The Dissertation Committee for Xin Sui
certifies that this is the approved version of the following dissertation:

Principled Control of Approximate Programs

Committee:

Keshav Pingali, Supervisor

Derek Chiou

Inderjit Dhillon

Donald S. Fussell

Vijaya Ramachandran

Principled Control of Approximate Programs

by

Xin Sui, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

Decemeber 2015

Dedicated to my father Yongjie Sui and my mother Xiuyan Dong.

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor Keshav Pingali for his invaluable guidance and support over the past years. Without his encouragement and help, this thesis would not have been possible. From Keshav, I learned not only research skills in one research area but also the way to think difficult problems and the way to present things clearly, which will benefit my entire life. I am really thankful to his great patience to listen to my talks many times and help me improve them.

I would like to thank all the members of my PhD committee. Inderjit Dhillon and Donald Fussell have devoted a lot of their time to solving problems and giving advice on the projects on which we collaborated. Inderjit, Donald, Vijaya Ramachandran and Derek Chiou have given very helpful feedback and spent a lot of time.

Martin Burtscher has been a great mentor for me during the early stage of my PhD study. Andrew Lenharth inspired many research ideas and provided constructive comments on technical details.

I am really grateful to Dimitrios Prountzos and Joyce Jiyoung Whang for being friends and colleagues. Dimitrios gave me a lot of helps during my PhD study and going lunch with him every weekday has been great memories in my life. Working with Joyce was very enjoyable.

I would like to thank the past and current members of Intelligent Software System research group for their technical and personal support: Noah Anderson, Roshan Dathathri, Gurbinder Gill, Amber Hassaan, Jiayuan He, Rashid Kaleem, Milind Kulkarni, Yi-Shan Lu, Mario Mendez-Lojo, Rupesh Nasre, Donald Nguyen, and Sreepathi Pai.

I would also like to acknowledge my friends and colleagues for all memorable moments in Austin: Meiru Che, Song Han, Byeongchol Lee, Dong Li, Na Meng, Bin Wang, Wei Tang, Guowei Yang, Xiuming Zhu, Zifei Zhong.

Lastly and most importantly I would like to thank my parents for their unconditional love and selfless support throughout my life.

Principled Control of Approximate Programs

Publication No. ________________

Xin Sui, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Keshav Pingali

In conventional computing, most programs are treated as implementations of mathematical functions for which there is an exact output that must computed from a given input. However, in many problem domains, it is sufficient to produce some approximation of this output. For example, when rendering a scene in graphics, it is acceptable to take computational shortcuts if human beings cannot tell the difference in the rendered scene. In other problem domains like machine learning, programs are often implementations of heuristic approaches to solving problems and therefore already compute approximate solutions to the original problem.

This is the key insight for the new research area, *approximate computing*, which attempts to trade-off such approximations against the cost of computational resources such as program execution time, energy consumption, and memory usage. We believe that approximate computing is an important step towards a more fundamental and comprehensive goal that we call *information-efficiency.* Current applications compute more information (bits) than are needed to produce their outputs, and since producing and

transporting bits of information inside a computer requires energy/computation time/memory usage, information-inefficient computing leads directly to resources inefficiency.

Although there is now a fairly large literature on approximate computing, system researchers have focused mostly on what we can call the *forward problem*; that is, they have explored different ways in both hardware and software to introduce approximations in a program and have demonstrated that these approximations can enable significant execution speedups and energy savings with some quality degradation of the result. However, these efforts do not provide any guarantee on the amount of the quality degradation. Since the acceptable amount of degradation usually depends on the scenario in which the application is deployed, it is very important to be able to control the degree of approximation. In this dissertation, we refer to this problem as the *inverse problem*. Relatively little is known about how to solve the inverse problem in a disciplined way.

This dissertation makes two contributions towards solving the inverse problem. First, we investigate a large set of approximate algorithms from a variety of domains in order to understand how approximation is used in real-world applications. From this investigation, we determine that many approximate programs are *tunable* approximate programs. Tunable approximate programs have one or more parameters called knobs that can be changed to vary the quality of the output of the approximate computation as well as the corresponding cost. For example, an iterative linear equation solver can vary the number of iterations to trade quality of the solution versus the execution time, a Monte Carlo path tracer can change the number of sampling light paths to trade the quality of the resulting image against execution time, etc. Tunable

approximate programs provide many opportunities for trading accuracy versus cost. By carefully analyzing these algorithms, we have found a set of patterns for how approximation is applied in tunable programs. Our classification can be used to identify new approximation opportunities in programs.

A second contribution of this dissertation is an approach to solving the inverse problem for tunable approximate programs. Concretely, the problem is to determine knob settings to minimize the cost while keeping the quality degradation within a given bound. There are four challenges: i) for real-world applications, the quality and cost are usually complex non-linear functions of the knobs and these functions are usually hard to express analytically; ii) the quality and the cost for an application vary greatly for different inputs; iii) when an acceptable quality degradation bound is presented, determining the knob setting has to be very efficient so that the extra overhead incurred by the identification will not exceed the cost saved by the approximation; and iv) the approach should be general so that it can be applied to many applications.

To meet these requirements, we formulate the inverse problem as a constrained optimization problem and solve it using a machine learning based approach. We build a system which uses machine learning techniques to learn cost and quality models for the program by profiling the program with a set of representative inputs. Then, when a quality degradation bound is presented, the system searches these error and cost models to identify the knob settings which can achieve the best cost savings while simultaneously guaranteeing the quality degradation bound statistically. We evaluate the system with a set of real world applications, including a social network graph partitioner, an image search engine, a 2-D graph layout engine, a 3-D game physics engine, a SVM solver and a radar signal processing engine. The experiments showed great savings in execution time and energy savings for a variety of quality bounds.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Achieving Information-Efficiency Via Approximate Computing

We live in an era in which computation is increasingly constrained by power and energy consumption. In portable devices like smart-phones and tablets, computation is often restricted both by battery capacity and by limits on power dissipation; in fact, for many applications, energy efficiency is more important than performance. At the other end of the computing spectrum, supercomputer and data center designs are being constrained by the cost of energy and by looming regulatory limits [33].

These concerns about power and energy efficiency are relatively recent because until a few years ago, Moore's Law and Dennard scaling reduced the size and energy consumption of digital circuits while simultaneously increasing their speed. As a result, contemporary applications have not been designed for energy efficiency.

One popular approach to improving the energy efficiency of applications is *approximate computing* [28, 61, 63, 70, 73, 76]. The key insight is that when executing a program, energy can be reduced by taking computational short-cuts such as skipping iterations and tasks or reducing the precision of data values, and while this changes the output of a program in general, the resulting output may still be "acceptable" for some applications. For example, when rendering a scene, computational short-cuts to save energy are acceptable as long as the changes to the rendered image do not matter to the person viewing the image.

We believe that approximate computing is an important step towards a more fundamental and comprehensive goal that we call *information-efficiency*. Current applications compute more information (bits) than is needed to produce their outputs, and since producing and transporting bits of information inside a computer requires energy, *information-inefficient computing leads directly to energy inefficiency*.

Besides approximate computing, there are other ways to achieve information efficient computing. For example, a low-precision ray-tracing accelerator for GPUs is proposed by [41, 42]. In this system, a conventional GPU is augmented with an accelerator that performs a very low-precision spatial search for tracing rays in which inaccuracies resulting from the lowered precision are corrected later in the computation and prevented from affecting the output. This information-efficient design improves performance but reduces the area and power spent in ray traversal while producing *exactly the same output* as the standard full-precision search.

Approximate computing achieves the area/energy/performance gain by reducing the information produced at the output, while the other approach achieves this by eliminating the computation and movement of intermediate information that was not needed to produce the output.

In this thesis, we focus on approximate computing as the way to achieve information efficiency. In particular, we focus on a class of approximate programs that we call *tunable* approximate programs. Intuitively, these programs have one or more *knobs* or parameters that can be changed to vary the fidelity of the produced output. Not all approximate algorithms are tunable. For example, Kempe's heuristic for graph coloring [24] works for most graphs that arise in register allocation, but if it fails for a particular graph, there is no way to tune it to produce a coloring even if one exists.

One of the oldest examples of a tunable approximate algorithm is the method used by Archimedes to estimate the value of π by constructing inscribed and circumscribed regular polygons for circles; the number of sides of the polygons is the knob, and more accurate estimates for π can be obtained by "dialing up" this knob. Iterative algorithms for solving linear or non-linear systems control the number of iterations to trade off solution accuracy for computational time and energy. Knobs might also control the number of iterations performed by a loop [12, 61], determine the precision with which floating-point computations are performed [63, 70], or switch between precise and approximate hardware [28].

Most of the existing literature in approximate computing focuses on solving the *the Forward Problem*: introducing hardware or software knobs into programs and studying what happens to the output of a tunable approximate program when its knobs are dialed up or down. For example, [68], [50], and [27] explore introducing approximations into computer hardware such as arithmetic operations, registers and memory; and [28] replaces code segments with hardware-implemented neural networks which produces approximate output generated by the code segments. In the context of software approaches, loop perforation [73] explores skipping iterations during loop execution; [60] explores randomly discarding tasks in parallel applications; and [61] and [15] explores relaxing synchronization in parallel applications. Those studies have demonstrated experimentally that in some programs, approximations can enable significant execution speedup and energy savings with little quality or fidelity degradation of the result.

While these studies of the forward problem are useful, optimizing energy, rather than merely reducing it, requires the solution to what we called the *Inverse Problem* for tunable approximate programs: roughly speaking, given a permissible error for the output, we want to set the knobs to minimize the cost of computation, such as running time or energy, while meeting the error constraint. This problem can be seen as the *Control Problem* for tunable programs. Few studies have attempted to solve this problem. One example is the Green system [6] but it is targeted for streaming applications in which the estimated error for one input is used to control error for the next input. This kind of *reactive* control is not useful for applications like the ones considered in this thesis that require *proactive* control since they are given a single input value.

1.2 Thesis Statement

The dissertation has the following two objectives.

I. Understand the space of tunable approximate algorithms, and identify opportunities for control knobs in these applications.

A comprehensive understanding of tunable approximate algorithms will help identify appropriate knobs and how they can be controlled.

To meet this objective, we survey a variety of tunable approximate algorithms across different application domains such as graphics, machine learning, database and internet, and classify these algorithms into a small number of categories. The classification provides a systematic understanding of approximate algorithms.

II. Solve the *Control Problem* for tunable approximate programs.

To understand how to achieve this objective, consider an example, the GEM algorithm [85] for clustering social networks. Given a social network graph and the number of clusters, GEM produces an assignment of nodes to clusters. The quality of the clustering is estimated using a metric called the normalized cut, defined in more detail in Chapter 2, but roughly speaking, it computes the ratio of inter-cluster edges to intra-cluster edges (lower is better since it means there are fewer edges connecting nodes in different clusters). The GEM program has two components: the first one extracts a representative skeleton of the original graph by picking high degree nodes and clusters this skeleton graph using a weighted kernel k-means algorithm, and the second one projects this clustering to the original graph and uses weighted kernel k-means again to refine this clustering. Weighted kernel k-means is an iterative algorithm, so there are two knobs in GEM, one for each component. For a given input graph, running both components to convergence produces a clustering of some quality; reducing the number of iterations in either component may reduce the computational cost but may impact the quality of the clustering. Given an input graph and some desired quality of the output, the *Control Problem* is the following: how do we set the knobs for the two components optimally to minimize computational time or energy, given some desired fidelity of the output?

To solve the control problem for programs such as GEM, there are several design goals.

- We would like to find an approach which can solve *Control Problem* for different tunable approximate programs. A control approach targeting a particular program is not our goal.

