration space of FPIs combinations would explode excessively.

Next, users specify multiple FPIs for any individual arithmetic instruction such as addition, subtraction, multiplication, and division for each operand. At last, MAMBA expects a mapping between the candidate code sections and the FPIs to calculate each FLOP in a program. By default, MAMBA enforces the FPIs at the function level, meaning all FLOPs executed within a specific function will be using the same customized FPI. Any function

that has at least one FLOP can be considered as a candidate for approximation.

Intel Pin Tool

The Pin instrumentation system was chosen as the backbone for this tool because of its clean API and efficient implementation. The Pin API makes it possible to write instrumentation routines to observe and alter the architectural state of a process. Pin uses a JIT compiler to generate a new instrumented code that can be executed without the extra runtime overhead from instrumentation.

Floating Point Operations

For the purposes of this tool, we identify floating point arithmetic operations as the Streaming SIMD Extensions (SSE) instructions for scalar arithmetic. These instructions are included in a SIMD instruction set extension to the x86 architecture and operate on 32-bit or 64-bit floating point numbers. More specifically, the instructions we use for our definition of floating point operation are ADDSS, SUBSS, MULSS, DIVSS, ADDSD, SUBSD, MULSD, and DIVSD.

Floating Point Arithmetic Implementation

Custom hardware units or accelerators have been considered for enriching the quality versus energy tradeoff spaces. Approximate adders [25, 97, 108] and multipliers [50, 53, 105] have been designed as a solution for lower power consumption and high performance. In the presence of inexact hardware units, MAMBA provides information on how to efficiently redirect the arithmetic instructions to these units.

The floating point formats with a lower number of bits emerge an appealing opportunity to reduce the energy consumption since it allows simplification of both hardware units and reduction of memory bandwidth required to transfer the data between the memory and

Table 4.1: Built-in placement rules in MAMBA.

Placement Rule	Description	tradeoff Space Size
WP	one FPI for the whole program	$24 - 53$
CIP	one FPI for the currently in progress function	$24^{10} - 53^{10}$
FCS	one FPI for the most recent function on the call stack	$24^{10} - 53^{10}$

registers. The FPI can be as simple as bit truncation in the FP format representation, enforcing direct approximation to the operands or result of arithmetic operations, or redirecting instruction to approximate hardware units or software libraries.

Execution of Floating Point Instructions

Defining an FPI is fairly trivial. The main challenge with enforcing FPI dynamically is the way to specify the exact mapping between FPI and the FLOPs. MAMBA allows users to define placement rules that determine which FPI is used to calculate each FLOP in a program. Every time a FLOP is about to be calculated in the user application, MAMBA examines all of the mappings and captures information about the current state of the application, and use them to determine which FPI will be applied to calculate the result of the FLOP.

MAMBA comes packaged with three predefined sets of FPI placements for the applications to cover many use-cases and show off its versatility. Table 4.1 includes the default placement rules and the corresponding tradeoff space size. Sets of rules are specified as C++ routines that accept the program state as input and return a single FPI as output.

The first set applies the same FPI for every FLOP in the whole-program (WP) regardless of the current function and the program state.

For finer granularity, the user can register callbacks through MAMBA that can be executed whenever a function is entered or exited in the instrumented application. This allows more complex information to be collected about the program state, such as the call stack of the application. The second set of placement rules allows the user to specify a map of function names to FPIs and employs each FPI for the FLOPs in the corresponding currently in

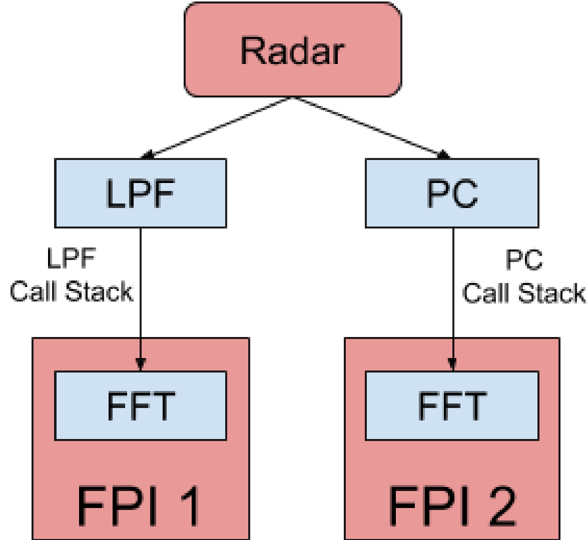


Figure 4.3: FCS placement considers FFT function call stack before selecting the approximate FPI.

progress (CIP) function. Similarly, the third set of placement rules uses callbacks registered with MAMBA to keep track of the function call stack (FCS) of the program. Instead of inspecting the current function, MAMBA first checks the most recent function on the call stack. If no functions in the call stack match the names of those in the user-supplied map, a default implementation is used.

To highlight the difference between CIP and FCS, we analyzed the structure of functions in a benchmark shown in Figure 4.3. The `radar` is an embedded real-time signal processing application that is used to find moving targets on the ground [55]. It includes both a low-pass filter (LPF) and pulse compression (PC). Both of these components use a Fast Fourier Transform (FFT) as a part of their computation.

With the CIP option, MAMBA enforces the same FPI every time the FFT function is called. For the FCS option, MAMBA distinguishes between the two occurrences of FFT based on who has made the function call. Therefore, MAMBA uses one FPI for the FFT in the low-pass filter (LPF) stage and a second FPI for the FFT in the pulse compression (PC) stage. Empirically, we have found the results of FCS and CIP for most of the benchmarks

do not differ as the callers of a FLOP intensive functions are the same. The `radar` is an example where multiple functions make numerous calls to the same FLOP-intensive function that is accuracy sensitive.

4.2.2 MAMBA Outputs

There are five outputs from this tool: the output from the user application, a trace of the operands and result of every FLOP executed by the program, the estimated FPU energy of FLOPs in the execution of the program, the estimated energy of off-chip memory accesses of the program, and the number of FLOPs executed per function in the program.

The trace of the FLOPs executed by the instrumented application is written to a file while the application is running. If FPIs are supplied to MAMBA by the user, the result of each operation will be printed after the operation is calculated with the chosen FPI. The operands and result of each operation are printed as hexadecimal numbers so that there is no confusion in rounding the floating point values.

MAMBA reports total energy consumed in FPU by using energy per instruction (EPI) of different classes floating point operations. We extracted the energy model of $fadd$, $fmul$ and $fdiv$ for single and double precision operations provided in related work [64].

To this end, MAMBA counts the number of bits manipulated in the operands and results of every FLOP in the instrumented program. Modifying the bit width in the exponent and sign of a floating point number changes the accuracy significantly where the quality of output becomes unacceptable. Hence, MAMBA only focuses on mantissa bits. MAMBA counts the number of zeros in the binary representation of the floating point number, starting with the least significant bit, and then subtracts it from available mantissa bits in the floating type (24/53 bits in single/double precision respectively) to calculate the number of manipulated bits. MAMBA uses the EPI models and the number of manipulated bits per FLOP to estimate the total floating point energy consumed in the FPU.

MAMBA also records the total number of bits used in FLOPs in the execution of the program is output to a file after the termination of the application. Unlike the FPU energy estimation, this metric can be used as a platform-independent way to evaluate the approximate amount of power used by FLOPs when instrumenting a program.

Currently, the memory accounts for more than 25% of energy spent in a large scale system. While on average, each single precision FLOP takes 400 pJ to execute, a byte read from memory consumes 1.5 nJ [15]. Accordingly, MAMBA counts the total number of bits transmitted to/from memory and then estimates the total memory access energy of the instrumented program [61]. This allows MAMBA to yield a better energy estimation of the program in a real system.

MAMBA generates in-detail statistics about the floating point instructions in the program. Users might operate MAMBA to profile the application before performing precision tuning to first, decide whether MAMBA is useful to their application and second, what type of FPIs, which functions, and how to map them. In general, MAMBA is a tool used at program design time. MAMBA allows users to evaluate many points on the accuracy/energy tradeoff curve without having to implement all possible alternatives. After profiling with MAMBA, users can then select a point and implement it with confidence that it will provide the desired behavior.

4.3 Interface and Runtime Manual

We explain how the user can manage floating point precision scaling with the MAMBA framework explained in the previous section. We specify the information that MAMBA expects to receive from the users and then, discuss steps to execute the runtime engine of MAMBA.

The MAMBA procedure follows as:

1. **Profile the program**: User runs the application. MAMBA records the single and

double precision instructions and the functions associated with them and generates the detailed report in csv format.

2. **Assign FP Optimization Target**: Since the applications usually use the same precision level across the source code, MAMBA enhances either single or double precision instructions at the same time. At this point, the user defines the directive for MAMBA to target 32 or 64 bit FLOPs.

3. **Develop FPIs**: User might define multiple FPIs to be enforced by the MAMBA. An FPI can be created by truncating mantissa bits of the FLOP representation or injecting direct approximation to the operands or results of floating point arithmetic operations. For example, approximating the inverse function [106] or sin function using a neural network[26] is considered an FPI too. The FPI can be applied to one or more floating point arithmetic instruction. For instance, one benchmark might include numerous accumulations but few divisions. Thus, the user defines an FPI with enforcing 8 precision bits for the add/sub arithmetic instructions and 24 precision bits for the multiply instructions. The user develops an FPI by creating an instance of $FpImplementation$ virtual class. Furthermore, user might customize the subroutine of $PerformOperation$ to modify the operands or results of a floating point instruction directly.

4. **Register FPI Placement Rules and Functions**: MAMBA expects to receive a mapping between FPIs and when to enforce them. For the WP approach, the user only needs to instantiate $Register_FP_selector$ class with the desired FPI as the argument. For the per-function rules, MAMBA by default considers the top 10 FLOP intensive functions. The user might pre-profile the program to detect and select any number of functions. The user then should provide a mapping between functions and FPIs by defining a $pair < functionName, FPI* >$ map data structure. Next, the user should combine the map with one the pre-packaged placement rules (CIP or FCS). This mapping is also referred as a *configuration*. Finally, the user creates an instance of $Register_FP_selector$ class and

passes the map and placement strategy as the input arguments. At the runtime, the user passes the registered instance name via `fp_selector_name` command line flag to MAMBA.

5. **Activate Exploration Scripts**: If CIP or FCS schema is selected, the tradeoff space of FPI to function mappings (configurations) becomes too huge to explore exhaustively. Hence, MAMBA uses the NSGA-II genetic exploration technique to search for energy efficient configurations [24]. If the user desires to enhance the exploration phase of the configuration space further, MAMBA provides an interface through the command line flags to manually modify the tuning parameters of NSGA-II such as population size, number of generations, or convergence threshold.

6. **Analyze the Output**: MAMBA reports detailed energy and performance data per configuration. Moreover, a python script is provided to generate scatter plots of tradeoff space with the lower convex hulls.

At the completion of these steps, the user finds information about the most appropriate precision level for each individual function or the whole program.

4.4 Evaluation

We evaluate the efficacy and flexibility of MAMBA to provide floating point approximation analysis. In general, MAMBA generates useful information on precision tuning of applications which can be used at design stage of a software or conveyed to other layers of system such as compilers or hardware (*e.g.* building a set of reduced-precision FPUs). Section 4.4.2 inspects the floating point profiling of MAMBA for the applications. The primary challenge of automatic precision tuning is creating approximation configurations. We examine the MAMBA's flexibility to produce customized FPI definitions in Sections 4.4.3 and 4.4.4. Moreover, the main mechanism of MAMBA—programmable placement rules—are investigated in Sections 4.4.5 and 4.4.6.

To navigate through the immense configuration space, MAMBA comes with a tunable

genetic exploration algorithm which is used in Sections above (from 4.4.2 through 4.4.3). Although, to ensure the robustness of MAMBA on unseen data, we evaluate the difference between predicted accuracy and energy on training and test data to demonstrate that MAMBA finds configurations that are robust across different test inputs that were not seen in training 4.4.7. Finally in section 4.4.8, we evaluate MAMBA's general applicability to find the appropriate reduced precision floating point configurations by evaluating it on a problem that has seen a tremendous amount of attention from human experts recently: trading accuracy for precision in neural network inference. We find that MAMBA can use the whole-program rule to automatically find a single floating point precision that is similar to those reported by human experts. Further, we find that by using different floating point implementations for different layers, MAMBA produces even greater energy savings for the same accuracy.

4.4.1 Experimental Setup

We evaluate MAMBA by exploring the tradeoff spaces of the placement rules for a variety of benchmarks. Table 4.2 lists the applications from Parsec 3.0 [14] and Rodinia 3.1 [18] suites with the configuration space size (default precision optimization target) and training and test inputs for each benchmark. These benchmarks cover domains from finance to image processing.

To create FPIs, we use bit truncation. For the single precision floating point numbers ($float$ type in C), we have 24 different FPIs corresponding to the mantissa bits. Similarly, we created 53 FPIs for the double precision floating point numbers. For the whole-program approach, the size of the tradeoff space is the total number of possible FPIs which are 24 and 53 points. For the per-function approaches, we consider the top 10 functions with most FLOPs to enforce the FP rules, so each of the top 10 functions may use a different FPI.

In each experiment, at most 400 configurations in the tradeoff space (less than 6^{-12} of

Table 4.2: Benchmarks used for evaluation.