- Tunable approximate programs such as GEM are given a single input, not a stream of inputs. Therefore, reactive control approaches such as the approach used in Green system [6] cannot be applied. In addition, reactive approaches require a cheap way to compute the quality of an output in order to generate the feedback for the control system, but the quality of the outputs of many tunable approximate programs cannot be evaluated without computing the exact output. Therefore a proactive approach is needed for general tunable approximate programs.

- Solving the *Control Problem* requires knowing the relationships between quality, cost and knobs. These relationships are usually very complex. Therefore, building an analytic model for the interactions between the quality and the knobs for tunable approximate programs such as GEM is very hard even for people with expert mathematical knowledge, if not impossible. Building an analytic model for the interactions between knobs and cost requires a deep understanding of the computer architecture. It is hard to find experts in both areas. In addition, the quality of the output or the cost usually depends on inputs, so analytic models are usually too conservative to be useful.

- Solving the *Control Problem* has to be efficient. Identifying optimal knob settings should not exceed the savings by executing the approximate version of the programs.

To meet the above design goals, we formulate the control problem as an optimization problem, justifying it by describing other reasonable formulations and explaining why we do not use them. We solve the optimization problem by using a machine learning based approach to build error/quality and cost models which model the interactions among quality, cost and knobs. Machine learning based approaches eliminate the requirement of analytic modeling skills. It is a general approach for any kind of tunable approximate programs. We validate the approach using a set of complex applications.

1.3 Contributions

The dissertation makes the following contributions.

- The dissertation analyzes approximation patterns in a variety of tunable approximate algorithms and proposes a classification for approximate programs. The classification helps the understanding of approximation patterns in approximate algorithms.
- The dissertation shows that the error and cost behaviors of approximate programs are very complex. The error and cost functions of the approximate programs are usually non-linear and vary greatly across inputs.
- The dissertation proposes a proactive approach to solve the control problem. It formulates the control problem as a constrained optimization problem, dealing explicitly with the problem of input variability. This formulation provides a systematic way to trade-off accuracy and cost for tunable approximate programs.

- The dissertation describes a machine learning based system to solve the formulated control problem. The machine learning based approach allows the system to control a variety of approximate programs without the requirement of expert analytic modeling skills from programmers. The approach uses machine learning approaches to build error and cost models. Then a control algorithm is used to find the optimal knob settings by querying the models. The proposed approach is validated using six complex approximate programs. The dissertation shows the proposed approach can not only tune the approximate programs using the compute-time as the cost metric, but can also successfully tune programs to optimize energy. We believe the proposed approach can be applied to other cost metrics.

- The dissertation shows the problem formulation can be easily extended to include multi-criteria error constraints. For example, it allows bounding the maximal error besides the average error.

- The dissertation describes how to scale the proposed approach to a large number of knobs.

1.4 Dissertation Organization

The dissertation is organized as follows:

Chapter 2 surveys a variety of approximate algorithms. By analyzing the patterns in the approximate algorithms, a classification scheme is proposed.

Chapter 3 describes a set of approximate algorithms in detail and the characteristics of their error and cost functions are presented.

Chapter 4 proposes three different ways to model the control problem systematically. Their advantages and disadvantages are discussed.

Chapter 5 proposes a machine learning based system to solve the control problem. The system is validated using a set of real-world complex applications.

Chapter 6 extends the system proposed in Chapter 4 to include multi-criteria error constraints and discusses how to scale the system to applications with larger number of knobs.

Chapter 7 surveys the related research work in approximate computing area.

Chapter 8 concludes the dissertation and discusses future directions for extending the work.

Chapter 2

Tunable Approximate Programs

This chapter addresses the question of where approximate programs come from. The literature in this area has focused mainly on creating approximate versions of arbitrary programs by making *ad hoc* code transformations, such as dropping iterations or tasks [60, 73], in application code that was not necessarily designed for exploiting approximation. The main problem with this approach is that there are no correctness guarantees: the modified program can crash on some inputs or provide arbitrary output.

We believe a more promising approach to producing tunable approximate programs is to recognize that tunable approximate *algorithms* arise naturally in a large number of problem domains. For example, approximations are used to solve NP-hard problems(Polynomial time approximation schemes (PTAS) [36] are used for solving NP-hard problems) or undecidable problems such as program optimization; approximations are used to simulate the physical world; approximations are used to discover rules from data. In this chapter, we argue that these applications can be written so as to expose a set of knobs that can be given to a control system that sets knob values for a desired level of output fidelity in a disciplined way. We also propose a classification scheme to understand the different approximation techniques in such algorithms.

2.1 Classification of Approximate Programs

Table 2.1 lists a number of tunable approximate programs from a variety of domains such as social networks, machine learning, graphics and image processing. By analyzing the approximation patterns in these approximate algorithms, we came up with a classification shown in Figure 2.1. In the following sections, each class in the classification is described in detail.

For the most part, these programs take a set X as input and compute a value $f(X)$. To compute the value of $f(X)$ approximately, we can perform the computation with function that approximates f, compute f precisely with an input that approximates X, or we can use approximations to both f and X. We call the former *computation approximation* and the latter, *data approximation*. Data and computation approximations may or may not be tunable. Tunable approximations have one or more parameters called *knobs* that can be changed to vary the fidelity of the output produced by the programs and execution costs such as running time or energy consumption are changed correspondingly.

2.1.1 Data Approximation

We classify data approximation into subset, superset and quotient approximation.

2.1.1.1 Subset approximation

In subset approximation, a function over a set is approximated by computing that function over a sample of elements from that set. The size of the subset is the knob for tuning the fidelity of result as well as for changing the cost of function evaluation.

	Name	Area
1	Multi-Level of Details based Collision Detection [56]	Graphics
2	Probability based Collision Detection [43]	Graphics
3	Monte Carlo Collision Detection [59]	Graphics
4	Smallpt: Path Tracing [9]	Graphics
5	MPEG2 Encoding [40]	Video Processing
6	ApproxQuickSort [74]	Graphics
7	FlatRendering [30]	Graphics
8	Hierarchical Rendering [47]	Graphics
9	KmeansClustering [79]	Data Mining
10	VLSISteinerTree [23]	VLSI
11	AbstractionRefinementPointToAnalysis [45]	Programming Language
12	ApproxStringJoin [35]	Database
13	Aqua: approximate query system [2]	Database
14	Image Compression [82]	Image Processing
15	Cascade Classifier [20]	Machine Learning
16	Iterative Linear Solver [64]	Numerical Analysis
17	Multigrid Solver [83]	Numerical Analysis
18	BarnesHut N-Body Simulation [8]	Physics
19	FPGA Placement and Routing [53]	FPGA
20	Survey Propagation [13]	SAT Solver
21	Relaxing Max-SAT [22]	SAT Solver
22	Mini-Buckets [25]	Machine Learning
23	Loopy Belief Propagation [54]	Machine Learning
24	Block Pagerank [10]	Internet
25	Monte-Carlo Pagerank [5]	Internet
26	SGDSVM: SVM Solver [12]	Machine Learning
27	GEM: Graph Clustering [85]	Data Mining
28	Ferret:Image Search Engine [11]	Internet
29	Radar Processing [37]	Signal Processing
30	OpenOrd:Graph Layout Engine [49]	Visualization
31	Dot 2D Graph Layout Engine [31]	Visualization
32	Clustered Low-rank Approximation [77]	Data Mining
33	Approximate Kernel Kmeans [21]	Data Mining
34	2D Multi-stoke Gesture recognition [81]	Mobile

Table 2.1: List of Approximate Algorithms

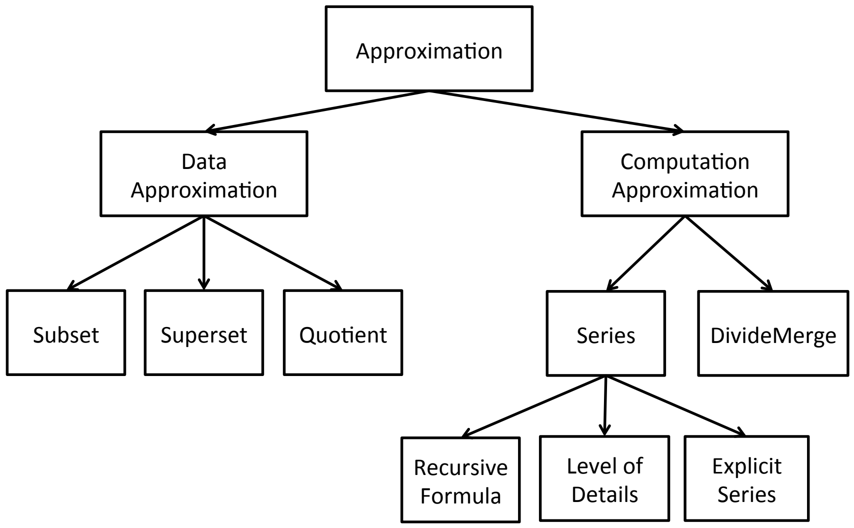


Figure 2.1: Classification of Approximate Programs

Monte Carlo methods are the best examples of this style of data approximation. One of the simplest Monte Carlo algorithms estimates the value of π by sampling points at random within a unit square centered at the origin and finding the fraction of samples that are within a distance of 0.5 from the origin. It is easy to see that this fraction approaches $\pi/4$ as the number of samples increases (this is the area of the circle of radius 1/2 centered at the origin). In this application, we can provide a knob that controls the number of samples. Sampling more points improves the accuracy of the estimate but requires more computation. Other examples are Monte Carlo path tracing, Monte Carlo collision detection [59] and Monte Carlo Page-Rank [5].

Numerical integration techniques can be viewed as performing subset approximation. Consider the computation of the integral $\int_a^b g(x)dx$, shown in Figure 2.2. The value of this integral is the sum of the values of $g(x)$ for all values of x in the interval $[a, b]$. Since this interval contains an infinite

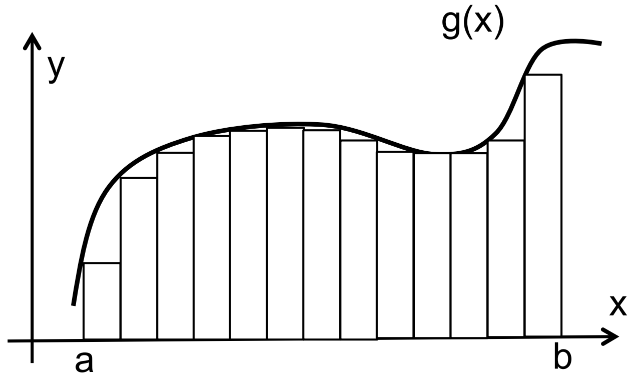


Figure 2.2: Subset Example: Numerical Integration

number of points, we cannot carry out this procedure literally, so in numerical integration, we select a subset of points in that interval, evaluate g at those points, and estimate the integral from these values; Figure 2.2 shows one estimation method called forward-Euler. The knob for this application is the number of points at which the function is evaluated: if more points are used, the approximation is improved at the cost of additional computation.

The third example is a multi-stroke gesture recognition algorithm [81] used in mobile phones. In this algorithm, users pre-save a set of gestures in the phones and each gesture is represented by a set of points (point cloud) in 2-D space, as shown in Figure 2.3. When the user inputs a gesture, the algorithm tries to match the gesture to one of the gestures saved in the phone. The recognition process is the following: first, the input gesture is transformed to a point cloud and each point is given an index. Then the recognition can be formulated as a minimum weighted bipartite graph matching problem. The point cloud of the input gesture can be regarded as one partite set of graph nodes in the bipartite graph and the point cloud of a gesture saved in the phone as the other partite set. Each point pair in the two point clouds has an edge

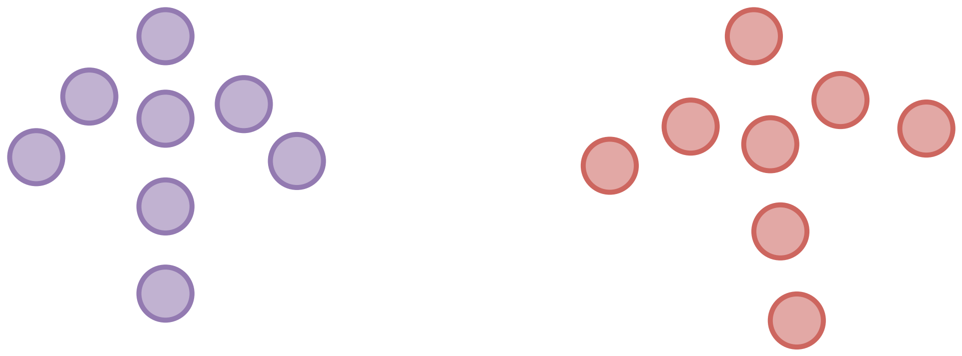


Figure 2.3: Subset Example: Multi-Stoke Gesture Recognition

weighted by the Euclidean distance between them. The gesture saved in the phone with the best matching to the query image is the recognized gesture. Since solving the minimum weighted bipartite graph matching problem has high time complexity, an approximate algorithm based on simple heuristics is used to solve it. The simple heuristic is as follows: the points in each cloud are indexed. For each point(C_i) in a point cloud, the matching algorithm finds the closest point from the other point cloud that has not been matched yet. Once the point C_i is matched, the matching algorithm continues with C_{i+1} until all points from C are matched. The matching algorithm tries different starting points to starts the above process and returns the best matching among those starting points. All of the possible starting points constitute a set and the knob controls the size of the subset of starting points that are actually evaluated. If more starting points are chosen, higher computational cost is incurred but may result in more accurate recognition.

2.1.1.2 Superset approximation

In contrast to subset approximation, superset approximates the data by enlarging the input data to a super set of the input data. Like in subset

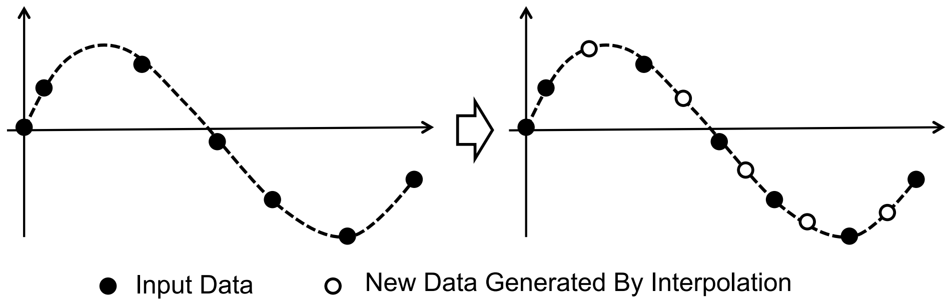


Figure 2.4: Superset Approximation Example: Interpolation

approximation, the size of the set affects the quality and usually affects the cost of the computation.

The example of superset is generating the new data points by interpolation. Interpolation is a technique to estimate the values on the new data points based on known data (input data). Interpolation assumes the data is generated by an underlying function which pass through the known data points. Figure 2.4 shows a one dimensional example. The solid black points represent input data. The dashed curve represents the underlying function computed by the interpolation method. The hollow black points represent the new generated points. The generated super set as the new input for the later phase. Depending on how well the new generated data align with the true data, larger size of the super set may provides better or worse inputs for the later phase and therefore leads to better or worse quality of the final output. Therefore, the quality is usually not monotonic with the size of the super set. Meanwhile, more data points generally lead to more computation in later phase.

One application of interpolation is used in the multi-stroke gesture recognition algorithm described before. When a user draws a gesture using her fingers on the screen of a phone or a tablet, the sensors on the screen will detect a set of points to represent the gesture. The points obtained by the sensors depends on the movement speed of the fingers. To make the recognition independent of the finger moving speed, the gesture recognition algorithm first interpolates the points obtained by the sensors and then generates N number of evenly spaced points from the interpolation for a input gesture. N is a parameter for the algorithm.

2.1.1.3 Quotient approximation

Quotient operations approximate a set by partitioning the elements of the set into equivalence classes and assigning a representative value to each equivalence class. Instead of computing with a given input value, we compute with the representative of its equivalence class. Quotient operations lose information; using more equivalence classes reduces the information loss at the cost of more computation.

Projective techniques in many problem domains usually involve quotient operations. One simple example is the projection of objects from a high dimensional space to a lower dimensional space. If finding intersections of objects in the high dimensional space is expensive, we can compute intersections approximately by projecting the objects into a lower dimensional space and compute intersections of the projections. However, this may result in false positives because it is possible that the projections intersect even when the objects do not; for example, the shadows cast by two people on a wall may intersect even when the two people are not touching each other. Increasing the dimension of the projected space may improve accuracy at the cost of additional computational complexity.

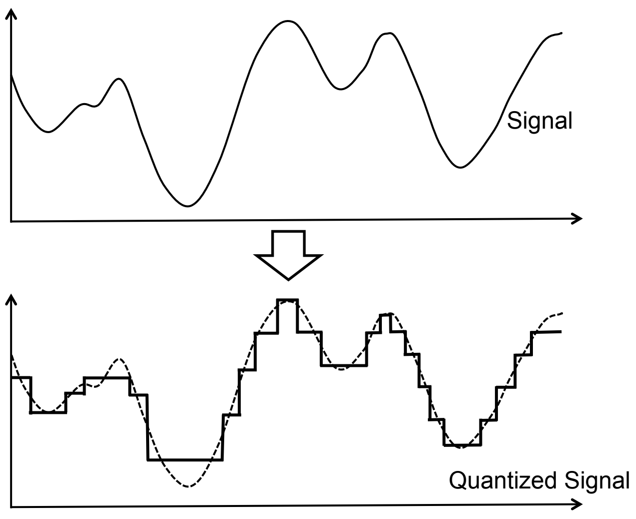


Figure 2.5: Quotient Example: Quantization

Another classical example of quotient approximation is quantization. For example, quantization is used to convert analog signals to digital signals. As shown in Figure 2.5, nearby values of the analog signal in the top sub-figure are converted to the same discrete value in the digital signal in the bottom sub-figure (for example, by rounding the continuous values). Analog signal values that map to the same digital signal value form an equivalence class. Floating-point representations of real numbers are other examples of quotient approximation. In these applications, the number of bits used to represent the approximate values is the knob. Computers support floating-point numbers of different precisions such as 32-bit/64-bit/128-bit. Using more bits in a floating-point computation may produce more accurate results at the cost of additional computation time and energy.

2.1.2 Computation Approximation

Computation approximation estimates the value of $f(x)$ by computing $\tilde{f}(x)$ where $\tilde{f}$ is an approximation of f. This subsection discusses common patterns of *tunable* computation approximations.

2.1.2.1 Series

Series like Taylor series and Maclaurin series are the basis for many computational approximations. For example, under suitable conditions, the Taylor series formula can be used to estimate function values:

$$f(x) = f(a) + \frac{f^{(1)}(a)}{1!}(x-a) + \frac{f^{(2)}(a)}{2!}(x-a)^2 + \frac{f^{(3)}}{3!}(x-a)^3 + ...$$

Summing up more terms of the series will make the approximation more accurate at the price of additional computation.

Many approximate algorithms follow the similar pattern as Taylor series. In such approximate algorithms, the computation can be divided into phases and each phase produces an output which has better quality/lower error than the output produced by the previous phase. The phases compose the series. After any phase, the computation can be stopped and the output of the current phase can be used as the final output. So the stopping criteria is a tuning knob. Depending how the series is defined, we further divided the series class into recursive formula, level of detail and explicit series.

Recursive formula In recursive formula class, the series is defined in a recursive way. Each phase takes the output of the previous phase as its input and perform the same computation as the previous step and produce the output as the input for the next phase. The output produced by later phases is expected to be better than the ones produced by the previous phases (This may not always monotonic for all the algorithms but with more phases, the output is expected to be better than the one with less phases).

One example is the linear equation solver by Jacobi method. The linear equation solver solves a system of linear equations $a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n = b_i, i \in [1, n]$ and output $x_i, i \in [1, n]$. Jacobi method based linear solver initialize x_i randomly and performs the computation in an iterative way. In each iteration, it uses the x_i output by the previous iteration and perform the following computation defined by the following formula: $x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x^k \right), i \in [1, n]$, where k is the kth iteration. The output x_i^{k+1} obtained by the $k + 1$th iteration is a refinement of the one x_i^{k+1} obtained by the previous iteration. Here the iterations construct the series.

Another example is the k-means clustering algorithm [79]. Given a set of points in n-dimensional euclidean space, k-means clustering groups the points into k clusters such that the sum of the distances between each point and the center of the cluster the point belongs to is minimized. The algorithm works as follows: each point is initialized to a cluster randomly, then the cluster each point belongs to is updated in an iterative way. In each iteration, the cluster center is computed by averaging over all the points currently belongs to this cluster; then each point is assigned to the cluster the point is closet to. In this algorithm, each iteration takes the result(the cluster assignment) as the input and output a cluster assignment for the next iteration. So the iterations construct the series.

Level of Details The level of details class constructs the series by first building a set of versions of the input data with different amount of approximation in each version. The different versions of the input data including the original one is organized in a way that the next more accurate version can be accessed from the current version. Then the series can be constructed by running the program on the least accurate version of input data, to running the program on next more accurate version until the original version. The benefit of this is running on the less accurate version of data usually has significant less computation cost than the more accurate version. Depending on the accuracy requirement, the computation can be stopped early without going to the original data. Different from Recursive formula, each phase in the Level of Details does not take the output of previous phase as input, but takes a more accurate version of the input to compute the output from scratch.

One example is the Barnes-Hut algorithm, an approximate algorithm for simulating the gravity interactions of n particles in 2-D or 3-D space. Barnes-Hut algorithm partitions the space recursively into subspace until in each subspace only one particle exists, as shown in Figure 2.6. In each subspace, an approximate version of the particles in this subspace is constructed by replacing all of the particles with one new particle whose position and mass is respectively the center of mass and the total mass of all of the particles in the subspace. The way that the approximate version is built belongs the quotient class in data approximation. All of the subspaces can be organized as a tree, as shown in Figure 2.6. The root of the tree represents the entire space, the internal node represents a subspace and the leaf node represents a particle. Barnes-Hut algorithm computes the force for a particle in the following way: starting from the root, for each children node A, if node A is far from the particle, computes the force between the particle and the approximate version of the particle in the subspace associated with node A; otherwise the algorithm

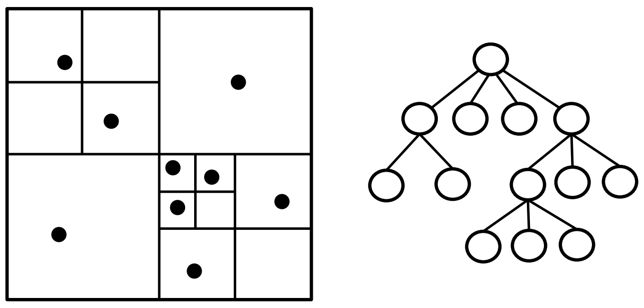


Figure 2.6: Level of Details Example: Barnes-Hut algorithm

will repeat the same process with the children of node A until the leaves are reached. From the algorithm, we can see the series is composed of taking each node from the root to the leaves as input to compute the force for a particle.

Explicit Series In Recursive Formula and Level of Details, the computation in each phase of the series remains the same. In Explicit Series, the computation in each phase is different and is defined explicitly.