Benchmarks	Training inputs	Test inputs	Possible Configuration Space
Blackscholes	10 lists with 100K initial prices	30 lists with 100K initial prices	24^4
Bodytrack	Sequence of 5 frames	Sequence of 20 frames	24^{24}
Fluidanimate	5 fluids with 15K+ particle	15 fluids with 15K+ particle	24^9
Ferret	5 databases of 16 images	15 databases of 16 images	24^{12}
Heartwall	Sequence of 15 frames	Sequence of 60 frames	24^4
Kmeans	10 vectors with 512 data points	30 vectors with 512 data points	24^9
Particlefilter	Sequence of 32 frames	Sequence of 128 frames	53^{10}
Radar	Sequence of 10 frames	Sequence of 40 frames	24^{13}

all possible configurations) have been evaluated through MAMBA's genetic algorithm.

4.4.2 Floating Point Precision Distribution

MAMBA can be used to analyze the type, distribution, and the intensity of the FLOPs in a program. Figure 4.4 depicts the ratio of single and double precision FLOPs for each benchmark.

Most of the benchmarks hold the same precision level across the source for correctness and portability. For example, `bodytrack`, `heartwall`, and `kmeans` are all implemented with *float* type while `canneal` is mainly using *double*. However, for some benchmarks such as `ferret`, `particlefilter`, and `srad` due to including external libraries, there is a mixture of both precision levels. In this case, users might choose the optimization target to be enforced. Specifying the right target opens up further opportunities for additional energy savings.

4.4.3 FPU Energy Saving

MAMBA provides the FPU energy estimation consumed by the FLOPs. We compare two rules: whole program (WP) and currently-in-progress (CIP). As a reminder, WP uses one floating point implementation through the entirety of the program, while CIP is free to choose

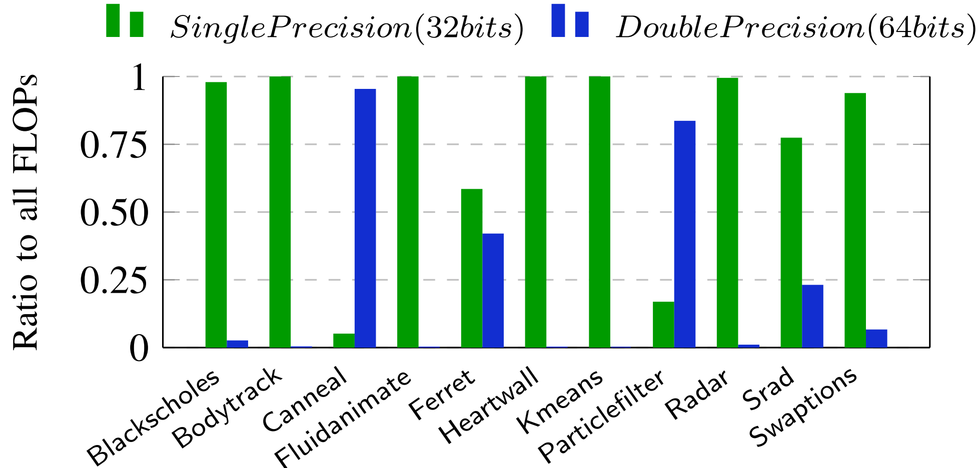


Figure 4.4: Floating-point type breakdown for benchmarks. While most benchmarks have a dominant FP type, some carry both.

a separate implementation for each of the top 10 functions (by FLOP count) in the program. For `particlefilter` and `canneal`, we set the optimization target to double precision as most of the FLOPs are *double*. For the rest of the benchmarks, we apply the single precision optimization.

We consider the top 10 FLOP intensive functions for the CIP placement. Although, one might ask how much of the FLOPs are included in the top 10 functions. For all benchmarks, at least 98% FLOPs were coming from the top 10 functions, thus MAMBA covers almost all of the FLOPs in the program.

Figure 4.5 illustrates the lower convex hull of normalized FPU energy and the error rate (also referred to as accuracy loss). The error rate metric is the relative error of a configuration comparing against the highest quality configuration (baseline) where no approximation happens. The horizontal axis is the error rate while the Normalized Energy Consumption (NEC) to the baseline is shown vertically (on the y-axis). The lower the curve is, the more efficient configuration is found which means higher energy efficiency. Since users generally do not care about extremely inaccurate outputs, only error rates less than 20% is shown in the subfigures. The results show that if we assign multiple FPIs at the function level, MAMBA

will retrieve more energy efficient configurations that are not explorable if we use single FPI for the whole program. This result further demonstrates MAMBA's value in design space exploration.

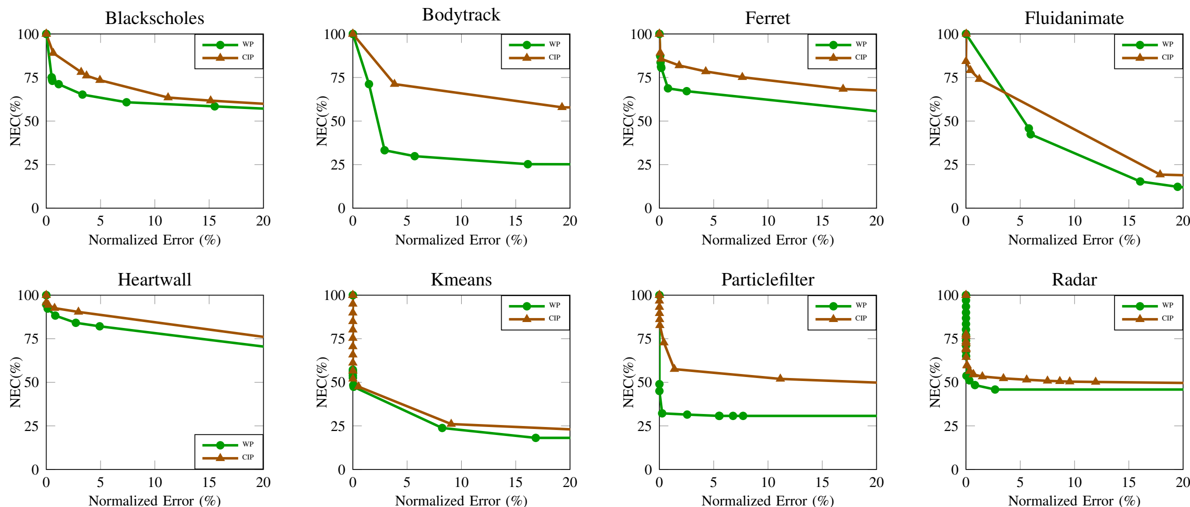


Figure 4.5: Lower convex hulls of FPU energy and error rates for the WP and CIP. Values are normalized to the baseline.

With a minimal error in final output of the benchmark, MAMBA reduces the FPU energy up to 60%. For some applications such as `blackscholes`, `fluidanimate`, and `particlefilter` the FPU energy savings are more considerable. These benchmarks have less than 10 FLOP intensive functions. Therefore first, CIP covers all the FLOPs in the program. Second, since the tradeoff space is relatively smaller, NSGA-II searches a larger portion of the tradeoff space in the same exploration time.

For `fluidanimate` and `ferret` benchmarks, there are only three and two configurations where the WP outperforms the CIP. The reason is that MAMBA's genetic algorithm fails to explore those specific configurations as it is a heuristic algorithm. The same pattern can be seen for the `radar` benchmark as well where the CIP does not dominate the whole-program approach.

The `heartwall` benchmark has only two FLOP functions where they are very sensitive to the bit width adjustment and any modification leads to more than 20% error. Consequently,

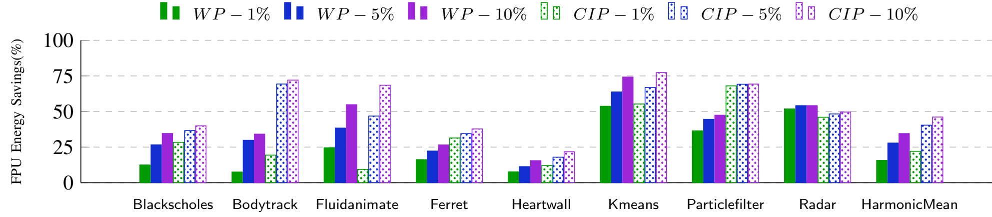


Figure 4.6: FPU energy savings at different error rates, normalized to the baseline. Higher the bars, the more energy efficient is.

MAMBA is not able to decrease FPU energy to less than 71% of the baseline with reasonable error rate. The opposite scenario happens for the `particlefilter` application where the major FLOP functions do not impact the quality of output considerably, hence MAMBA aggressively reduces the FPU energy without causing much error.

For a more detailed comparison, we re-illustrate a quantized representation of the previous plot. Figure 4.6 displays how the FPU energy savings enhance as the tolerated error threshold increases. Higher bars indicate more energy savings. By harmonic mean, applying the CIP versus WP approach results in 7%, 12%, and 13% more energy savings at 1%,5%, and 10% error rate, respectively.

The steeper slope in the lower convex hull curves in subplots of Figure 4.5 translates into higher bars in Figure 4.6 as the error threshold increases. The `blackscholes` and `particlefilter` benchmarks demonstrate such behavior. On the contrary, by increasing the error threshold in `particlefilter` and `radar` applications, the FPU energy savings do not inflate similarly.

From these graphs, we draw two conclusions. First, specifying the FPIs placement at a finer granularity results in more efficient FPI to function mappings. In other words, per-function rules use less energy with the same error comparing to use a single FPI for the whole application. This type of insight is really only achievable with the an automated system like MAMBA. Second, if higher error rates are allowed, MAMBA achieves higher efficiency of FPU energy. Thus, MAMBA can navigate the whole tradeoffs space and give users a range

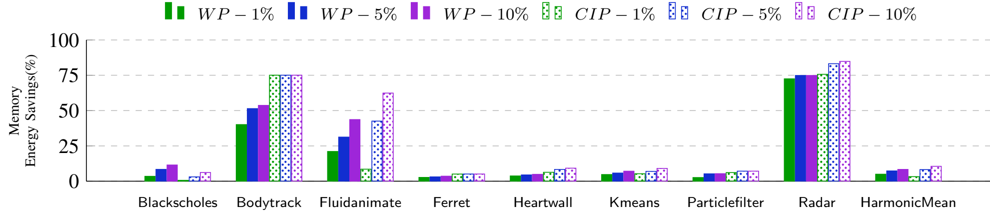


Figure 4.7: Memory transfer energy savings at different error rates, normalized to the baseline.

of options depending on tolerable error rate.

4.4.4 Memory Instructions

Main memory (DRAM) consumes as much as half of the total system power in a computer today, due to the increasing demand for memory capacity and bandwidth [61]. Hence, reducing the memory traffic directly derives into substantial energy savings. MAMBA estimates the memory energy with accounting only accesses to/from an off-chip memory by keeping track of memory operations such as MOVSS and MOVSD. Figure 4.7 depicts memory accesses energy for a range of error rates for both whole-program (WP) and per-function (CIP) approaches respectively across the benchmarks. Same as before, higher bars indicate higher energy efficiency. Values are normalized to non-approximated version of the application, which acts as a baseline. On a harmonic mean, increasing the error rate from 1% to 10% results in 3.2-10.5% less energy consumption.

If the FLOP functions are memory intensive, reducing the precision bits results in lower memory bandwidth, and consequently more energy savings. That is the reason why benchmarks such as `bodytrack`, `fluidanimate`, and `radar` reduces the memory energy by more than 60%. In rest of the benchmarks, the FLOP functions were solely compute intensive.

To put the experiments above to a conclusion, we illustrate the WP rule as a sample for prior work [99] which tries to find a single most optimal approximation for the whole application. The per-function rules of MAMBA show off the ability of the replacement rules

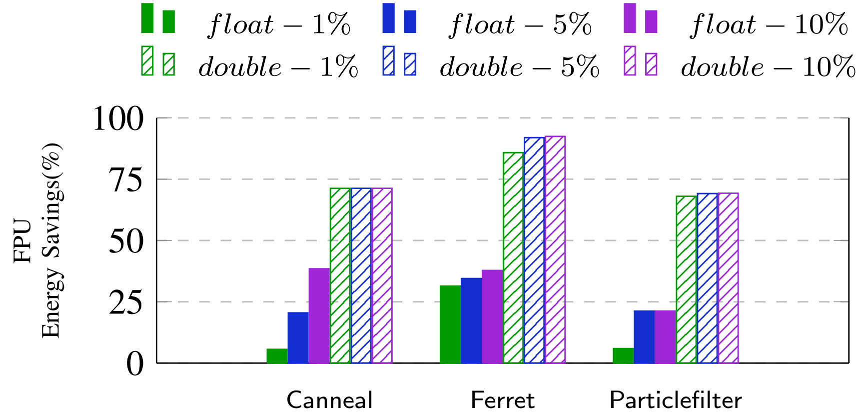


Figure 4.8: FPU energy savings with different optimization targets for MAMBA.

to allow programmers to explore a richer set of tradeoffs without having to come up with whole new implementations of existing program functionality.

4.4.5 Flexible Precision Level

In previous sections, we observed some benchmarks have a mixture of both *float* and *double* FLOPs. To choose the right optimization target, we compare the energy and accuracy of selected benchmarks in both single and double optimization targets. The FPI to function mapping is CIP in this experiment.

Figure 4.8 shows the normalized energy savings for both single and double optimization targets. As expected, if we choose the optimization target to be the same as the FP type which has larger ratio in FLOP distribution, higher energy savings would be achieved. This observation can be easily justified by the looking back at Section 4.4.2. Both `canneal` and `particlefilter` contain more 64-bit than 32-bit FLOPs. Thus, double precision as MAMBA directive is the right choice to achieve substantially higher energy efficiency.