One example is Cascade Classifier for optical character recognition (OCR) [20]. The input of the algorithm is a image representing an character and the output is the class the character is classified into. As shown in Figure 2.7, a set of neural network based classifier is combined into a sequence. The neural network at $stage_i$ has less computational cost and also less accuracy than the one in $stage_{i+1}$. Given an image, each classifier can output the classification of this character as well as an error rate which measures the confidence of the classification. When the error rate produced by a classifier C_i is larger than $1 - T_i$, where T_i is the threshold in C_i, it pass this image to its following classifier in the sequence. Otherwise, it absorbs this character and return the classification result. Depending on the threshold setting for each classifier, an input image may pass to more or less classifiers and lead to

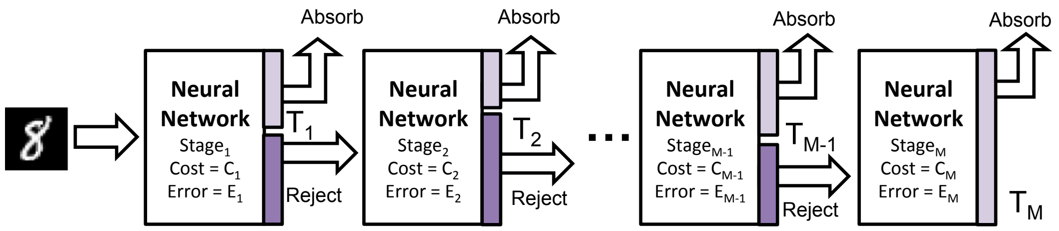


Figure 2.7: Explicit Series Example: Cascade Classifier for Optical Character Recognition (OCR)

more accurate or less accurate output. Here the sequence of classifier forms the series. Different from Recursive formula and Level of Details, each phase in the series is different and is explicitly defined.

2.1.2.2 DivideMerge

Divide and Conquer approach is very efficient if a problem can be divided into sub-problems and each sub-problem can be solved independently. However, many problems cannot be divided into independent sub-problems, due to global constraints. One pattern in approximate programs is that they relax global constraints so that the problem can be divided into independent sub-problems. After solving each sub-problem, the solutions for sub-problems are merged into the final solution. During the merging phase, extra work may be done to compensate the relaxation when dividing the problems.

An example is the block Page-Rank algorithm [10], which is an approximate version of the classic Page-Rank algorithm. The input of the Page-Rank algorithm is the web graph and the output is a Page-Rank value assignment for each node in the graph. The Page-Rank value is a estimation of the importance of the node. The classic Page-Rank algorithm initialize each node with a Page-Rank value, and then repeatedly iterates the nodes in the graph until a stable assignment of the Page-Rank value is obtained. In each iteration, each node propagates its current Page-Rank values to its neighbors and gathers Page-Rank values from its neighbors to update its Page-Rank values. Since the web graph is very large, the classical algorithm takes very long time to finish. The block Page-Rank algorithm [10] approximates the Page-Rank algorithm in the following way. First, the algorithm divides the graph into blocks, for example, by the domain name of the hosts. Second, for each block, the algorithm uses the classic Page-Rank algorithm to compute a Page-Rank value for each node in the block (Local Page-Rank value). This step is dividing the problems into a set of small problems by ignoring the edges between blocks. Third, each block can be treated as a node and all of the blocks form a new graph. The algorithm uses the classic Page-Rank algorithm again on the new graph and computes a Page-Rank value for each block (Block Page-Rank value). Fourth, the final Page-Rank value for each node in the original graph is the local Page-Rank value of the node weighted by the Block Page-Rank value of the block the node belongs to. The third and fourth steps are merging the solutions of sub-problems. The algorithm uses Block Page-Rank value to compensate the relaxation in the first step.

Table 2.2 shows the classification of each algorithm listed in Table 2.1.

2.2 Summary

In this chapter, we surveyed a variety of tunable approximate algorithms in different domains. By analyzing the patterns in these algorithms, we proposed a classification for them. The classification provides a systematic way for understanding tunable approximate algorithms.

	Name	Data Approx	Comput Approx
1	Multi-Level Collision Detection [56]		Level of Details
2	Probability based Collision Detection [43]		Level of Details
3	Monte Carlo Collision Detection [59]	Subset	Level of Details
4	Smallpt: Path Tracing [9]	Subset	
5	MPEG2 Encoding [40]		Explicit Series
6	ApproxQuickSort [74]		Recursive Formula
7	FlatRendering [30]		Explicit Series
8	Hierarchical Rendering [47]		Explicit Series
9	KmeansClustering [79]		Recursive Formula
10	VLSISteinerTree [23]		Explicit Series
11	AbstractionRefinementPointToAnalysis [45]		Level of Details
12	ApproxStringJoin [35]	Superset	
13	Aqua: approximate query system [2]	subset	
14	Image Compression [82]	Quotient	
15	Cascade Classifier [20]		Explicit Series
16	Iterative Linear Solver [64]		Recursive Formula
17	Multigrid Solver [83]		
18	BarnesHut [8]		Level of Details
19	FPGA Placement and Routing [53]		Level of Details, Recursive Formula
20	Survey Propagation [13]		Recursive Formula
21	Relaxing Max-SAT [22]		DivideMerge
22	Mini-Buckets [25]		DivideMerge
23	Loopy Belief Propagation [54]		Recursive Formula
24	Block Pagerank [10]		DivideMerge
25	Monte-Carlo Pagerank [5]	Subset	
26	Clustered Low-rank Approximation [77]		DivideMerge
27	SGDSVM [12]	Subset	Recursive Formula
28	GEM [85]		Recursive Formula
29	Ferret [11]		Explicit Series
30	Radar Processing [37]	Quotient	
31	OpenOrd [49]		Recursive Formula
32	Dot 2D Graph Layout Engine [31]		Recursive Formula
33	Approximate Kernel Kmeans [21]	Quotient	Recursive Formula
34	2D Multi-stoke Gesture recognition [81]	Superset, Subset	

Table 2.2: List of Approximate Algorithms

Chapter 3

Characteristics of Tunable Approximate Programs

In this chapter, we study the characteristics of some of the tunable approximate programs introduced in the previous chapter. In particular, we present experimental results that show that the relationship between knob settings and output fidelity or quality in these programs is very input-dependent and non-linear, making it unlikely that one can find closed-form expressions describing such relationships.

Most of the programs considered in this thesis produce complex objects like graphs or sets as output. The notion of quality for the output of such programs is usually application dependent, so we require the application programmer to provide a *divergence* function that quantifies the difference in quality between a given output and the "reference" output without approximation, for a given input. The reference output can be the output produced by the exact execution, if this is available, or the best execution in the space of knob settings, for that input. Note that smaller divergences are better. The *error* is defined as a normalized version of divergence:

$$\text{Error}(d) = d/d_{max}$$

where d_{max} represent the maximum values of divergence over the space of knob settings for a given input.

In our experiments, we also study how computational cost changes as knob values are changed. This cost can be execution time, energy consumption, memory consumption or the cost of other resources consumed by the execution. In the following sections, we use execution time and energy consumption to exhibit the characteristics of tunable approximate programs.

3.1 Description of Applications

3.1.1 GEM

GEM [85] is a graph clustering algorithm for social networks. Given a social network graph and the number of clusters, GEM produces an assignment of nodes to clusters. The GEM program has two components: the first one extracts a representative skeleton of the original graph by picking high degree nodes and clusters this skeleton graph using a weighted kernel k-means algorithm, and the second one projects this clustering to the original graph and uses weighted kernel k-means again to refine this clustering. Weighted kernel k-means is an iterative algorithm, so there are two knobs in GEM, one for each component. For a given input graph, running both components to convergence produces a clustering of some quality; reducing the number of iterations in either component reduces the computational cost but may impact the quality of the clustering.

Knobs There are two components, and both use a weighted kernel k-means algorithm, and have a knob controlling the number of iterations for that component. Each knobs can be tuned up to 40 levels. All of the graphs are partitioned into 100 clusters in our experiments.

Error Metrics The output of GEM is the cluster assignments of each node in the graph. There is a standard way to measure the quality of graph clustering, using the notion of a *normalized cut*, which is defined as follows:

$$\frac{\sum_{k=1}^{N}\sum_{i=1,i\neq k}^{N} \mathrm{edges}(C_k, C_i)/\mathrm{edges}(C_k)}{N}$$

where N is the number of clusters, edges(C_k, C_i) denotes the number of edges between cluster k and cluster i, and edges(C_k) denotes the edges inside cluster k.

The reference execution is the execution achieving the smallest normalized cut. As shown in Figure 3.1, the divergence function first applies the normalized cut to convert the output of GEM to a numerical value and then compute the divergence by computing the numerical difference between the calculated numerical value and the numerical value of the reference output. The error is a normalized version of the divergence which is the divergence value for an output divided by the maximum divergence value.

3.1.2 Ferret

Ferret [11] performs content-similarity based image search from a image database. Given a query image, Ferret first decomposes it into a set of segments, and for each segment, Ferret finds a set of candidate image matches by indexing its database. After indexing, all of the candidate images are ranked, and the top K images are returned. There are two major computational phases in this process: finding the candidate image set and ranking the candidate images. In the first phase, the candidate images consist of the $2K$ nearest neighbors of the query image in the database. An algorithm called Multi-probe LSH is used to approximately find $2K$ nearest neighbors. This

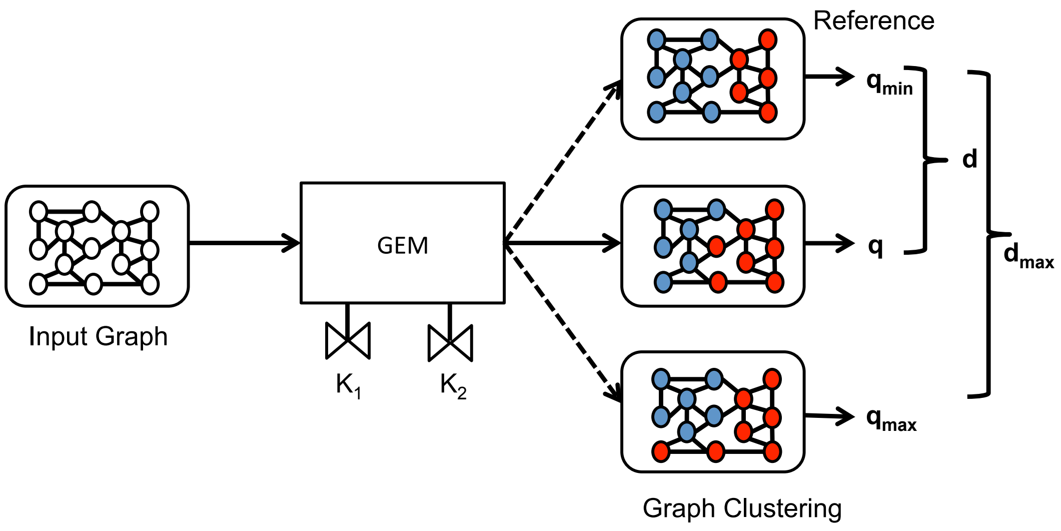


Figure 3.1: Error Metric for GEM

algorithm uses multiple hash tables, each with a different hash function, which map similar images to the same hash bucket with high probability. Besides the mapped hash bucket, Multi-probe LSH probes nearby buckets in each hash table until a specified number of buckets are explored. In the second phase, the Earth Mover's Distance (EMD) between each candidate image and the query image is computed. EMD is a measure of distance between two probability distributions and is computed by an iterative optimization algorithm which calculates the minimum work to transform one probability distribution to the other.

Knobs In the multi-probe LSH, the number of hash tables per bucket and the number of buckets probed can be changed to trade off error for cost. Computing EMD is an iterative optimization algorithm, and the number of iterations trade-off error for cost. The first knob can be tuned to change the number of hash tables to up to 4 levels, the second knob can tune the number of probed buckets to up to 10 levels and the third knob can be tuned to change the number of iterations to up to 25 levels in our experiments.

Error Metrics The output of Ferret is an ordered image list. There is no method to convert the ordered image list to an numerical value to measure the output quality, but there is method to compare two ordered image list. As shown in Figure 3.2, unlike GEM, divergence is directly computed by applying the divergence function on the output of Ferret.

We adopt a metric [7] which is widely used in the comparing search engine results as the divergence function. Given two image lists L_1 and L_2 returned by two executions, the divergence function is:

$$\begin{aligned} 2 \times (k - z)(k + 1) + \sum_{i \in Z} |rank_1(i) - rank_2(i)| \\ - \sum_{i \in S} rank_1(i) - \sum_{i \in T} rank_2(i) \end{aligned} \tag{3.1}$$

where Z is the set of images appears in both L_1 and L_2, S and T are the sets of images only appearing in L_1 and L_2 respectively. k, z are the sizes of L_1(or L_2) and Z respectively. $rank_1(i)$ and $rank_2(i)$ are the rank of the image i in L_1 and L_2.

The reference execution is obtained by turning three knobs to the maximum levels.

3.1.3 ApproxBullet(Bullet)

ApproxBullet(Bullet) is a 3D physics game engine for approximately simulating the rigid body dynamics of objects. The input of ApproxBullet is

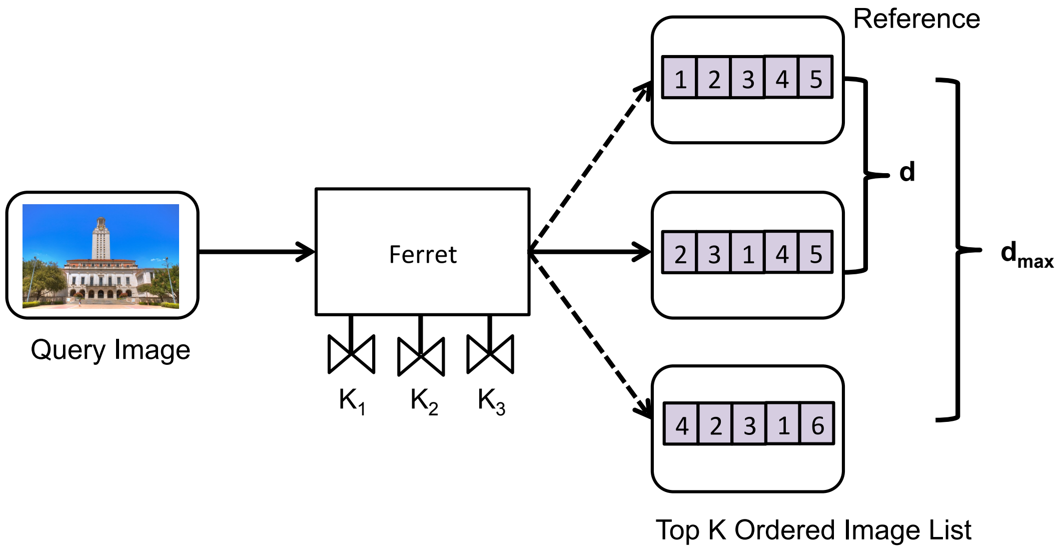


Figure 3.2: Error Metric for Ferret

a set of objects represented by triangular meshes. ApproxBullet simulates the behaviors of the objects in frames. In each frame, ApproxBullet performs two major computations: approximate collision detection and sequential impulse-based constraint solving. The approximate collision algorithm is a simplified version of [56]. It first builds a multi-resolution representation for each object by coarsening its triangular mesh repeatedly [32]. A surface deviation threshold can be specified to control the level of mesh detail used for detecting collisions. The approximate collision detection component outputs a set of collision points to the constraint solver. The collision points are formulated as a set of constraints of a linear complementarity problem (LCP). The LCP problem is solved by an iterative Gauss-Seidel method.

The inputs for this benchmark consist of a set of object pairs represented by triangle meshes. We make one object a stationary object and the other one a moving object. The moving object starts from a high position and drops on the stationary object. We only consider the computation in the frame when the two object first collide. Different inputs are created by enumerating object pairs from a set of objects and rotating them randomly.

Knobs When a coarse mesh is used in the approximate collision detection algorithm, computation will be reduced but the collision detected may be inaccurate. In the sequential impulse-based constraint solver, the number of iterations can trade-off the accuracy of the solution for running time. The knob for the first component is the surface deviation threshold, and the knob for the second component is the number of iterations for the constraint solver. The first knob can be tuned to 15 levels from the original mesh to the coarsest mesh. The second knob can be tuned to set the number of iterations for the Gauss-Seidel solver at 20 different levels.

Error Metrics The output of Bullet for one frame is the "delta change vector" of the speed of the moving object. The divergence function is the Euclidean distance between the two delta speed vectors of the moving object. The reference execution is obtained by turning both knobs to their maximum levels.

3.1.4 SGDSVM(SGD)

Support Vector Machines (SVM) are a supervised machine learning method for binary classification. Classifier training is formulated as an optimization problem that can be solved in many ways. SGD [12] is an approximate SVM solver using an iterative stochastic gradient descent algorithm. The input of SGD is a set of training examples and the output is a model to classify new data. In each iteration of SGD, an approximate gradient is computed to update the model parameters. The process stops after a specified number of iterations.

Knobs The number of training instances and the accuracy of solving the underlying optimization problem can be tuned to trade off higher classification error for faster runtimes. One knob controls the size of the subset, and another knob controls the number of iterations of the stochastic gradient descent algorithm. The first knob can be tuned to randomly sample the training instances at 20 different levels from 5% to 100%. The second knob can be tuned up to 100 levels.

Error Metrics SGD outputs the learned SVM model. The misclassification rate when using the resulting model to classify the test instances is a standard way to measure the quality of the output. The divergence function is computing the difference of the misclassification rates between two models resulted from two executions. The reference execution is the one achieving the minimum misclassification rate for the same input.

3.1.5 OpenOrd

OpenOrd [49] is a graph layout algorithm for drawing graphs in two dimensions. It specializes the force-directed graph layout algorithm to scale to large graphs. The drawing problem is formulated as an optimization problem where nodes connected by high edge weights attempt to move together but nodes attempt to push their nearby nodes far away. The optimization problem is solved by initially positioning all the nodes at the origin and iteratively moving the nodes until their relative positions do not change where the objective is minimized. The iterations are divided into five phases, liquid, expansion, cool-down, crunch and simmer. In each phase, simulated annealing is used to control how far the nodes are allowed to move.

Knobs In practice, the algorithm is not run until convergence. The number of iterations used in each phase affects how far the resulting solution is from the exact solution as well as the amount of time required to compute it. Since there are 5 phases, the application has 5 knobs. In the experiments, the five knobs can be tuned into 3, 6, 6, 3 and 4 levels respectively.

Error Metrics The objective value is used as the quality metric to measure how well the optimization problem is solved. The divergence function is comparing the difference between the objective values from two executions. The reference execution is the one achieving the minimum objective value.

3.1.6 Radar processing

The radar processing application [37] was developed by a team at the University of Chicago. Unlike the five applications described above, this code was already instrumented with knobs, so we used it out of the box as a blind test for our system. This code is a pipeline with four stages. The first stage (LPF) performs a low-pass filter to eliminate high-frequency noise. The second stage (BF) does beam-forming which allows a phased array radar to concentrate in a particular direction. The third stage (PC) performs pulse compression, which concentrates energy. The final stage is a constant false alarm rate detection (CFAR), which identifies targets.

Knobs The application supports four knobs. The first two knobs change the decimation ratios in the finite impulse response filters that make up the LPF stage. The third knob changes the number of beams used in the beamformer. The fourth knob changes the range resolution. The application can enter 512 separate configurations using these four knobs.

Error Metrics The signal-to-noise ratio (SNR) is used to measure the quality of the detection. The reference execution is the one achieving the highest SNR.

3.2 Characteristics

3.2.1 Cost versus Error

Figures 3.3a through 3.3l show the Runtime versus Cost for the applications described in the Section 3.1.

In the Runtime versus Error sub-figures of these figures, each point represents a single input and knob settings combination; points that correspond to the same input are colored identically. It can be seen that even for a single input, there are many knob combinations that produce the same output error, and that these combinations have widely different costs. For a given input and output error, we are interested in minimizing cost so only the leftmost point for each such combination is of interest. The Pareto-optimal curve for each application shows these points. Except for the Radar application, each application exhibits significant input variance: the Pareto-optimal points vary greatly across inputs for an application such as GEM, Ferret and SGD.

3.2.2 Errors and Costs as Functions of Knobs

For an application, both error and cost are functions of knobs and inputs. Figure 3.4 shows the functions for GEM application for two inputs. Figure 3.4a shows the function for input MS-Set5 and Figure 3.4b shows the function for the input youtube. In both figures, the top figure shows when $knob_1$ is set to 25, how the error and cost changes with the change of $knob_0$, represented by the pink and blue curve respectively. The bottom figures shows when $knob_0$ is set to 25, how the cost and error changes with the change of $knob_1$. The x-axis represents a knob. The left y-axis represents the error(pink color) and the right y-axis represents the cost in terms of running time(blue color).

Let us look at the error first. From Figure 3.4a, it can be seen that the error is not monotonically changed with the increase of $knob_0$ for the fixed $knob_1$, although there is a trend that the error decreases when increasing the knobs. On the other hand, the error monotonically decreases when the $knob_1$ increases for the fixed $knob_0$. This phenomena can be explained when we look at the computations in GEM. GEM has two components, which are sequentially connected with each other. The output of the first component is the input of the second component. $knob_0$ controls the number of iterations in the first component and $knob_1$ controls the number of iterations of the second component. If we look at each component separately, the quality of the output of each component is monotonically decreases when the corresponding knob increases. Therefore if we fix $knob_0$, the error decreases when $knob_1$ increases. However, when tuning $knob_0$, a better output of the first component may not yield a better input for the second component, so the error may not be monotonically change with $knob_0$. In addition, we can also see the error is non-linear function of the knobs. In Figure 3.4b, the curves are different from the ones for input MS-Set5. In top sub-figure, it is more obviously showing the trend that the error decreases when $knob_0$ increases. In the bottom sub-figure, the error drops much faster with the $knob_1$ increases than in Figure 3.4a. This shows the error behaviors of knobs vary greatly for different inputs.

For input MS-Set5, the cost is almost linear increases with the increases of the knobs. This can be easily understood since the knobs controls the number of iterations and the computation is roughly the same in each iteration. However, for input Youtube, the cost shows very complicated behavior. In Figure 3.4b, in the top sub-figure, the cost first decreases and then increases; in the bottom sub-figure, the cost first increases and then stays flat. In GEM, both knobs controls the maximum number of iterations. The kernel k-means algorithm may converge before it reaches the maximum number of iterations, so setting larger value for the knob may have no effect. That's why in the bottom sub-figure, the cost stays flat. In addition, the first component with some value of $knob_0$ may output a result which lead to a faster converge rate for the second component. Therefore, although the first component runs more iterations, but the second component may run much less iterations, so the overall cost may become less. Overall, the cost may be a complex non-linear functions of knob settings and may vary greatly across inputs.

Figure 3.5 shows the error and cost functions for GEM when the cost metric is changed from runtime to energy consumption. The error function remains the same. The cost function remains roughly the same except for the settings in Figure 3.5a. In general, when the cost metrics changed, the cost functions may change significantly.

Figure 3.6, Figure 3.7, Figure 3.8, Figure 3.9 and Figure 3.10 shows the functions for other applications. Except for the application Radar, other applications have the similar behaviors as GEM. The error and cost functions for Radar are non-linear but they has little input variances.