The `ferret` requires special attention as it is not obvious how to choose the optimization target based on FLOP distribution ratio since it has almost equal amount of *float* and *double* FLOPs. At the 10% error rate, MAMBA saves up to 92% of FPU energy corresponding to

double instructions while only 38% savings is available if we consider only *float* instructions. There are two reasons for the discrepancy. One is that generally *double* FLOPs yield more precise output, but they use more precision bits in return. Thus, MAMBA has more freedom to cut down unnecessary floating point bits while not losing much accuracy because the *double* baseline is already more accurate than the *float* one. Second, the *double* functions in `ferret` are not accuracy sensitive, meaning that enforcing approximation on these functions would not excessively change the quality of the output. This is a great example of how MAMBA determines the most efficient configurations for any benchmark regardless of how their floating point precision is specified in the source (or binary).

4.4.6 Function Call Stack

As we mentioned in section 4.2.1, if we map an FPI to a function, depending on the caller, the quality of output could change. While on most benchmarks, CIP and FCS approaches produce the same result, on the `radar` they differ. Hence, we examine the impact of the caller of the FFT function on the energy and accuracy of the benchmark. Figure 4.9 illustrates the FPU energy savings normalized to a baseline for CIP and FCS placement rules. FCS was able to explore a handful of more optimal configurations, resulting in 7% more energy savings at 1% accuracy loss comparing to CIP without extra runtime overhead. At 5% and 10% error rate, the additional energy savings are 4% and 2% respectively.

4.4.7 Input Sensitivity

Since we employ a heuristic exploration technique, we ensure that MAMBA produces statistically sound results by evaluating each application with multiple inputs divided into training and test sets. We take the median of normalized accuracy loss and FPU energy for each set of inputs, compute a linear least squares fit of training data to test data, and compute the correlation coefficient of each fit. Higher correlation coefficients imply less input sensitivity;

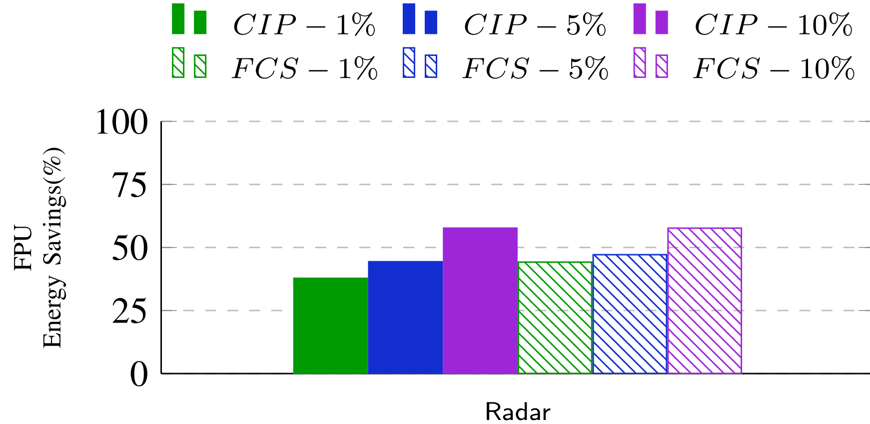


Figure 4.9: Comparison of CIP and FCS for the FPU energy savings in radar.

Table 4.3: Correlation coefficients for error rates and FPU energy.

Benchmark	**Error Rates**	**FPU Energy**
Blackscholes	0.999	0.999
Bodytrack	0.958	0.989
Fluidanimate	0.995	1.0
Ferret	0.973	1.0
Heartwall	0.999	1.0
Kmeans	0.932	1.0
Particlefilter	0.991	1.0
Radar	0.992	1.0

i.e. the behavior of configurations found during training data is a good predictor of test behavior.

Table 4.3 show the correlation coefficient (R-values) for accuracy loss and FPU energy for each benchmark. Due to the heuristic nature of the exploration technique, it might be possible to select configurations that perform differently on unseen data. For instance, `kmeans` clearly stresses the difference between training and test inputs. Although, all benchmarks have uniformly high R-values on accuracy loss and FPU energy—at least 0.93. This demonstrates that MAMBA's search techniques are robust and the accuracy and energy results they predict on training inputs hold up well for test inputs. The robustness of the energy results is, perhaps, not surprising as those should be highly predictable (simpler FLOP implementations should predictably lower energy). The robustness of the accuracy results is

perhaps more surprising as it not intuitively obvious that floating point implementations that work well for one set of inputs would also work for another set.

4.4.8 Neural Network Integration

The energy and resource constraints in neural networks create an intriguing challenge. More recently, a growing body of literature has tried to sacrifice the precision of training and inference for the lower runtime and energy consumption[22]. MAMBA can be used to identify the FLOP intensive sections of the network and then provide the minimum precision required for the computation without considerable model accuracy reductions. This tradeoff (small accuracy loss for large energy savings) is well known, and we perform this study not to claim a new result here, but to demonstrate that MAMBA's automated approach can produce the same types of savings for this problem that have been produced by human domain experts. We also believe that using MAMBA's programmable replacement rules to create CNNs with differing precision throughout the network is a new contribution that would (due to the size of the search space) be quite difficult even for human experts.

We use a hand-written digit classification with the MNIST dataset which includes 60K images and 10K labels. For the CNN, we consider the LeNet-5 model with the architecture summary listed in Table 4.4. The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier[56].

Figure 4.10 shows the FLOPs breakdown for CNN training with minibatch size of 4, learning rate of 1, and 30 epochs. We first measured how much of the operations are floating point to determine the applicability of MAMBA. For the inference, more than 73% of operations were FLOPs which makes MAMBA absolutely beneficial to apply. Next, we analyze the FLOP distribution between the layers. We observe that more than 69% of floating point computation happens in the convolutional layers where they extract interesting

Table 4.4: LeNet-5 architecture summary.

Layer		Feature Map	Size	Kernel Size	Activation
Input	Image	1	32x32	-	-
1	Convolutional(1)	6	28x28	5x5	tanh
2	Average Pooling(1)	6	14x14	2x2	tanh
3	Convolutional(2)	16	10x10	5x5	tanh
4	Average Pooling(2)	16	5x5	2x2	tanh
5	Convolutional(3)	120	1x1	5x5	tanh
6	Fully Connected	-	84	-	tanh
Output	Fully Connected	-	10	-	softmax

Table 4.5: Mantissa bits For single precision FP recommended by MAMBA for each layer at different error rates.

Layers / Error Rates	Conv 1	Avg Pool 1	Conv 2	Avg Pool 2	Conv 3	FC	Tanh	Internal Func.
1 %	10	23	14	4	19	4	20	17
5 %	10	5	5	16	13	4	18	15
10 %	6	16	12	9	13	1	17	11

features in an image. Activation phases and internal compute functions are responsible for the majority of remainder. Finally, we show that the number of FLOPs decreases as we approach the latter layers of the CNN since the size of transferred data between layers reduces as well.

To apply the FPI to function placement rules for a CNN, there are two options. First, apply one FPI per layer category (we refer to as PLC) meaning that all convolutional layers use the same precision level. The second approach is to apply a different FPI Per Layer Instance (PLI) where in this case the first and third layers might use distinct precision levels, however, they are both convolutional layers.

Picking the right FPI placement policy is not trivial for the CNNs. Unlike the WP versus CIP rules where one has significantly larger tradeoff space, the PLC and PLI tradeoff spaces are both large enough that heuristic exploration is required. Thus, any of these rules could outperform the other with the same exploration time. For the PLC, MAMBA explores a

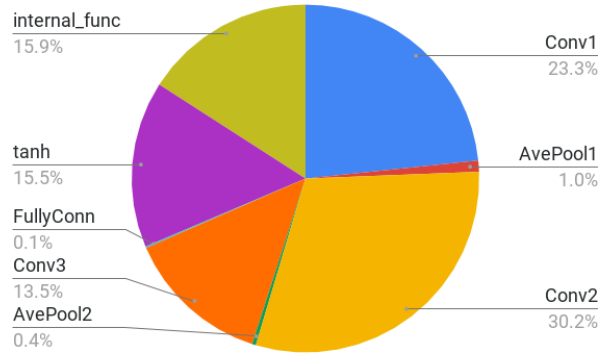


Figure 4.10: 32-bit FLOP breakdown per layer in digit recognition CNN.

larger portion of the tradeoff space, leading to locating efficient configurations more quickly. On the other hand, PLI examines FPI mappings at a finer granularity, hence it has a higher chance of discovering more optimal configurations.

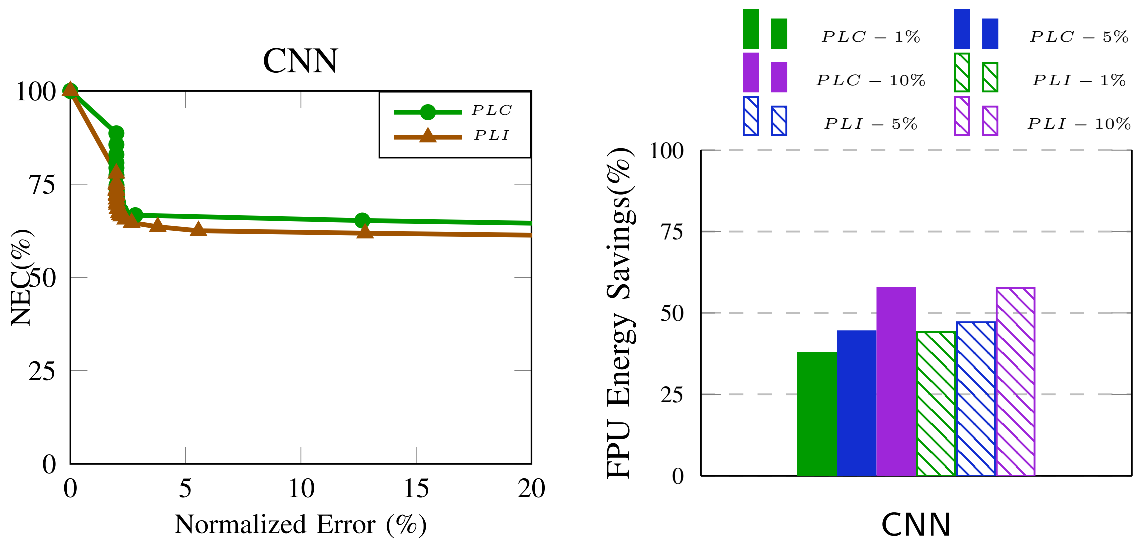


Figure 4.11: Comparison of PLC and PLC replacements for the CNN. (a) Lower convex hull curves of energy and error rate. (b) Quantized energy savings at different error rates.

Figure 4.11 illustrates the lower convex hull of normalized FPU energy and accuracy for both approaches. The accuracy loss is the error difference to the baseline configuration without approximation. The baseline recognition accuracy in the inference stage is 99.04% with a full accurate trained model. Each point in the tradeoff space demonstrates an FPI

to layer (category or instance) mapping. Closer points to the origin indicate higher energy efficiency.

As can be seen, the lower convex hull of the PLI (finer granularity) outperforms the PLC curve for the error rates of less than 20%. The quantized representation of FPU energy versus error rates tradeoff space is shown in Figure 4.11 for both PLC and PLI placements. Similar to previous evaluation, finer granularity results in higher energy efficiency. With 1%, 5%, and 10% accuracy loss, MAMBA with PLI placements achieves 6%, 4%, and 3% more energy savings compared to the default configuration.

MAMBA's programmable placement rules allow developers to analyze various precision levels for different components of their neural network without requiring them to instrument the source code or re-design the architecture.

Since the FPIs are based on the bit truncation of mantissa, using the above analysis, MAMBA finds the required precision bits for each layer in the LeNet-5 network under accuracy loss constraints. By default, each layer is implemented with single precision floating point numbers (24 mantissa bits) bits. Table 4.5 demonstrates the mantissa bits required for every layer in the network. These precisions could later be integrated with the MPFR library in C [31] or mpmath library in Python [45].

CHAPTER 5

COBRA AND ARCHITECTURAL APPROXIMATION

This chapter starts with architectural approximation and discussing approximate memory techniques, then followed by an overview of the COBRA (Cache Optimization via Bit Reversal Approximation) framework. Afterwards, COBRA runtime manual has been explained, and finally wrapped up with an evaluation of the unreliable data cache.

5.1 Background and Motivation

Architectural techniques are responsible to translate the hardware and circuit tradeoffs (approximation at processor and memory systems) to the software interface through constructs such as customized Instruction Set Architectures (ISA). Similarly, they expose information about hardware components to software to allow software to optimize the computations further. Therefore, architectural AC techniques play a critical role as the middleware in the cross-layer approximate computing paradigm.

Early work in architectural approximation relied on voltage overscaling as the primary mechanisms to deliver accuracy versus energy tradeoffs. Truffle [28] provides uncorrected nondeterministic errors in processor for higher energy efficiency. Truffle ISA provides compiler directives to specify approximate data and code. For non-approximate sections, a higher voltage is used to be error free while a lower voltage has been enforced on approximate portions.

Another set of articles suggest neural accelerators that implement a hardware neural network trained to emulate and accelerate applications. One method [27] proposes parrot transformation, a program transformation that selects and trains a neural network to mimic a region of imperative code. Based on the learned model, compiler replaces the original code with an invocation of neural processor unit (NPU) that is coupled to the processor

pipeline. Offloading approximable code regions to NPUs is faster and more energy efficient than executing the original code.

The MLP NN-based accelerators can be used for approximating the floating point transcendental functions (such as *cos*, *sin*, *exp*, *log*, and *pow*) as well [26]. In this technique, a NN accelerator is trained on limited range input (*e.g.* [0, pi/4] for *sin* function) followed by mathematically identities to compute the output of the mentioned functions. This approach aligns the Approximate Math Library that we evaluated in the Chapter 3. Instead of using Taylor series expansion at software level, a hardware neural network could be used for faster and more efficient computation.

Approximate Storage [82] trades the costly error control mechanisms with energy savings. Instead of detecting and correcting the errors, the approximate solid-state memory system allows controlled data errors happen in the application.

In general, approximation at data storage allows trading quality of output with modifications to memory elements or re-designing memory system hierarchy. Memory approximation techniques apply reduced read/write latency and access energy, fewer memory accesses, and reduced leakage power dissipation. These techniques target different memory components such as cache, DRAM, and even non-volatile memories.

Many memory approximation approaches are well-suited to this task, including but not limited to: Reducing the refresh rate of DRAM [47], voltage scaling at SRAM [33], memoization at different parallel lines of a SIMD architecture [73], skipping memory accesses [79], and partially forget-ful memories[86].

Approximate Storage techniques relax the costly reliability requirements of a memory system. Ganapathy et al. present a framework for minimizing the magnitude of errors when using unreliable memories [33]. They circularly shift the data for storing the least-significant bits at faulty cells of memory. Comparing to ECC, their framework provides more improvements in latency, power, and area. Moreover, it allows tolerating a limited number

of faults before invalidating the cell, which reduces the manufacturing cost compared to the conventional zero-failure yield constraint.

Another study analyzes the minimum error-protection required from a microarchitecture perspective [104]. They propose a Memory Fence Unit (MFU) to constraint memory accesses within the bounds of the legal address range allowed for a particular scope or instruction. If a memory access is out-of-range, the MFUs response for a data reference is to silence the fault either by skipping this memory instruction or by referencing a dummy physical location instead. Even with reasonably frequent errors in video and audio applications, their technique sustains good output quality and avoids hangs and catastrophic results such as segmentation fault.

Partially forget-ful memories provide cache energy savings by relaxing the guard-banding in memories [86]. Device manufacturers cover the expense of process variation by over-design and guard-banding the memories. Partially Forget-ful Memories (PFM) reduce the guard-banding to achieve higher energy efficiency, increased life span of memory component, and reduced memory access latency. However, the users should partition their data into critical and noncritical sections. On a cache miss, a victim block is selected from the relaxed ways if the data is noncritical. A critical data item is always stored in a fault-free block. Thus, voltage scaling and power-gating lead to leakage energy saving while guaranteeing the acceptable quality results.

In many memory intensive applications, memory costs are higher than the computations [12]. Looking back at the energy per instruction (EPI) in the Figure 4.1, memory instructions (specifically off-chip memory transactions) consume significantly more energy than computation instructions. In some cases, it is worthwhile to recompute the data rather than fetching them from last level cache or main memory. Thus, reducing data storage energy would allow higher efficiency in end-to-end approximate computing.

Hybrid approximate memories with energy efficient memory elements permits aggressive

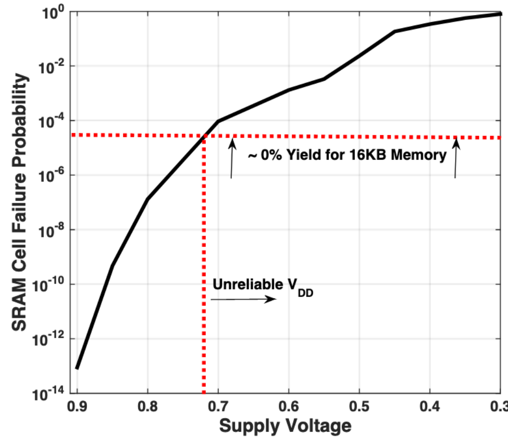
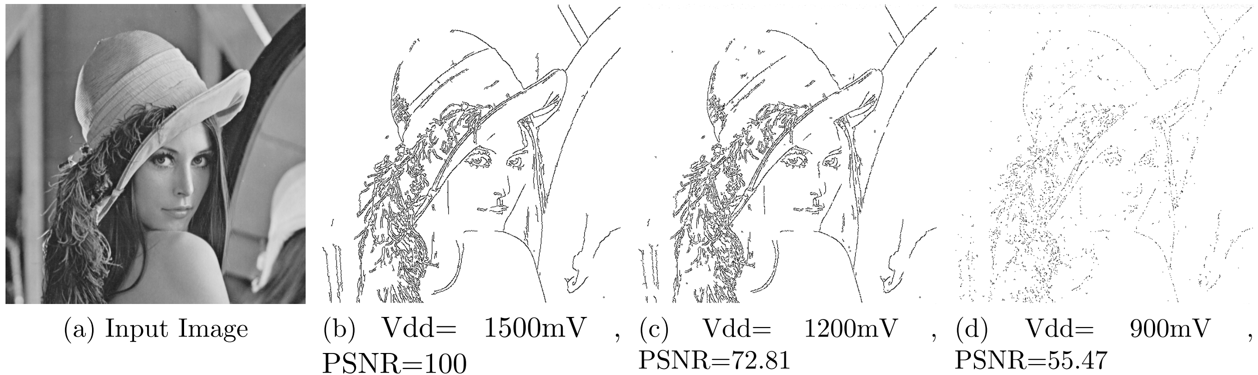


Figure 5.1: SRAM cell failure probability under voltage overscaling in 28nm process technology. (From [33])

voltage overscaling for improving energy efficiency. As an example, we illustrate the effects of unreliable energy efficient memory on the accuracy of an edge detection application. In a static random access memory (SRAM) there is a direct correlation between the supply voltage and cell failure probability. Figure 5.1 shows the total cell failure probability of a classical six-transistor (6T) SRAM cell in a 28nm processor under Vdd scaling.



(a) Input Image (b) Vdd= 1500mV , PSNR=100 (c) Vdd= 1200mV , PSNR=72.81 (d) Vdd= 900mV , PSNR=55.47

Figure 5.2: Unreliable memory and quality of output in canny application.

Since voltage scaling is the primary mechanism for energy savings at unreliable memories, we investigate SRAM supply voltage modifications on the quality of output. Figure 5.2 displays the quality of edge detection in `canny` benchmark at 3 different supply voltages for

level 1 data cache. The quality of edge detection is determined by Peak Signal to Noise Ratio (PSNR) metric. Higher PSNR demonstrates higher quality. As the voltage decreases, the quality of edge detector drops as well while less energy is consumed in data cache.

While approximate memory brings energy efficiency, it raises some concerns as well. Identifying the critical and noncritical portions of data and code is challenging. Approximating the pointers and addresses values may cause application crashes or hangs. Also, reducing supply voltage and disabling error correction methods causes random bit failures which could result in unpredicted behavior of the application. Furthermore, if the higher order bits fail (*e.g.* most significant bit (MSB) flips), substantial accuracy loss will be induced which subsequently invalidates many memory approximation configurations. Hence, COBRA employs protection mechanisms for critical data and MSB to ensure anticipated performance every time.

Most of the prior techniques designed and implemented their own simulation software or employed modular simulators such as gem5 or sniper. Differently, a runtime tool could be used to emulate the underlying memory system. This motivates us to design COBRA as a runtime tool to be platform-independent and provides a quicker and more convenient evaluation setup.

In conclusion, we design COBRA framework that provides cache energy efficiency, operates as a runtime tool, and allows critical and sensitive data protection. COBRA includes an interface to communicate with approximate computing techniques at programming language level (compilers and precision tuning tools like MAMBA).

5.2 Design

Energy efficiency in memory systems originates from improving the memory hierarchy, enhancing individual memory components, or augmenting the memory communication schemas. For volatile memories (such as SRAM and DRAM), voltage scaling reduces the static en-

ergy by inducing memory errors [28]. In DRAM, the energy savings come from reducing the refresh rate of specific DRAM rows which contain noncritical data. In SRAM, supply voltage is scaled down and Error Correction Codes (ECC) are neglected. COBRA focuses on improving level 1 cache energy as it has the most utilization rate and total energy consumption than other cache types. Also, since approximating the instructions and addresses are most likely to cause corruption, COBRA only targets the data cache. Thus, we refer to the approximate L1 data cache as the inexact cache for the rest of the paper.

Critical (precise) data are defined as the data whose correctness should be guaranteed and approximating them causes application to crash or catastrophic failure. Rest of the data and source code are considered as noncritical. The applications tolerate approximating the noncritical data and illustrate the output quality loss in response. There are software frameworks for automatically discovering approximable data by using statistical methods [77] or resilience characterization of the application [20]. Although, we believe the developers are the best candidates for the data partitioning as they have the best intuition about the flow of the application.

The SRAM cells do not reliably operate at lower supply voltage. There is a tradeoff between energy savings and bit failures. Figure 5.1 showed the relation between the bit failure (flip) probability (BFP) and supply voltage (Vdd). BFP is the probability that a bit flips during a memory read/write operation. BFP is considered as an approximation knob for the inexact cache. Internally, COBRA uses the energy model from [16] to convert the bit flip probability to SRAM supply voltage. The Vdd is necessary for measuring the energy consumption of inexact cache.

While using the unreliable memory, the magnitude of errors could increase rapidly and become an issue in the absence of the ECC methods. However, adding the costly ECC measures to reduce the unpredicted application behavior negates the energy efficiency from the inexact memory. One approach uses bit shuffling instead of commodity ECC methods to

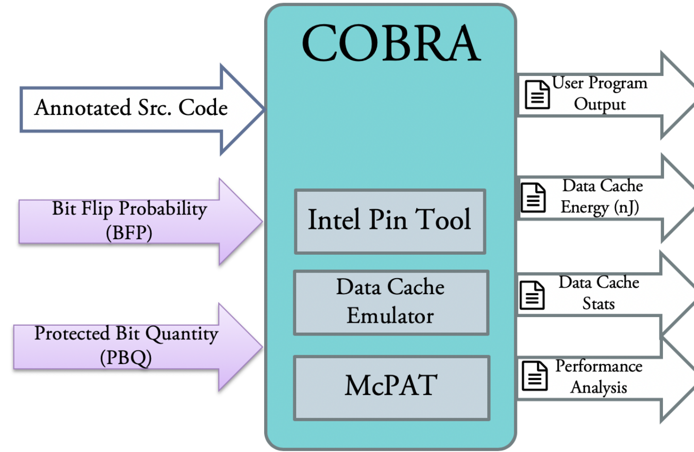


Figure 5.3: COBRA design.

ensure the balance between accuracy loss and energy utilization [33]. Also, their technique allows a limited quantity of faulty bits, which reduces the manufacturing cost comparing the zero-failure yield constraint.

To limit the error magnitude, instead of random bit shuffling, COBRA protects higher order bits by safe-guarding them from failures. In practice, we can have the different supply voltages for the memory and sub-blocks [8]. In COBRA, we define Protected Bit Quantity (PBQ) which contains the number of bits (starting from most significant bit) that are secured from failures. If PBQ is 0, this means any bit could flip during the memory operation. If $PBQ = 1$, the MSB is protected. Similarly, if $PBQ = 4$, only bottom half of a byte might encounter failures while top half remains safe. As we increase the PBQ, the effect of BFP (indirectly voltage scaling) disappears. This creates a tradeoff between controlling the error amplitude and energy efficiency. Hence, PBQ counts as another approximation knob for the COBRA.

Figure 5.3 illustrates the COBRA framework internal structure from the user perspective. Users specify annotated source code, bit flip probability (BFP), and the number of protected bits (PBQ). COBRA then executes the application, injects the faults to the inexact cache based on the given BFP and PBQ using the Pin tool, calculates the data cache energy with

McPAT and eventually provides energy, accuracy loss, and performance measurements.

The Intel Pin Instrumentation platform provides instrumentation routines for every memory operation [59]. Instead of evaluating the inexact cache on a simulator, Pin tool brings runtime emulation of the underlying memory system.

COBRA expects an annotated version of the input application where the noncritical data is specified. If the user is unaware of the internal structure of the application, COBRA is able to help by providing data cache profiling report. COBRA generates a heat map of data cache accesses based on the virtual addresses. The user could find the corresponding variable to any specific address, and then apply approximation on that address range.

A set of address ranges are given to COBRA to be considered for the fault injections. If the memory operation address falls into any of the defined address ranges, then the memory operation should be approximated and COBRA applies BFP and PBQ mechanisms on the operation. COBRA for each bit generates a random probability and if the probability is less than BFP threshold, then the bit is considered as failure. If the bit is not protected by the PBQ, then the value of that bit would flip and the updated value would be returned. Otherwise, the actual value is sent back.

While Pin tool identifies the memory operations, it can not distinguish between a hit and a miss access at the data cache. Thus, we implemented a simple cache simulator within the COBRA to mimic the memory hierarchy. COBRA gathers the information about the memory elements (size of L1 data cache, block width, and associativity) through the operating system profile. As memory operations arrive, the data cache emulator keeps track of memory operations, determines if it is a miss or hit, and at the end of the execution, reports the hit/miss stats of the inexact cache.

To calculate the energy utilization of the data cache, COBRA has a built-in McPAT framework. McPAT is an integrated power, area, and timing modeling framework for multicore and manycore architectures [57]. To calculate the inexact cache, McPAT requires data

cache statistics (total accesses and hit/miss ratio) and the supply voltage. As mentioned before, COBRA uses analytical models to convert the BFP into Vdd value. McPAT also depends on processor description files which includes information about the micro-architectural parameters of the current processor to compute the accurate energy consumption. To have a precise energy utilization info, the user might review the micro-architectural parameters.

COBRA generates four outputs: the output from the user application, estimated L1 data cache energy/power from McPAT, data cache hit/miss stats, and the performance data.

COBRA provides a simple, yet effective emulation of inexact cache. It comes with internal precise energy modeling framework and can be used as a profiler to help users to data partition and annotate the application.

5.3 Interface and Runtime Manual

In this section, the interface of COBRA and how a user could operate it to evaluate unreliable memory is explained. First, the input parameters of the COBRA is described, followed by the step by step manual.