Figure 3.11a, Figure 3.11b, Figure 3.11c and Figure 3.11d use Ferret as an example to show the dependence between different knobs given a fixed input. In Figure 3.11a, $knob_0$ can only lower the error a little from 1.0 to 0.985 when $knob_1$ and $knob_2$ is set to low values although the cost can be increased. In Figure 3.11b, when $knob_1$ and $knob_2$ are lifted to larger values, tuning $knob_0$ can reduce the error to much lower. $knob_1$ and $knob_2$ has similar phenomena when other knobs are fixed at increased levels shown in Figure 3.11a, Figure 3.11b, Figure 3.11c and Figure 3.11d. This shows that in order to search a knob setting to make the error within a certain bound, it is more efficient to make all knobs reach a certain value than searching on one dimension to its extreme. Consider a simple case: all the knobs has same value range and the values in each knob plays the same effects on error and cost, what is the knob setting which has the lowest cost to achieve a certain error bound? If we consider the knobs construct a multi-dimensional matrix with each dimension represents a knob, the knob setting to search will be located in the diagonal entries of the multi-dimensional matrix. When the knobs have different value range and the values for each knob plays different effects, the knob settings are deviated from the diagonal entries of the matrix.

3.3 Summary

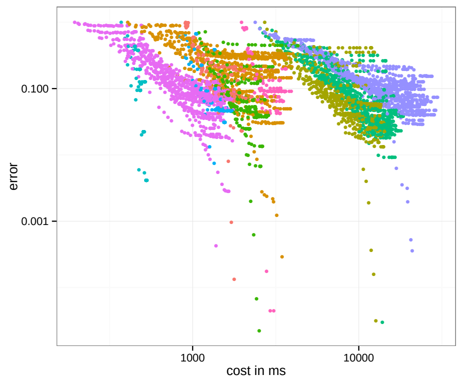
In this chapter, we describe six complex tunable approximate applications in details. In particular, we describe the knobs and the error metric for each application. We show the characteristics of tunable apporixmate algorithms through experiments. The characteristics of the tunable approximate programs can be summarized as follows:

- The Pareto-optimal curves for an application vary greatly across inputs. So for a fixed amount of cost, it may lead to different errors for different inputs. Similarly, a certain amount of error can be achieved by different amount of costs for different inputs.

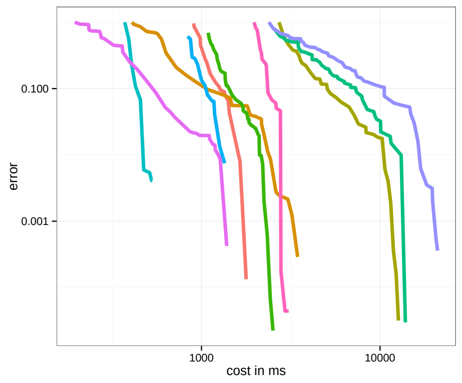
- Error or cost is a function of knobs and the input. For a particular input, there are many different knob value combinations to achieve the same error with different amount of cost or the same cost with different errors. For a fixed knob setting, the error and cost vary greatly across inputs.

- For a particular input, the error function or cost function is usually non-linear and is not monotonic with the change of knobs. In addition, when the cost metric is changed to other metric, the cost functions may change significantly.

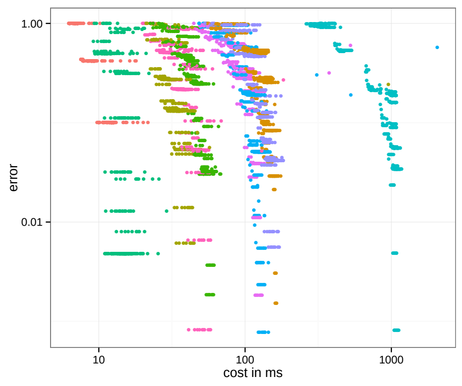
- For a particular input, the error or cost behavior of a knob highly depends on the values of other knobs. For example, changing a knob may only change the error within a small range when setting other knobs to their small values. On the other hand, changing a knob can change the error to a increasing range when setting other knobs to their higher values. The dependence is usually not an additive relationships. This dependence also vary greatly across inputs.



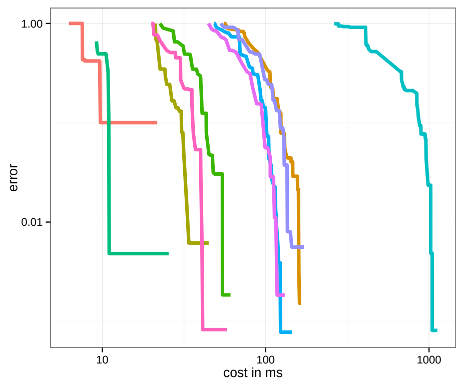
(a) GEM: Runtime vs. Error



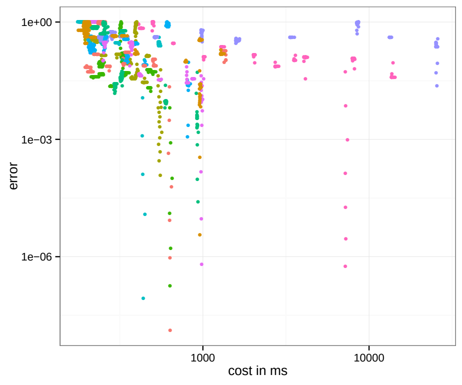
(b) GEM: Pareto-Optimal Curves



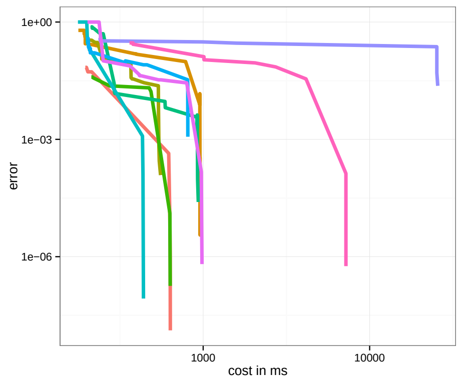
(c) Ferret: Runtime vs. Error



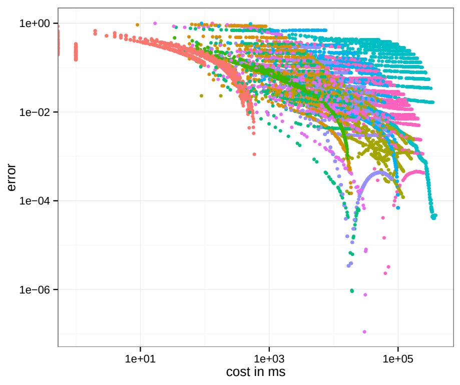
(d) Ferret: Pareto-Optimal Curves



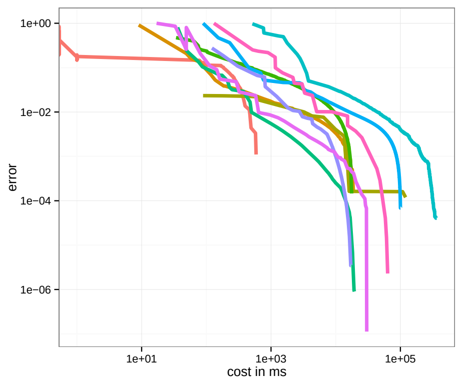
(e) Bullet Runtime vs. Error



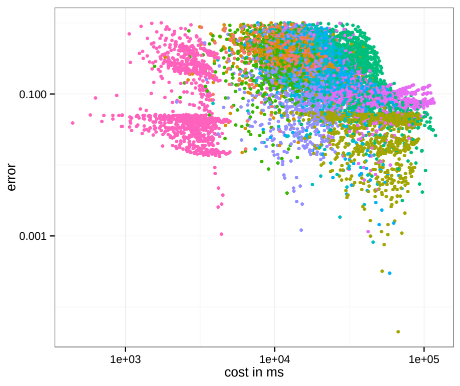
(f) Bullet: Pareto-Optimal Curves



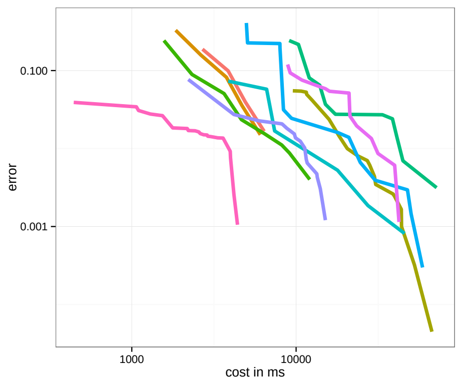
(g) SGD: Runtime vs Error



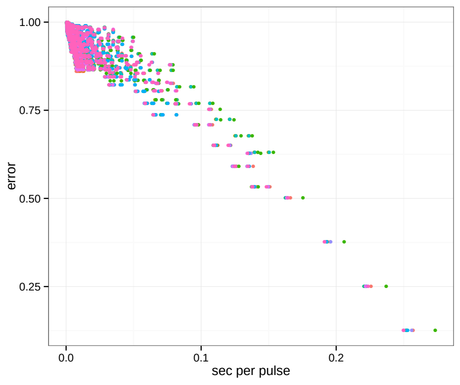
(h) SGD: Pareto-Optimal Curves



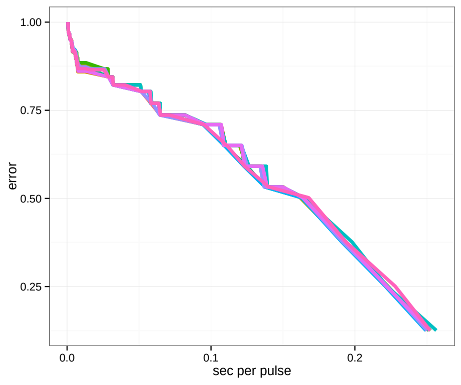
(i) OpenOrd: Runtime vs. Error



(j) OpenOrd: Pareto-Optimal Curves

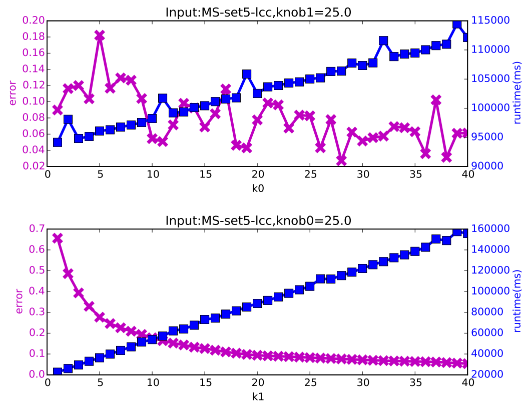


(k) Radar: Runtime vs. Error

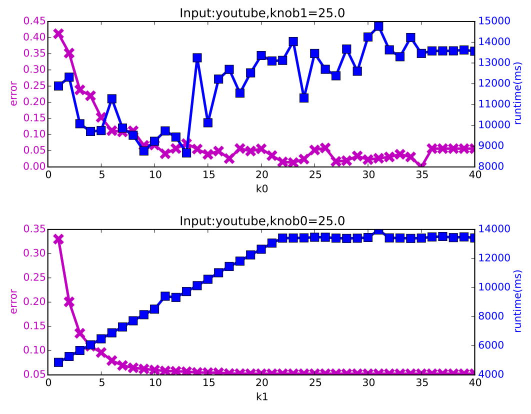


(l) Radar: Pareto-Optimal Curves

Figure 3.3: Runtime vs Error and Pareto-Optimal Curves

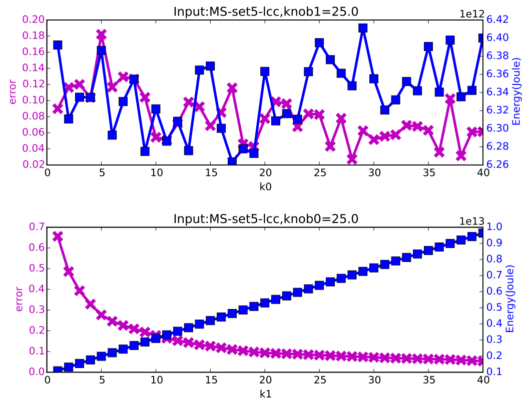


(a) Input: MS-Set5

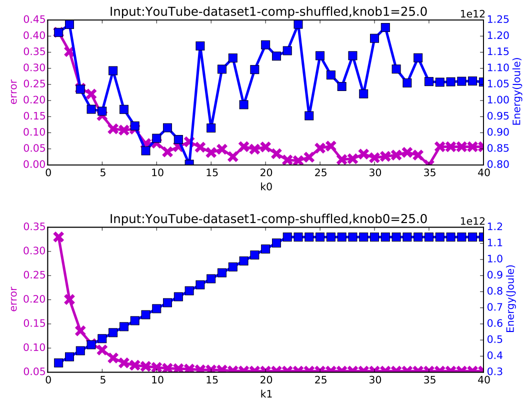


(b) Input: Youtube

Figure 3.4: GEM: the relationships between knobs, error, and runtime for input MS-Set and Youtube