The COBRA procedure is:

1. **Profile the application**: (optional) If the user is unaware of approximable data in the application source code, (s)he might first run the application without setting approximation knobs to identify the noncritical data partitions. COBRA provides heat map of memory accesses and the corresponding data variables.

2. **Specify Data Partitioning**: The data criticality should be defined by the user as the main input of COBRA. Considering the developers insight into the application and the profiling stage, noncritical data should be identified to be mapped to the unreliable portion of data cache. COBRA expects a set of range of addresses as the representatives of the noncritical data. The programmer uses `Add_approx` function from the COBRA interface to declare noncriticality dynamically within the code. Depending on the runtime phase of the

```
float* image;
image = (fp*)malloc(sizeof(float) * 640 * 480);

#ifdef APPROX_on
 add_approx((unsigned long long)image,
    (unsigned long long)(image + sizeof(float) * Ne));
#endif
```

Figure 5.4: Sample code snippet of using COBRA for data partitioning.

application, the developer might cancel the approximation. The `Remove_approx` function eliminates approximation on the fly.

Figure 5.4 includes a code snippet of declaring approximable data, executing computation on the data, and omitting the approximation in the end.

3. **Configure the inexact cache**: COBRA provides two approximation knobs for the inexact cache, bit failure probability (BFP) and Protected Bits Quantity (PBQ). Higher BFP translates into lower energy utilization but less accurate results. Higher PBQ results in more energy consumption and higher quality outputs. These knobs can be set statistically through command line parameters `bfp` and `pbq`. They also can be modified dynamically through the `set_bfp` and `set_pbq` functions. Depending on the application phase and user goals, the approximation level might change during the execution. By default, $BFP = 0$ and $PBQ = 0$ which means no approximation would happen.

4. **Review McPAT Configuration**: (optional) McPAT calculates the energy/power of the inexact cache. Parameters of the underlying hardware of the host system need to be set in the processor description file. McPAT have very detailed parameter settings in the $*.xml$ interface file for entries tagged as `param`. The description file includes architectural and microarchitectural parameters such as core's properties, memory hierarchy, and network on chip. The user might review the parameters before running the COBRA.

5. **Analyze the output**: COBRA reports inexact data cache performance and energy stats, the application performance and runtime data, and the output from the user application (which can be fed into accuracy measurement tool separately).

Table 5.1: Host system architectural parameters.

Parameters	ISA	CPU Model	Cores	Repl. Policy	Cache Size	Cache Block Size	Assoc.
Values	x86	Detailed	20	Random	32 KB	64 B	$8-WAY$

5.4 Evaluation

This section evaluates COBRA framework by testing the approximation knobs for the inexact data cache on a set of benchmarks and then, analyzing energy, accuracy, and performance output.

5.4.1 Experimental Setup

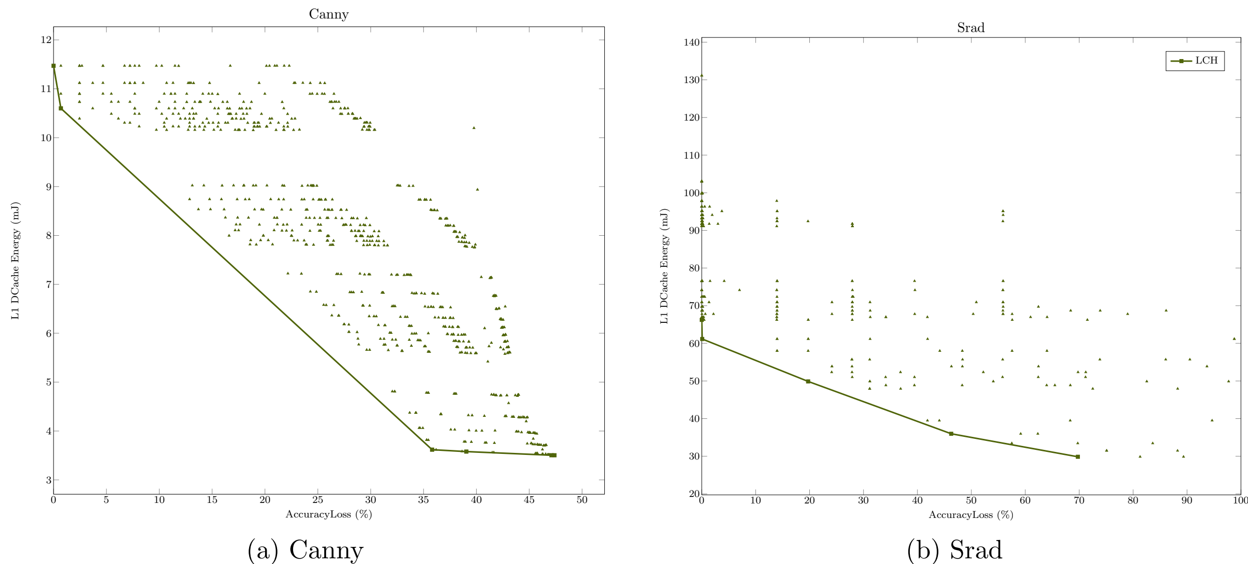
We use an Intel Xeon E5 server system is used as the evaluation platform for testing COBRA. Table 5.1 illustrates the architectural parameters of the examination host platform. We used the *Random* block replacement policy in the internal dcache emulator.

The technology size of the processor determines the supply voltage range for the SRAM. Hence, at different processor types, different BFP would be expected. In our evaluations, the range of BFP is $[10^{-6}, 10^{-2}]$. The PBQ knob works at byte scale, protecting bits from the most significant bit to the lowest significant bit. Thus, range of PBQ is $[0, 7]$. In total, we examined 864 configurations of inexact cache via COBRA. A pair of $< BFP, PBQ >$ is considered as a configuration. Each configuration provides an output accuracy and data cache energy consumption in nJ unit.

The streaming benchmarks are a compelling target for unreliable memory evaluation. They have temporal locality of memory accesses as they process a sequence of frames or inputs. The approximation of temporal accesses limits the possible crashes or substantial output quality loss of the application. We investigated the inexact cache via COBRA framework on `canny` and `srad` image processing application. Also, multiple runs have been performed to analyze the unpredicted behavior of the application using unreliable memory.

For each configuration, we used the median of measurements.

5.4.2 Tradeoff Spaces and Lower Convex Hull



(a) Canny (b) Srad

Figure 5.5: Tradeoff space of inExact cache on benchmarks.

Figure 5.5 illustrates the tradeoff spaces of inexact cache for the benchmarks. The vertical axis presents the L1 data cache energy consumption and the horizontal axis show the accuracy loss (error rate) of the output. The closer the point is to the origin, the more efficient is the configuration. These figures display the range of accuracy loss and energy consumption that inexact cache provides. Also, they show the extension and distribution of the configurations in the tradeoff space. At 20% error rate, COBRA provides 45% and 32% energy reduction comparing to the default configuration.

5.4.3 Energy Efficiency and BFP

Voltage overscaling at COBRA is translated to bit flip probability. The higher the BFP, the lower is the voltage and the cache is more susceptible to failures. Figure 5.6 summarizes the energy utilization at different supply voltages with regarding to protected bits for the

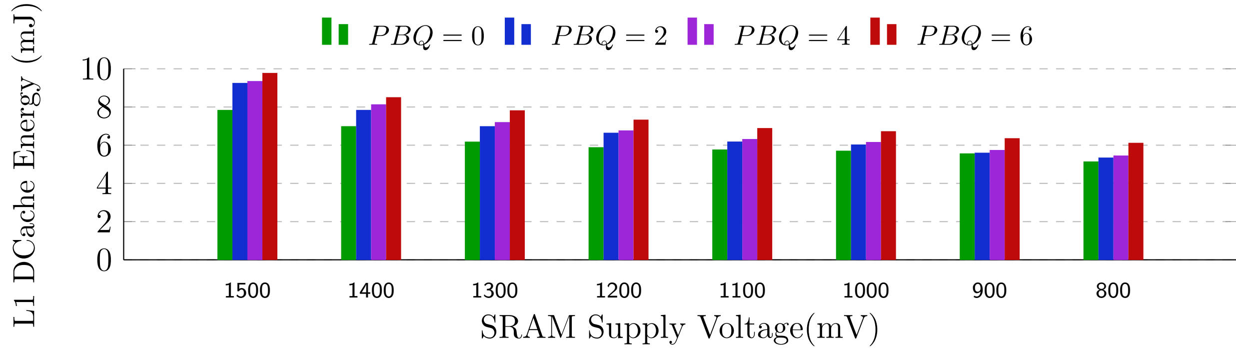


Figure 5.6: Energy efficiency at various supply voltages and bit protections.

`srad` application. The baseline Vdd is 1500 mV and power-gating is performed on protected blocks. The lower the bar, the more energy efficient is the configuration.

As expected, by reducing the voltage, the higher energy savings are achieved. Up to 34% of data cache energy could be saved if the supply voltage reduces by almost half.

5.4.4 Energy Efficiency and PBQ

Same setup as the previous section, we analyze the relationship of accuracy loss and PBQ values instead of targeting bit flip probabilities. Figure 5.6 also shows the fidelity analysis of the `canny` application. As expected, increasing the PBQ leads to higher quality output images, yet more energy consumption. Another observed trend is how energy consumption increases as more bits are protected. Expanding the BPQ is considered as more costly ECC procedure in practice, thus higher energy consumption. If the top half of a byte is protected from the failures, the energy consumption escalates by 19% in return.

5.4.5 Accuracy Variation and Protected Bits

While the unreliable (inexact) memory provides energy efficiency, in many configurations could end up with unacceptable output quality loss. If the higher order bits flip, the more accuracy loss is expected. We performed multiple experiments for each configuration and

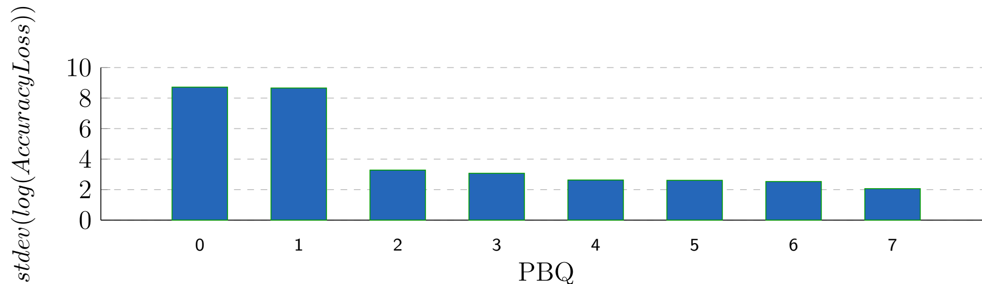


Figure 5.7: Controlling unpredicted behavior of inExact cache with bit protection.

measured the accuracy loss per function. For the `srad` benchmark, since the accuracy is measured using pixel by pixel comparison, the actual error rate could become a huge number. Hence to show the variation of output accuracy, the standard deviation of the logarithm of error rates is computed.

Figure 5.7 shows the average of $stdev(log(accuracyLoss))$ over all configurations at different PBQ values for the `srad` application. Without protection, the variance of output quality is substantial. As PBQ increases, the variation disappears.

This figure advocates the importance of the PBQ in the inExact cache. Higher error magnitudes are usually undesirable for the users and the accuracy loss should be constrained. The PBQ knob provides such feature.

CHAPTER 6

HARDWARE AND CIRCUIT APPROXIMATION

This chapter presents hardware and circuit approximation techniques. A reduced-precision FPU is constructed using the MAMBA's precision tuning analysis. We evaluate the inexact FPU on OpenPiton processor with the same set of benchmarks used in MAMBA evaluation.

6.1 Background and Motivation

At hardware and circuit level, approximations rely on timing relaxation or functional units enhancements [39].

The first one refers to voltage overscaling in electronic elements such as transistors. Let Vdd is the supply voltage for the CMOS transistor, f the clock frequency, and C the capacity of the load of the circuit. Then, the dynamic power dissipation P would be [72]

$$P \propto CV_{dd}^2 f \tag{6.1}$$

Scaling the supply voltage changes the frequency f as well. If Vt is the threshold voltage for the circuit, and $alpha$ is the processor independent variable, then the f is related to Vdd as

$$P \propto \frac{(V_{dd} - V_t)^\alpha}{V_{dd}} \tag{6.2}$$

Hence, by reducing the supply voltage, energy efficiency is achieved but timing errors happen too which leads to output accuracy loss. Probabilistic CMOS exploits the intrinsic thermal noise of CMOS circuits to enhance the probabilistic algorithms which use a source of entropy inside [58]. Stochastic computing (SC) uses a data representation of bit streams that denote real-valued probabilities [2]. The unary encoding of numbers on SC permits variable precision computation that is applicable in approximate computing [3].

The second set of hardware approximation techniques focuses on pruning circuits that means simplifying parts of circuit based on probability of usage or importance to output quality [89]. For example, approximating the logic of 1-bit full adder [38] or carry propagation [98] provide inexact results within defined error bounds.

GeAR is a generic accuracy configurable adder that provides a generalized model of build multi-bit approximate adders [83]. GeAR also accounts for reducing carry propagation error through overlapping sub-adders and a light-weight error correction method. Register Transfer Level (RTL) optimizations improve area efficiency as well as energy. SALSA offers a systematic method that encodes the quality constraints into functional units [93]. It automatically generates and synthesizes Approximation Dont Cares (ADCs) in circuit simplification of imprecise components ranging from arithmetic building blocks (adders, multipliers, MAC) to entire datapaths.

Neural networks expose significant parallelism and include immense floating point computation. Adding dedicated hardware support via Neural Processing Units (NPU) provides considerable energy and performance improvements. NPUs deliver low-power inexact functionality and have been developed for GPUs [103] and FPGAs [69]. For example, Google Edge Tensor Processing Unit (TPU) is a specific purpose-build ASIC chip that offers a peak throughput of 92 Tera Ops per second [46].

Reduced precision floating point optimizations are available through software bit width reduction [34, 78], compiler and ISA level [1, 40], and enhanced FPUs [48]. For the case of NNs, low-precision floating point schemas provide up to 4x performance gain comparing to traditional 32 bit systems [79].

Recent proposals promote putting many different approximate units or customized accelerators on a single core [32]. While an FPU accounts for 2-5% area on the chip, the floating-point instructions consume significantly more energy compared to other classes of instructions such as integer, memory, and control as well [10, 64].

Table 6.1: Power consumption of single precision FP multiplier (From [91])

Functional Block	Power Dissipation (% of total)
Mantissa Multiplier	81.2
Rounding Unit	17.9
Exponent Unit	0.833
Others (Exception Handling and etc.)	0.066

In the presence of multiple FPUs on chip where they expose different precisions (8 FP bits [65], 12 bits [70], 16 brain float bits in addition to the traditional 32 FP bits [48]), the challenge is determining the level of approximation to apply on occurrence of each FLOP. The MAMBA tool provides such a solution as it has a comprehensive perspective of floating point operations at the runtime of the application.

To exploit the MAMBA information at the hardware level, the corresponding FPU and datapath should be re-designed accordingly. An improved FPU is synthesized to examine the MAMBA findings on the actual hardware test platform.

6.2 System Design

The energy cost of FPUs are prohibitively expensive. Minimizing the bitwidth on FP representations results in significant energy savings.

Table 6.1 shows the power dissipation distribution in a single precision (32 bits) multiplier [91]. The mantissa bits are responsible for more than 80% of energy consumption. Our reduced-precision FPU follows the same procedure by masking the mantissa bits only. Exponents bits are accuracy sensitive and consume less than 1% of total energy utilization.

OpenPiton is an general purpose, many core processor and framework for building scalable architecture research prototypes. It provides open source across the entire computing stack, from the hardware to the firmware and software. OpenPiton leverages OpenSPARC T1 core with modifications and builds upon it with a scratch-built, scalable uncore.

To create a reduced-precision FPU, we modified the RTL source code of the FPU to

```
parameter MBIT_WIDTH = 32 + l0;
parameter MASK = {64{1'b1}} << MBIT_WIDTH;


assign fp_cpx_data_ca_76_0_in[76:0]= ({77{dest_rdy[2]}}
                & {div_exc_out[4:0], 8'b0, {div_out[63:0]} & MASK})
        | ({77{dest_rdy[1]}}
                & {mul_exc_out[4:0], 8'b0, {mul_out[63:0] & MASK}})
        | ({77{dest_rdy[0]}}
                & {add_exc_out[4:0], 2'b0, a6stg_fcmpop,
                add_cc_out[1:0], add_fcc_out[1:0], 1'b0,
                {add_out[63:0] & MASK}});
```

Figure 6.1: Applying bitwidth reduction in the source sode of openPiton.

mask the mantissa bits based on a constant value stored in register file. OpenSpark T1 core provides address space identifiers (ASIs) that are useful for exposing internal registers to user-level and privileged software. ASI accesses look very much like normal load and store instructions to the assembly programmer.

```
#define ASI_EXAMPLE     0x1a
setx 0x08,         %g2, %g1
setx 0xdeadbeef0, %g2, %l0
stxa %l0, [%g1] ASI_EXAMPLE
ldxa [%g1] ASI_EXAMPLE, %l1
```

The code fragment above write a value in architectural register $\%l0$ to virtual address (VA) 0x8 at ASI 0x1A (needs to implemented independently) and then reads from the same location into architectural register $\%l1$.

In the FPU structure, mantissa bits are masked based on the stored value in the $\%l1$ register. The Verilog code fragment above in the Figure 6.1 shows the implementation of masking the result of a FLOP in the source code of OpenSparc T1 core.

As the FLOPs are directed to FPU, the desired mantissa bitwidth is applied to the inputs and output of the operation the same as the procedure taking place in MAMBA with Pin tool. Note that we do not modify the FP datapath components to align with the updated bitwidth as it brings substantial design complexity.

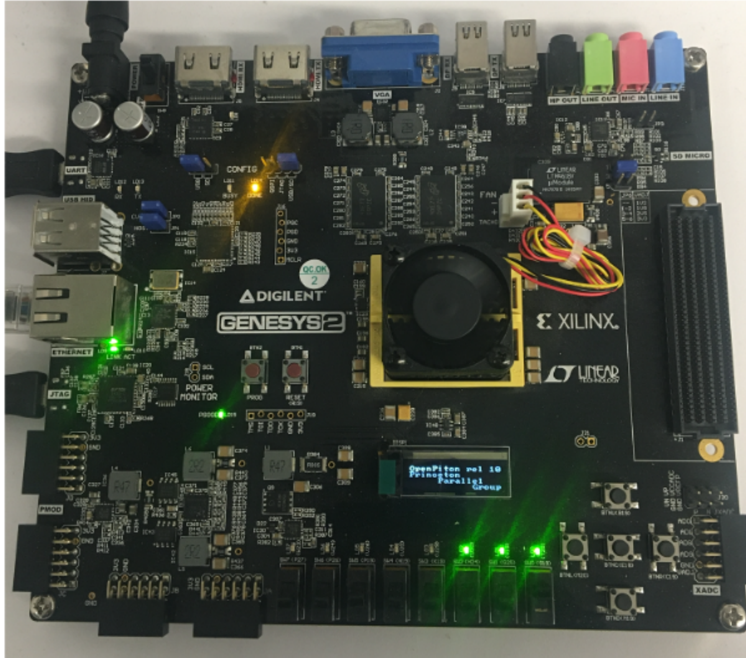


Figure 6.2: Loaded OpenPiton processor on the Kintex FPGA.

6.3 Evaluation

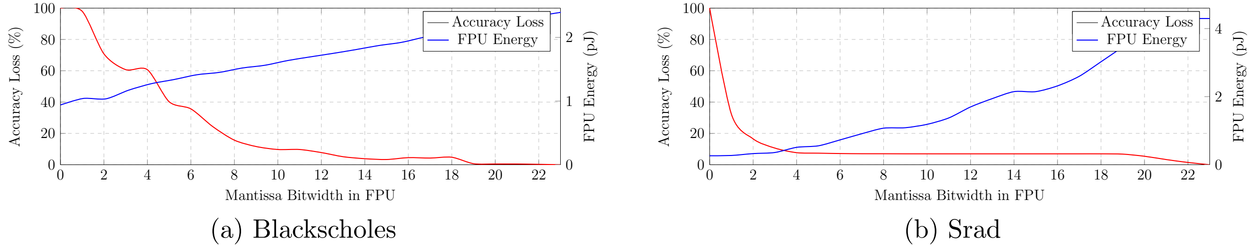
To evaluate the reduced-precision FPU, an OpenPiton processor with different bitwidths have been synthesized. Then, we port the processor on a Kintex-7 (Digilent Genesys 2) FPGA.

After verifying the functionality of FPU using bare metal test unit files, we boot full stack Debian Linux on the board to prepare the testbed for evaluation of our benchmarks. Figure 6.2 shows the hardware evaluation setup for analyzing the reduced-precision FPU.

Since OpenPiton is based on the OpenSparc core, all the application binaries should be cross compiled for the SPARC architecture. The size of the input dataset has been adjusted to accommodate the synthesized core speed and storage.

6.3.1 *FPU Energy and Accuracy Loss*

We evaluate the energy efficient FPU by measuring the accuracy and FPU energy for different mantissa bitwidth per application.



(a) Blackscholes

(b) Srad

Figure 6.3: FPU energy and accuracy loss of selected benchmarks on reduced-precision FPU.

The results of reduced-precision FPUs are shown in Figure 6.3. As the number of mantissa bits decreases in the FPU, the error rate goes higher, but the energy consumptions decline. Unlike the multiplier from [91], the energy consumption is not linear to the bitwidth. These plots correspond accurately with the MAMBA evaluation.

To put everything in conclusion, these chapters demonstrate a practical example of cross-layer approximate computing. The precision tuning report of MAMBA is exploited to build an inexact hardware function component.

CHAPTER 7

TOWARDS CROSS-LAYER APPROXIMATE COMPUTING

This chapter concludes the cross-layer approximate computing. First, existing research and the motivation for end-to-end AC are explained. Next, a holistic framework is introduced that connects approximation techniques from multiple layers of the system stack. For the evaluation, BOA family is applied to examine cross-layer combinations of approximations. We wrap it up by discussing the challenges ahead of enabling holistic cross-layer approximation.

7.1 Background and Motivation

Today, the computing stack paradigm has shifted towards enforcing approximation at all layers of the system stack. Traditionally, the applications and user perceptual in addition to sensors and devices were the only targets for the approximations. All the mid layers were assumed to be completely precise and provide exact numerical and Boolean equivalences. AC frameworks have relaxed all the layers to tolerate a level of incorrectness in return for performance and energy efficiency.

While a plenty of work has been done on approximating the isolated system layers, considering the errors and dependency between approximations at different layers leads to a better intuition of possible approximation opportunities and subsequently higher accuracy-energy tradeoffs.

In Section 4.4.8, we performed precision tuning with MAMBA on a digit recognition application that was implemented with a convolutional neural network. MAMBA framework belongs to the programming language layer. Other AC opportunities exists for CNNs at other layers of system stack. For example, at software layer, the domain specific solutions such as random dropout and weight quantization substantially reduce the computations. At the

architecture level, the approximate SRAM (Minerva) [75] could be used to store the quantized weights. At the hardware level, the neural processing units provide significant energy and performance improvement for the sparse matrix computations. While there are efforts at different layers, connecting them through a holistic framework pushes the accuracy-efficiency tradeoffs further.

In the context of cross-layer AC, we classify the prior work into two categories. First, the survey articles which provide an overview of AC frameworks at different layers and discuss fundamental limits and benefits of end-to-end optimizations [6, 67, 76, 84, 89]. Second, the domain specific approaches [85] that connect two or more layers are focused on a single target such as floating point optimization or neural networks specific enhancements. For instance, Truffle [28] is a multi-layer AC framework which uses voltage overscaling at hardware level to implement approximate operations and storage. At architecture layer, Truffle provides ISA extensions to allow compiler to specify data criticality. By combining Truffle with software probabilistic approximations or including customized accelerators (to replace original functional components), higher efficiency should be expected. Unlike others, our generic framework (with BOA exploration) could combine any AC framework regardless of their domain or system layer.

To enable the cross-layer AC, the domain knowledge and algorithm specific optimization should translated into tradeoffs that is comprehensible by different approximate components at different layers. Complementing techniques leverages efficiency improvements on the redundancy. For example, any precision tuning optimization (MAMBA) could use a reduced-precision FPU at the hardware layer (OpenPiton Processor).

Many of software approximation techniques employ high level models to emulate the underlying platform or estimating the energy and error values. Often many assumptions have been made for simpler evaluation. To make credible claims of cross-layer AC, the end-to-end approximation should be implemented and evaluated through all the layers, either on

actual hardware or realistic simulators [89]. Our framework not only includes synthesized hardware component, but also incorporates different approximations independent of floating point optimization.

While cross-layer AC seems promising, involving multiple frameworks might raise scalability issues as the design space size significantly increases. For example, although hardware accelerators bring considerable improvements, they still need to be configured which results in additional design complexity. Our BOA exploration techniques are responsible to deal with immense combined tradeoff space.

7.2 Design

Different layers of system stack focus on different optimization goals. While higher levels provide computation efficiency, the lower layers often target energy or storage efficiency. In other words, software approximations reduce the number of computation operations and lower layers reduce the energy of individual operation. Combining approximations across the stack provides substantial energy and performance improvement opportunities.

There are multiple potential connections between AC frameworks discussed in this dissertation. For example, the Approximate Math Libraries includes numerous FLOPs. Attaching it to MAMBA to accommodate custom bitwidth allow higher efficiency. While these techniques complement each other, at the same time, they could operate negatively as well. Calculating a *sqrt* function with many Taylor series expansions could be disproved if the MAMBA mask out the majority of mantissa bitwidth.

We evaluated BOA in Section 3.3.1 with three software approximation frameworks. To enable cross-layer AC, we expand it to include PL approximation (MAMBA) and architectural approximation (COBRA).

Figure 7.1 shows an overview of our holistic framework where multiple techniques from different layers are incorporated together. The approximation frameworks include three

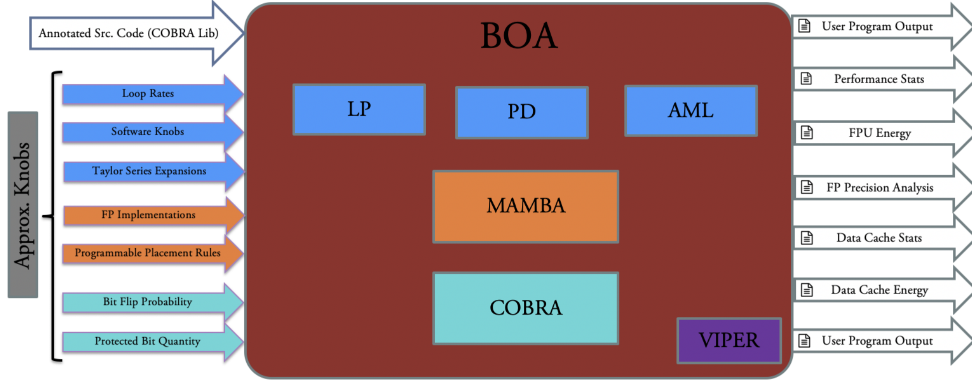


Figure 7.1: Cross-layer AC with BOA.

software (LP, PD, and AML), one programming language (MAMBA), and one architectural (COBRA) framework. BOA is used to locate the Pareto-efficient configurations in the combined tradeoff space. VIPER provides more satisfying representation of tradeoff spaces. The approximation knobs are the aggregate knobs of internal AC frameworks. The outputs provide performance and energy stats from all layers of the system stack.