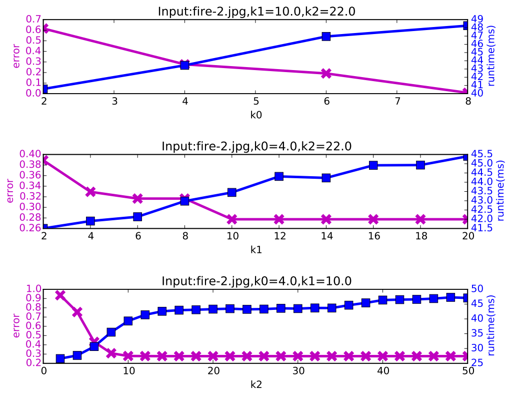


(a) Input: MS-Set5

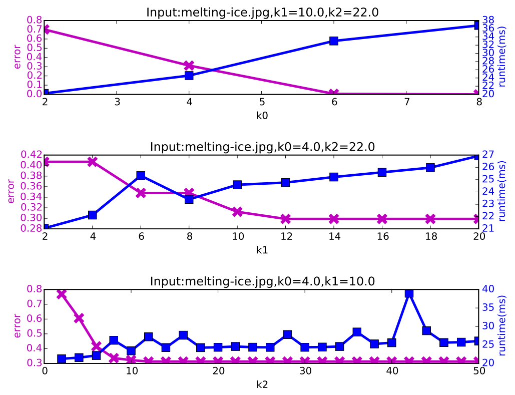


(b) Input: Youtube

Figure 3.5: GEM: the relationships between knobs, error, and energy for input MS-Set and Youtube

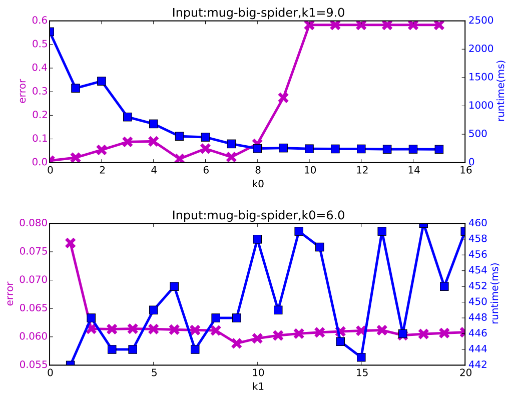


(a) Input: fire

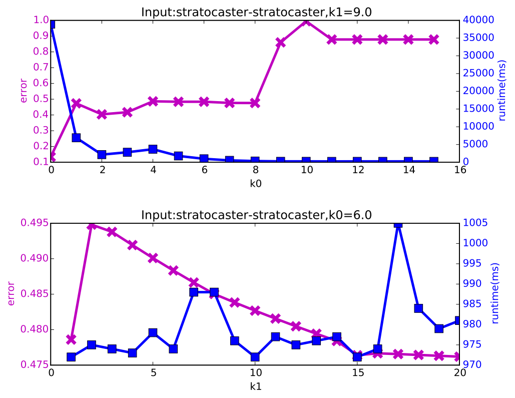


(b) Input: melting-ices

Figure 3.6: Ferret: the relationships between Knobs, Error, and Runtime for input Fire and Melting-Ices

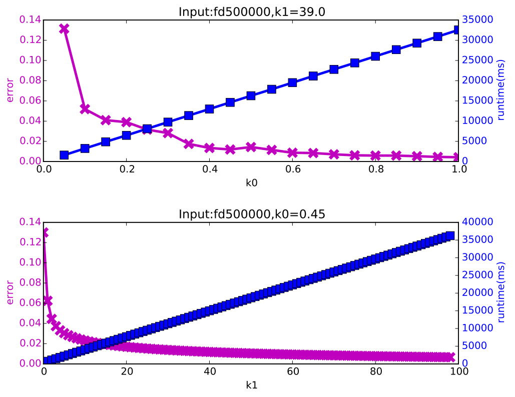


(a) Input: MugSpider

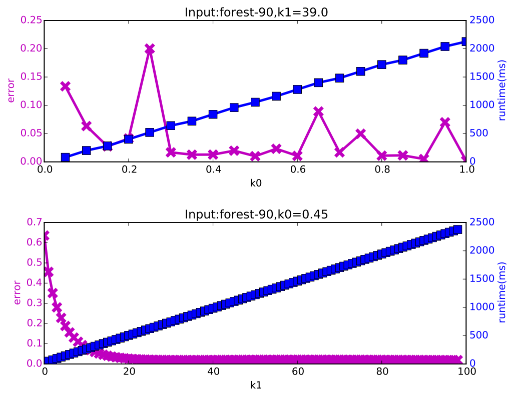


(b) Input: StratoCaster

Figure 3.7: Bullet: the relationships between knobs, error, and runtime for input MugSpider and StratoCaster

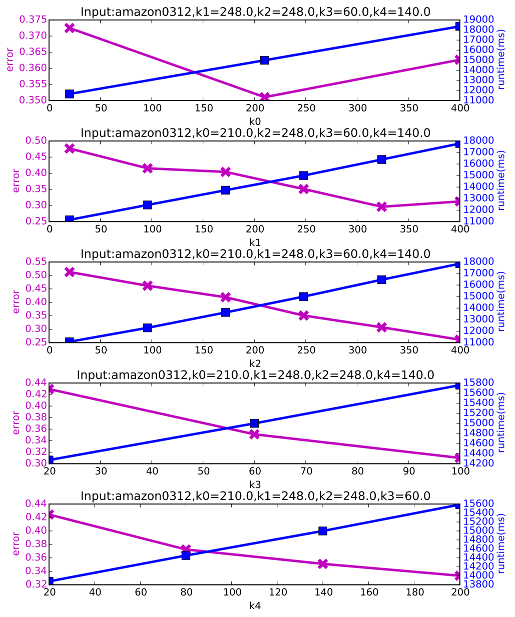


(a) Input: fd500000

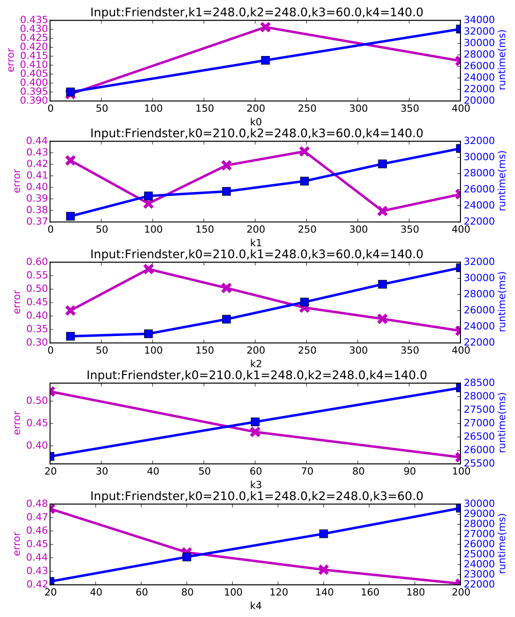


(b) Input: forest

Figure 3.8: SGD: the relationships between knobs, error, and runtime for input Epsilon and Forest

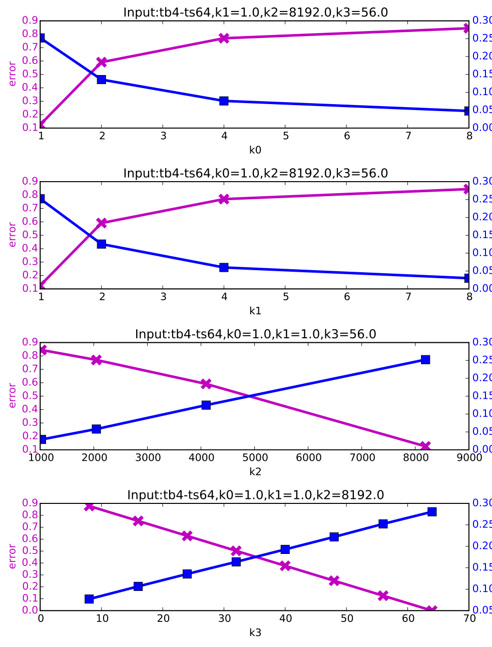


(a) Input: Amazon

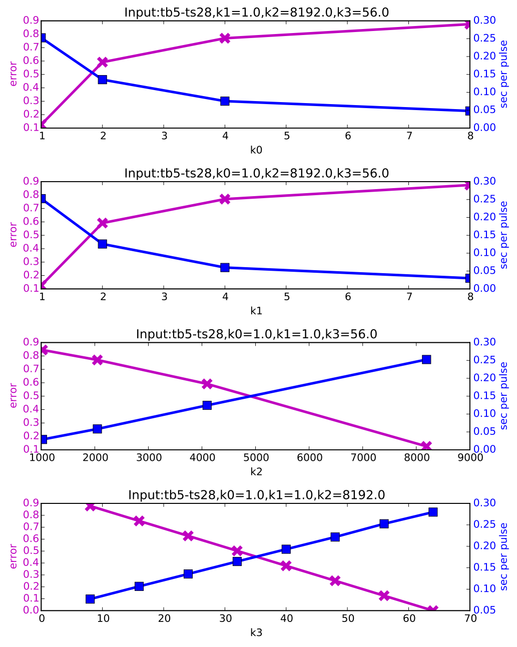


(b) Input: Friendster

Figure 3.9: OpenOrd: the relationships between knobs, error, and runtime for input Amazon and Friendster

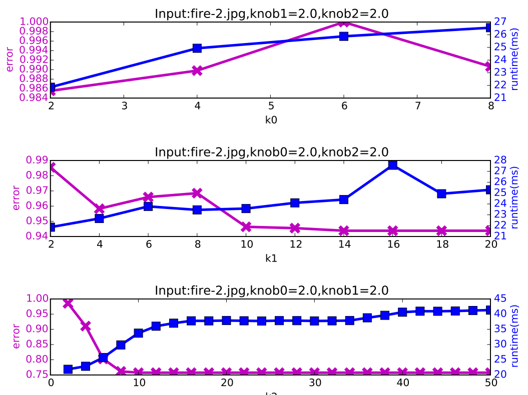


(a) Input: tb4-ts64

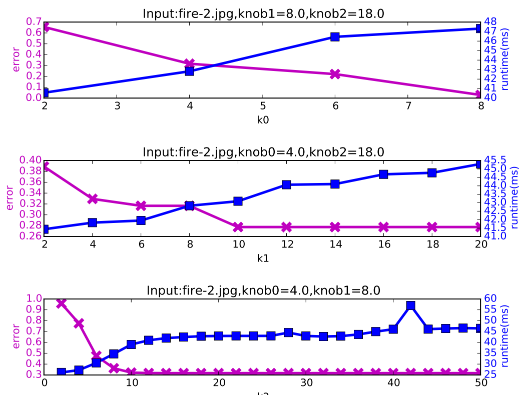


(b) Input: tb5-ts28

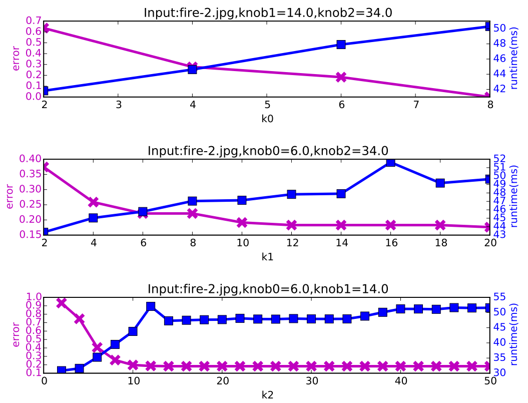
Figure 3.10: Radar: the relationships between knobs, error, and runtime for input tb4-ts64 and tb5-ts28