7.3 Evaluation

In this section, we evaluate cross-layer combination of approximate techniques with BOA and NSGA in a similar procedure as Chapter 3. First, size of search space in each approach is compared. Next, tradeoff space and Pareto-efficient curves are compared. Afterward, we use the VIPER to compare the optimality of located configurations by BOA and NSGA. We finally wrap up the evaluation by analyzing the effect of each AC technique on the located Pareto-efficient configurations.

7.3.1 Evaluation Setup

For the cross-layer AC evaluation, we only examined `srad` benchmark as it was tested through all layers of the system with different approximation frameworks. Next, we combine the

Table 7.1: Number of explored configurations in cross-layer AC combination.

Expl. Technique	NSGA-II	BOA-simple	BOA-flex	BOA-prob
Configurations	2,400	2,640	15,400	18,000

individual AC frameworks and a new immense tradeoff space is created which requires NSGA-II and BOA family to locate the Pareto-efficient configurations. Due to the employment of the unreliable memory, multiple inputs and runs have been performed to ensure the robustness of results.

7.3.2 Comparison by Size of Search Space

Table 7.1 demonstrates the number of configurations that each exploration technique locates. While NSGA-II and BOA-simple are in the same order, the expansions of BOA search significantly more configurations. Since we are combining numerous frameworks, the configurations near the Pareto-efficient curves are too many. Thus, even a small expansion of near-optimal configurations results in an explosion of configurations.

7.3.3 Comparison by Configuration Space

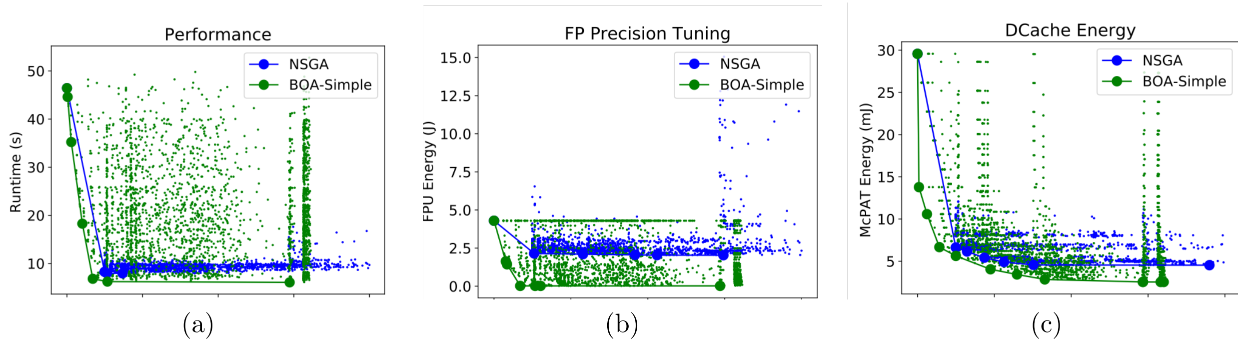


Figure 7.2: Tradeoff space and Pareto-efficient curves of NSGA-II and BOA-simple in the cross-layer AC.

In cross-layer AC, multiple frameworks from various layers are combined together to deliver performance and energy efficiency. Figure 7.2 present tradeoff spaces of cross-layer

BOA and NSGA-II while combining AC frameworks depicted in Figure 7.1. Instead of merging efficiency metrics to build a single comparison metric, we present three points of comparison as the internal AC frameworks have different optimization target. The software approximations improves performance by reducing the runtime. MAMBA enhances the FPU energy consumption and COBRA provides lower energy utilization at data cache.

The x-axis shows accuracy loss in all subfigures and y-axis from left to right are runtime in seconds, FPU energy consumption in J, and inExact cache energy utilization in mJ respectively.

In all subplots of Figure 7.2, BOA at its simplest version outperforms NSGA-II by locating more optimal configurations. NSGA-II is susceptible to get stuck in local optima and this is clearly observed in performance and FPU energy consumption graphs. While BOA illustrates higher sparsity in the tradeoff space, NSGA-II explores configurations around a certain curve which is not even as optimal as BOA-simple.

7.3.4 Comparison in BOA Family

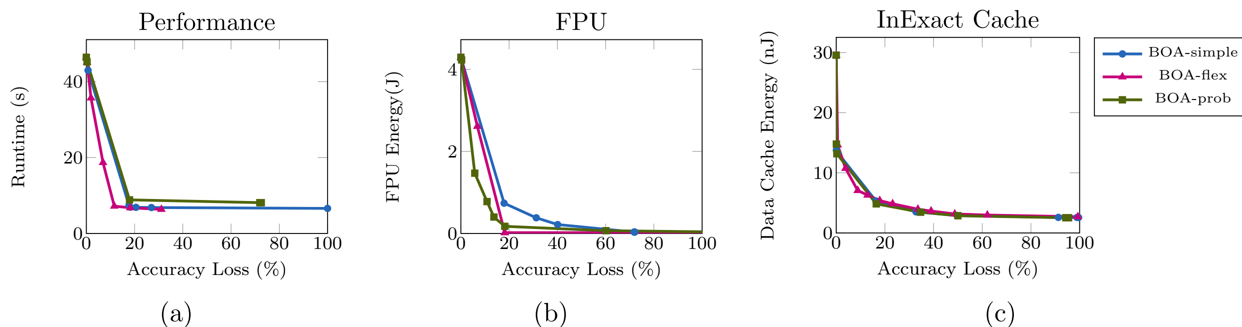


Figure 7.3: Lower convex hulls of BOA variations in cross-Layer AC.

In previous experiment, we observe that BOA-simple was able to locate more Pareto-efficient configurations in the tradeoff spaces comparing to the NSGA-II with the same exploration time. While BOA-simple is uniformly better than NSGA-II, in this section we

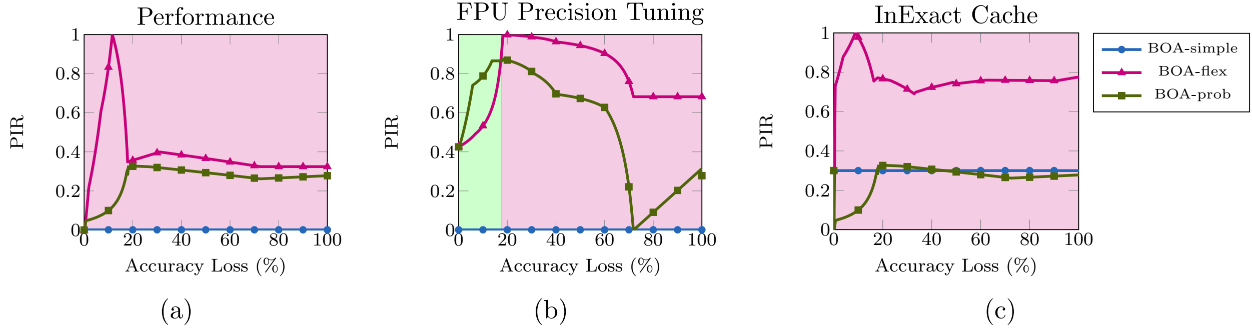


Figure 7.4: VIPER plots of BOA variations in cross-Layer AC.

examine expanding the BOA configurations further with deterministically and probabilistically known as BOA-flex and BOA-prob respectively.

Figure 7.3 shows the lower convex hulls for different version of BOA. Same as before, we evaluate the efficiency with three points of comparisons. The horizontal axis is accuracy loss (%) as expected, while the vertical axis differs.

All plots exhibit the same general pattern. Expanding the search space with BOA-flex and BOA-prob results in higher efficiency. Although, comparing the efficiency margins to the previous BOA evaluation (in Section 3.3.1), cross-layer AC indicate smaller improvements.

This is not surprising as the more frameworks are combined together while they might interfere with the efficiency of other frameworks. We analyzed a case where AML and MAMBA could both negate and complement each other's efficiency. Also, however the unpredictable behavior of unreliable memory is controlled with PBQ approximation knob, it still affects other AC frameworks in the combination process.

Figure 7.4 illustrates the VIPER plots of the tradeoff spaces of BOA variations. In most cases, BOA-flex shows superiority over the BOA-simple and BOA-prob. For a short range of accuracy loss, BOA-prob outperforms the BOA-flex in terms of FPU energy efficiency. Also, the inExact cache energy consumption plot indicates that adding more random configurations to the search space does not provide much more efficiency. Even for accuracy losses of less than 20%, BOA-simple locates more efficient configurations.

Table 7.2: Unique configurations from each individual AC framework found on the Pareto-efficient curve.

Expl. Technique	Pareto-efficient Configs.	PD	LP	AML	MAMBA	COBRA
BOA-simple	10	4	2	3	4	4
BOA-flex	10	1	4	4	8	4
BOA-prob	8	1	3	3	7	3

A higher efficiency margin becomes available if domain specific knowledge and potential connections between AC frameworks examined throughly. In this case, many conflicting configurations would be eliminated from the search space.

7.3.5 Combination Distribution

We previously observed that even if an individual frameworks is uniformly worst than others, in the combination process, it might become useful as the AC frameworks interaction is extremely complicated. Table 7.2 includes the number of configurations found on the Pareto-efficient curve of the combined tradeoff space and the number of unique configurations that each AC frameworks contributes.

The configuration distribution shows that there is substantial benefit of combining AC frameworks from different layers as they all contribute to the located Pareto-efficient configurations.

To conclude, we evaluated BOA to combine the AC frameworks from different layers of system stack to build the first holistic cross-layer AC. The ultimate goal is to enable hardware and software coordinated approximation. The key challenges ahead are finding the potential connections between frameworks and then transferring the output quality constraints through the layers considering the error propagation and data resilience.

CHAPTER 8

FUTURE WORK

While this dissertation provides holistic framework for cross-layer approximate computing, more avenues to extend this work are available.

In this thesis, at architecture and hardware level, energy efficiency is achieved by reducing the mantissa bits of the floating point representation while the datapath design is untouched. For the ultimate coordinated approximation, the storage, computing, and algorithm constructs used should all be approximate. Compiler and ISA extensions should be included as well to support testing, debugging, and dynamic quality monitoring of cross-layer approximation.

Approximate computing is used more and more in the applications running low-power and embedded systems which makes the energy efficiency as the main target. Many hardware error and energy models are based on numerous assumptions about the architecture of underlying platform. A precise examination is feasible if the energy measurements are accurate and produced with fine grained power meters.

Domain specific knowledge could significantly reduce the computation and subsequently provide energy efficiency. Neural Networks and machine learning techniques lack the ground truth result, and hence mathematical and algorithmic optimization are simply applicable to them. While MAMBA provides FP precision tuning for NNs, it can be merged with NN optimization techniques such as Random Dropout [88], axNN [94], and Weight Quantization [21] to achieve higher energy and resource efficiency.

To handle intractable tradeoff spaces, both Genetic Algorithms (GA) and Reinforcement learning (RL) are proposed as they search for solutions that maximize or minimize an objective function. Since GAs have a tendency to converge to local optima not global optima, scalable RLs with the policy as a NN could be employed for a quicker, yet more optimal tradeoff space exploration. Moreover, while BOA finds near optimal configurations, but in

the presence of numerous frameworks, an immense tradeoff space is inevitable. Therefore, cascading BOA with a RL agent seems promising as BOA finds near optimal configurations, and RL finds expands to the global optima.

In BOA evaluation, we observed that most approximation frameworks are not completely independent and configurations combine in non-linear and unpredictable fashion. Identifying the connection between approximate frameworks and forming theoretical error bounds on resource usage could drastically reduces the tradeoff space size as many conflicting configurations would be omitted.

COBRA only evaluates the on-chip data cache. Approximable data storage components such as data cache, register file, functional unit, load-store queue), data movement strategies (memory accesses skipping [79], load value approximation [73]), and main memory approximations (saving refresh energy in DRAM [47]) could be integrated with COBRA as well.

In this thesis, the approximation frameworks perform at the compile time. However, all the approximation knobs are exposed at the runtime and can dynamically modified. Therefore, self-aware [11, 60] and runtime [30, 41, 42] systems could benefit from BOA combinations and deliver the guaranteed quality of service.

BIBLIOGRAPHY

[1] T. M. Aamodt and P. Chow. "Compile-Time and Instruction-Set Methods for Improving Floating- to Fixed-Point Conversion Accuracy". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008).

[2] A. Alaghi et al. "The Promise and Challenge of Stochastic Computing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1515–1531.

[3] A. Alaghi and J. P. Hayes. "Fast and Accurate Computation Using Stochastic Circuits". In: *Proceedings of the Conference on Design, Automation and Test in Europe.* DATE 14. Dresden, Germany: European Design and Automation Association, 2014.

[4] J. Ansel et al. "Language and Compiler Support for Auto-tuning Variable-accuracy Algorithms". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 85–96.

[5] J. Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation.* PACT '14. Edmonton, AB, Canada: ACM, 2014, pp. 303–316.

[6] A. Aponte-Moreno et al. "A review of approximate computing techniques towards fault mitigation in HW/SW systems". In: *2018 IEEE 19th Latin-American Test Symposium (LATS).* 2018, pp. 1–6.

[7] G. Ascia et al. "A GA-based design space exploration framework for parameterized system-on-a-chip platforms". In: *IEEE Transactions on Evolutionary Computation* 8.4 (2004), pp. 329–346.

[8] M. Ashouei et al. "A voltage-scalable biomedical signal processor running ECG using 13pJ/cycle at 1MHz and 0.4V". In: *2011 IEEE International Solid-State Circuits Conference.* 2011, pp. 332–334.

[9] W. Baek and T. M. Chilimbi. "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation". In: *SIGPLAN Not.* 45.6 (June 2010), pp. 198–209.

[10] J. Balkind et al. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '16. ACM, 2016, pp. 217–232.

[11] S. Barati et al. "Proteus: Language and Runtime Support for Self-Adaptive Software Development". In: *IEEE Software* 36.2 (2019), pp. 73–82.

[12] L. A. Barroso and U. Hlzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007), pp. 33–37.

[13] C. Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, 2011.

[14] C. Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, 2011.

[15] S. Borkar. *The Exascale Challange.* Keynote Talk, Parallel Architectures and Compilation Techniques (PACT), Galveston Island, Texas, USA. Oct. 2011.

[16] D. Bortolotti et al. “Approximate Compressed Sensing: Ultra-Low Power Biosignal Processing via Aggressive Voltage Scaling on a Hybrid Memory Multi-Core Processor”. In: *Proceedings of the 2014 International Symposium on Low Power Electronics and Design.* ISLPED 14. La Jolla, California, USA: Association for Computing Machinery, 2014, 4550.

[17] M. Carbin et al. “Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications.* OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 33–52.

[18] S. Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on.* 2009, pp. 44–54.

[19] W.-F. Chiang et al. “Rigorous Floating-point Mixed-precision Tuning”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* POPL 2017. New York, NY, USA: ACM, 2017, pp. 300–315.

[20] V. K. Chippa et al. “Approximate computing: An integrated hardware approach”. In: *2013 Asilomar Conference on Signals, Systems and Computers.* 2013, pp. 111–117.

[21] M. Courbariaux et al. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: *CoRR* abs/1511.00363 (2015). arXiv: `1511.00363`.

[22] D. Das et al. “Mixed precision training of convolutional neural networks using integer operations”. In: *arXiv preprint arXiv:1802.00930* (2018).

[23] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[24] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[25] K. Du et al. “High performance reliable variable latency carry select addition”. In: *2012 Design, Automation Test in Europe Conference Exhibition (DATE).* 2012, pp. 1257–1262.

[26] S. Eldridge et al. “Neural network-based accelerators for transcendental function approximation”. In: *Proceedings of the 24th edition of the great lakes symposium on VLSI.* ACM. 2014, pp. 169–174.

[27] H. Esmaeilzadeh et al. “Neural Acceleration for General-Purpose Approximate Programs”. In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.* 2012, pp. 449–460.

[28] H. Esmaeilzadeh et al. “Architecture Support for Disciplined Approximate Programming”. In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 301–312.

[29] H. Esmaeilzadeh et al. “Neural Acceleration for General-Purpose Approximate Programs”. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 449–460.

[30] A. Farrell and H. Hoffmann. “MEANTIME: Achieving Both Minimal Energy and Timeliness with Approximate Computing”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 421–435.

[31] L. Fousse et al. “MPFR: A multiple-precision binary floating-point library with correct rounding”. In: *ACM Transactions on Mathematical Software (TOMS)* 33.2 (2007), p. 13.

[32] N. Gajjar et al. “Scalable LEON 3 based SoC for multiple floating point operations”. In: *2011 Nirma University International Conference on Engineering*. 2011, pp. 1–3.

[33] S. Ganapathy et al. “Mitigating the impact of faults in unreliable memories for error-resilient applications”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[34] M. Gao and G. Qu. “A novel approximate computing based security primitive for the Internet of Things”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017, pp. 1–4.

[35] T. Givargis et al. “System-level exploration for Pareto-optimal configurations in parameterized systems-on-a-chip”. In: *Computer Aided Design, 2001. ICCAD 2001. IEEE/ACM International Conference on*. 2001, pp. 25–30.

[36] M. Gottscho et al. “Power / Capacity Scaling: Energy Savings With Simple Fault-Tolerant Caches”. In: *Proceedings of the 51st Annual Design Automation Conference*. DAC ’14. New York, NY, USA: ACM, 2014, 100:1–100:6.

[37] Q. Guo et al. “High-Density Image Storage Using Approximate Memory Cells”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 413–426.

[38] V. Gupta et al. “Low-Power Digital Signal Processing Using Approximate Adders”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.1 (2013), pp. 124–137.

[39] C. Hankendi et al. “Adapt Cap: Coordinating System- and Application-Level Adaptation for Power-Constrained Systems”. In: *IEEE Design Test* 33.1 (2016), pp. 68–76.

[40] N. Ho et al. “Efficient floating point precision tuning for approximate computing”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 63–68.

[41] H. Hoffmann. “CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems”. In: *2014 26th Euromicro Conference on Real-Time Systems*. 2014, pp. 223–232.

[42] H. Hoffmann. “JouleGuard: Energy Guarantees for Approximate Applications”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP 15. Monterey, California: Association for Computing Machinery, 2015, 198214.

[43] H. Hoffmann et al. “Dynamic Knobs for Responsive Power-aware Computing”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 199–212.

[44] M. Holzer et al. “Design Space Exploration with Evolutionary Multi-Objective Optimisation”. In: *2007 International Symposium on Industrial Embedded Systems*. 2007, pp. 126–133.

[45] F. Johansson et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.14)*. `http://code.google.com/p/mpmath/`. 2010.

[46] N. P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), 112.

[47] M. Jung et al. “Invited: Approximate computing with partially unreliable dynamic random access memory Approximate DRAM”. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–4.

[48] D. Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: `1905.12322 [cs.LG]`.

[49] A. Kanduri et al. “Approximation knob: Power Capping meets energy efficiency”. In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–8.

[50] Khaing Yin Kyaw et al. “Low-power high-speed multiplier for error-tolerant application”. In: *2010 IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*. 2010, pp. 1–4.

[51] J. Knowles and D. Corne. “The Pareto archived evolution strategy: a new baseline algorithm for Pareto multiobjective optimisation”. In: *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*. Vol. 1. 1999, 105 Vol. 1.

[52] M. de Kruijf et al. “Relax: An Architectural Framework for Software Recovery of Hardware Faults”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. Saint-Malo, France: ACM, 2010, pp. 497–508.

[53] P. Kulkarni et al. “Trading Accuracy for Power with an Underdesigned Multiplier Architecture”. In: *2011 24th Internatioal Conference on VLSI Design*. 2011, pp. 346–351.

[54] T.-J. Kwon and J. Draper. “Floating-point division and square root using a Taylor-series expansion algorithm”. In: *Microelectronics Journal* 40.11 (2009). International Conference on MicroelectronicsDigital and Mixed-Signal Circuits and Systems, pp. 1601 –1605.

[55] J. Lebak et al. “Parallel VSIPL++: An Open Standard Software Library for High-Performance Parallel Signal Processing”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 313–330.

[56] Y. LeCun et al. “Object recognition with gradient-based learning”. In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[57] S. Li et al. “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2009, pp. 469–480.

[58] A. Lingamneni et al. “Synthesizing Parsimonious Inexact Circuits Through Probabilistic Design Techniques”. In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013), 93:1–93:26.

[59] C.-K. Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: ACM, 2005, pp. 190–200.

[60] M. Maggio et al. “Automated Control of Multiple Software Goals Using Multiple Actuators”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 373384.

[61] K. T. Malladi et al. “Towards energy-proportional datacenter memory with mobile DRAM”. In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 37–48.

[62] L. Marti et al. “An approach to stopping criteria for multi-objective optimization evolutionary algorithms: The MGBM criterion”. In: *2009 IEEE Congress on Evolutionary Computation*. 2009, pp. 1263–1270.

[63] L. Martí et al. “A Cumulative Evidential Stopping Criterion for Multiobjective Optimization Evolutionary Algorithms”. In: *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO ’07. New York, NY, USA: ACM, 2007, pp. 2835–2842.

[64] M. McKeown et al. “Power and Energy Characterization of an Open Source 25-Core Manycore Processor”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 762–775.

[65] N. Mellempudi et al. *Mixed Precision Training With 8-bit Floating Point*. 2019. arXiv: `1905.12334 [cs.LG]`.

[66] J. S. Miguel et al. “Load Value Approximation”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 127–139.

[67] S. Mittal. “A Survey of Techniques for Approximate Computing”. In: *ACM Comput. Surv.* 48.4 (Mar. 2016).

[68] T. Moreau et al. “Approximate Computing: Making Mobile Systems More Efficient”. In: *IEEE Pervasive Computing* 14.2 (2015), pp. 9–13.

[69] T. Moreau et al. “SNNAP: Approximate computing on programmable SoCs via neural acceleration”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 603–614.

[70] M. Ortiz et al. *Low-Precision Floating-Point Schemes for Neural Network Training.* 2018. arXiv: 1804.05267 [cs.LG].

[71] G. Palermo et al. “ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.12 (2009), pp. 1816–1829.

[72] J. M. Rabaey. *Digital integrated circuits: a design perspective.* Vol. 7. 2003.

[73] A. Rahimi et al. “Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 60.12 (2013), pp. 847–851.

[74] A. Ranjan et al. “Approximate storage for energy efficient spintronic memories”. In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[75] B. Reagen et al. “Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 267–278.

[76] M. Rinard et al. “Patterns and Statistical Analysis for Understanding Reduced Resource Computing”. In: *SIGPLAN Not.* 45.10 (Oct. 2010), 806821.

[77] P. Roy et al. “ASAC: Automatic Sensitivity Analysis for Approximate Computing”. In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES 14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, 95104.

[78] C. Rubio-Gonzlez et al. “Precimonious: Tuning assistant for floating-point precision”. In: *SC ’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12.

[79] M. Samadi and S. Mahlke. “CPU-GPU collaboration for output quality monitoring”. In:

[80] A. Sampson et al. “EnerJ: Approximate Data Types for Safe and General Low-power Computation”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 164–174.

[81] A. Sampson et al. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 164–174.

[82] A. Sampson et al. "Approximate Storage in Solid-state Memories". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 25–36.

[83] M. Shafique et al. "A low latency generic accuracy configurable adder". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.

[84] M. Shafique et al. "Invited - Cross-Layer Approximate Computing: From Logic to Architectures". In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC 16. Austin, Texas: Association for Computing Machinery, 2016.

[85] Q. Shi et al. "A Cross-Layer Multicore Architecture to Tradeoff Program Accuracy and Resilience Overheads". In: *IEEE Computer Architecture Letters* 14.2 (2015), pp. 85–89.

[86] M. Shoushtari et al. "Exploiting Partially-Forgetful Memories for Approximate Computing". In: *IEEE Embedded Systems Letters* 7.1 (2015), pp. 19–22.

[87] S. Sidiroglou-Douskos et al. "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*. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 124–134.

[88] N. Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958.

[89] P. Stanley-Marbell et al. "Exploiting Errors for Efficiency: A Survey from Circuits to Algorithms". In: *arXiv preprint arXiv:1809.05859* (2018).

[90] B. Thwaites et al. "Rollback-free value prediction with approximate loads". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 493–494.

[91] J. Y. F. Tong et al. "Reducing power by optimizing the necessary precision/range of floating-point arithmetic". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.3 (2000), pp. 273–286.

[92] V. Vassiliadis et al. "A Programming Model and Runtime System for Significance-aware Energy-efficient Computing". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA: ACM, 2015, pp. 275–276.

[93] S. Venkataramani et al. "SALSA: Systematic logic synthesis of approximate circuits". In: *DAC Design Automation Conference 2012*. 2012, pp. 796–801.

[94] S. Venkataramani et al. "AxNN: Energy-efficient neuromorphic systems using approximate computing". In: *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2014, pp. 27–32.

[95] S. Venkataramani et al. “Quality Programmable Vector Processors for Approximate Computing”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 1–12.

[96] S. Venkataramani et al. “Quality Programmable Vector Processors for Approximate Computing”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 1–12.

[97] A. K. Verma et al. “Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design”. In: *2008 Design, Automation and Test in Europe*. 2008, pp. 1250–1255.

[98] A. K. Verma et al. “Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design”. In: *2008 Design, Automation and Test in Europe*. 2008, pp. 1250–1255.

[99] S. Wu et al. “Training and inference with integers in deep neural networks”. In: *arXiv preprint arXiv:1802.04680* (2018).

[100] P. Yang and F. Catthoor. “Pareto-optimization-based run-time task scheduling for embedded systems”. In: *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*. 2003, pp. 120–125.

[101] Y.-H. Yang et al. *Language Support for Adaptation: Intent-Driven Programming in FAST*. 2019. arXiv: `1907.08695 [cs.PL]`.

[102] A. Yazdanbakhsh et al. “Axilog: Language support for approximate hardware design”. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 812–817.

[103] A. Yazdanbakhsh et al. “Neural Acceleration for GPU Throughput Processors”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 482–493.

[104] Y. Yetim et al. “Extracting useful computation from error-prone processors for streaming applications”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2013, pp. 202–207.

[105] G. Zervakis et al. “Design-Efficient Approximate Multiplication Circuits Through Partial Product Perforation”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.10 (2016), pp. 3105–3117.

[106] H. Zhang et al. “Low Power GPGPU Computation with Imprecise Hardware”. In: *Proceedings of the 51st Annual Design Automation Conference*. DAC ’14. San Francisco, CA, USA: ACM, 2014, 99:1–99:6.

[107] A. Zhou et al. “Multiobjective evolutionary algorithms: A survey of the state of the art”. In: *Swarm and Evolutionary Computation* 1.1 (2011), pp. 32 –49.

[108] N. Zhu et al. “Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.8 (2010), pp. 1225–1229.

[109] E. Zitzler et al. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Tech. rep. 2001.