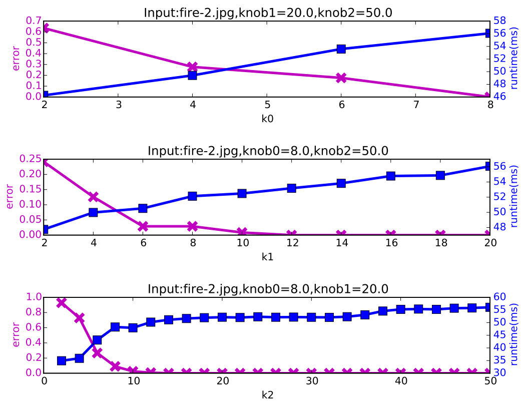
(a) Ferret: Setting 1



(b) Ferret: Setting 2



(c) Ferret: Setting 3



(d) Ferret: Setting 4

Figure 3.11: Ferret: the dependence between knobs for an input

Chapter 4

Formulation of the Control Problem

In this chapter, we give a precise mathematical formulation of the problem of controlling knobs optimally, given a quality bound for the output of a tunable approximate program. We justify this formulation by describing other reasonable formulations and explaining why we do not use them. To keep notation simple, we consider a program that can be controlled with two knobs K_1 and K_2 that take values from finite sets κ_1 and κ_2 respectively. We write $K_1 : \kappa_1$ and $K_2 : \kappa_2$ to denote this, and use k_1 and k_2 to denote particular settings of these knobs. The formulation generalizes to programs with an arbitrary number of knobs in an obvious way.

It is convenient to define the following functions.

- Output: In general, the output value of the tunable program is a function of the input value i, and knob settings k_1 and k_2. Let $f(i, k_1, k_2)$ be this function.
- Error/quality degradation: Let $f_e(i, k_1, k_2)$ be the magnitude of the output error or quality degradation for input i and knob settings k_1 and k_2.
- Cost: Let $f_c(i, k_1, k_2)$ be the cost of computing the output for input i with knob settings k_1 and k_2. This cost can be running time or energy or any other quantity associated with program execution which should be optimized.

In Section 4.1, we formulate the control problem as an optimization problem in which the error is bounded for the particular input of interest. This optimization problem is difficult to solve, so in Section 4.2, we formulate a different optimization problem in which the expected error over all inputs is less than the given error bound. In the final variation of this problem, this error bound is satisfied with some probability, which gives the implementation more flexibility in finding low-cost solutions.

4.1 Input-specific error bound

One way to formulate the control problem informally is the following: given an input value and a bound on the output error, find knob settings that minimize the cost and meet the error bound. This can be formulated as the following constrained optimization problem.

Problem Formulation 1. Given:

- a program with knobs $K_1{:}\kappa_1$ and $K_2{:}\kappa_2$, and
- a set of possible inputs I.

For input $i \in I$ and error bound $\epsilon > 0$, find $k_1{\in}\kappa_1, k_2{\in}\kappa_2$ such that

- $f_c(i, k_1, k_2)$ is minimized
- $f_e(i, k_1, k_2) \leq \epsilon$

In the literature, the constraint $f_e(i, k_1, k_2){\leq}\epsilon$ is said to define the *feasible region*, and values of (k_1, k_2) that satisfy this constraint for a given input are said to lie within the feasible region for that input. The function $f_c(i, k_1, k_2)$ is the *objective function*, and a solution to the optimization problem is a point that lies within the feasible region and minimizes the objective function.

For most tunable programs, this is a very complex optimization problem. As shown in Chapter 3, the Pareto-optimal curves vary greatly across inputs. Therefore, it is difficult to predict the knob settings that produce the Pareto-optimal point, for a given input graph and output error, without exploring much of the space of knob settings for a given input, which is intractable for non-trivial systems.

4.2 Controlling expected error

One way to simplify the control problem is to require only that the *expected* output error over all inputs be less than some specified bound ϵ. Since some inputs may be more likely to be presented to the system than others, each input can be associated with a probability that is the likelihood that that input is presented to the system. This lets us give more weight to more likely inputs, as is done in Valiant's probably approximately correct theory of machine learning [80]. Since the cost function is still a function of the actual input, knob settings for a given value of ϵ will be different in general for different inputs, but the output error will be within the given error bounds only in an average sense.

To formulate this as an optimization problem, we assume that the probability of getting input j is $p(j)$.

Problem Formulation 2. Given:

- a program with knobs $K_1{:}\kappa_1$ and $K_2{:}\kappa_2$,
- a set of possible inputs I, and
- a probability function p such that for any $i \in I$, $p(i)$ is the probability of getting input i.

For input $i{\in}I$ and error bound $\epsilon{>}0$, find $k_1{\in}\kappa_1, k_2{\in}\kappa_2$ such that

- $f_c(i, k_1, k_2)$ is minimized
- $\sum\limits_{j \in I} p(j) f_e(j, k_1, k_2) \quad \leq \epsilon$

This optimization problem seems more complex than the previous one since it also requires knowing the probability of being asked to solve the control problem for each possible input. However, it is intended to capture the intuition that if we can come up with knob settings that work well for the most common inputs, these knob settings will be acceptable in an average sense. Note that the feasible region for this optimization problem does not depend on the particular input i for which the control problem must be solved, which is a major simplification.

4.3 Controlling expected error probabilistically

Formulation 2 has the drawback that even if the *average* error is below ϵ for given knob settings, the error could be large for some likely inputs; if this is unacceptable, there is no way out for the application developer other than to avoid approximation. To solve this problem, we consider a variation of this optimization problem, inspired by [80], in which we are also given a probability π with which the error bound must be met. Intuitively, values of π less than 1 give the control system a degree of slack in meeting the error constraint, permitting the system to find lower cost solutions. This control problem can be formulated as an optimization problem as follows.

Problem Formulation 3. Given:

- a program with knobs $K_1 : \kappa_1$ and $K_2 : \kappa_2$,

- a set of possible inputs I, and
- a probability function p such that for any $i \in I$, $p(i)$ is the probability of getting input i.

For an input $i \in I$, error bound $\epsilon > 0$, and a probability $1 \geq \pi > 0$ with which this error bound must be met, find $k_1 \in \kappa_1$, $k_2 \in \kappa_2$ such that

- $f_c(i, k_1, k_2)$ is minimized
- $\sum\limits_{(j \in I) \wedge (f_e(j,k_1,k_2) \leq \epsilon)} p(j) \quad \geq \pi$

In the rest of this dissertation, we refer to Problem Formulation 3 as "the control problem".

4.4 Discussion

We conclude this section with a justification of the choices made in the problem formulation, and a presentation of experimental results that demonstrate some of the difficulties in solving the control problem.

Design choices: We have framed the control problem in terms of bounding the *expected* error but for some problems, it may be preferable to bound the *maximum* error or some other function of the error distribution. The techniques we discuss here apply to such measures as well. This is discussed in Chapter 6.1.

In Problem Formulation 3, the feasible region depends on the values of π and ϵ, *but the objective function f_c depends on the actual input.* This is an important design choice. The machine learning techniques used in Chapter 5.1 to build models for f_c exploit *features* of the input, and in most problems, these features are simply the parameters in the asymptotic complexity of the algorithm (for example, the asymptotic complexity of GEM is $O(N * K + E)$ where N and E is the number of nodes and edges in the graph respectively and K is the number of clusters). In contrast, our experience is that it is difficult to find input features that are strongly correlated with the error or quality, which is why the feasible region in our formulation depends only on the values of π and ϵ. Therefore, the objective function (computational cost) in our formulation depends on the actual input but the feasible region does not. More insight into how features of the input affect output error will be useful.

Difficulties in solving the control problem: To gain insight into the control problem, it is useful to study the feasible region defined by the inequality for given values of the parameters ϵ and π. Notice that this feasible region is independent of the input. It is intuitively reasonable that if a knob setting (k_1, k_2) is in the feasible region for given parameter values (ϵ_1, π_1), then this knob setting is also in the feasible region for points $(\epsilon \geq \epsilon_1, \pi_1)$ (less stringent output error requirement) as well as points $(\epsilon_1, \pi \leq \pi_1)$ (less stringent likelihood of meeting the output error requirement).

Figure 4.1 illustrates this for the GEM benchmark. Each line in this graph corresponds to one knob setting. This knob setting is in the feasible region for all values of (ϵ, π) on this line or below it. We call this line the *isocline* for the given knob setting. In general, a given point (ϵ, π) will be contained in several isoclines, each of which corresponds to a different setting of knobs, and therefore a different (input-dependent) cost. To solve the optimization problem for a given input, and given ϵ and π values, we must find the lowest cost isocline that contains the point (ϵ, π) (or report that the problem is infeasible if there is no such isocline).

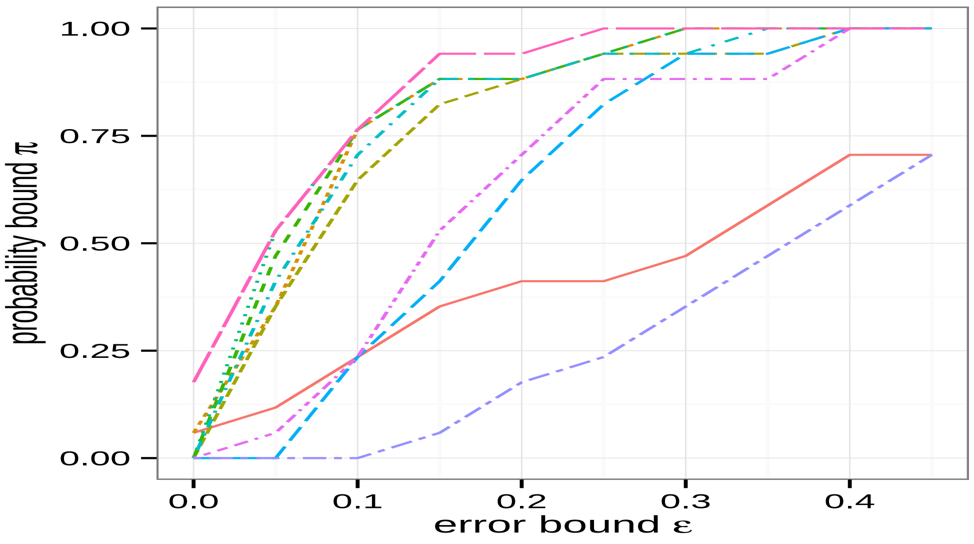


Figure 4.1: Isoclines for GEM benchmark. Colors represent knob settings. The region below each line is the feasible region for that knob setting.

Figure 4.2 shows how the probability of meeting a fixed quality bound ϵ changes as knobs are tuned. The x- and y-axes represent knob settings, and each line in the graph is a contour for some probability value. If output quality increases monotonically with knob settings, we would expect the probability contours to be shaped roughly like rectangular hyperbolas, with higher probability contours being closer to the top-right corner of the graph. This can be seen for small values of probability in Figure 4.2. The more complex contours for higher values of probability arise from the fact that output quality does not increase monotonically with the setting of knob1, which adds to the complexity of the control problem.

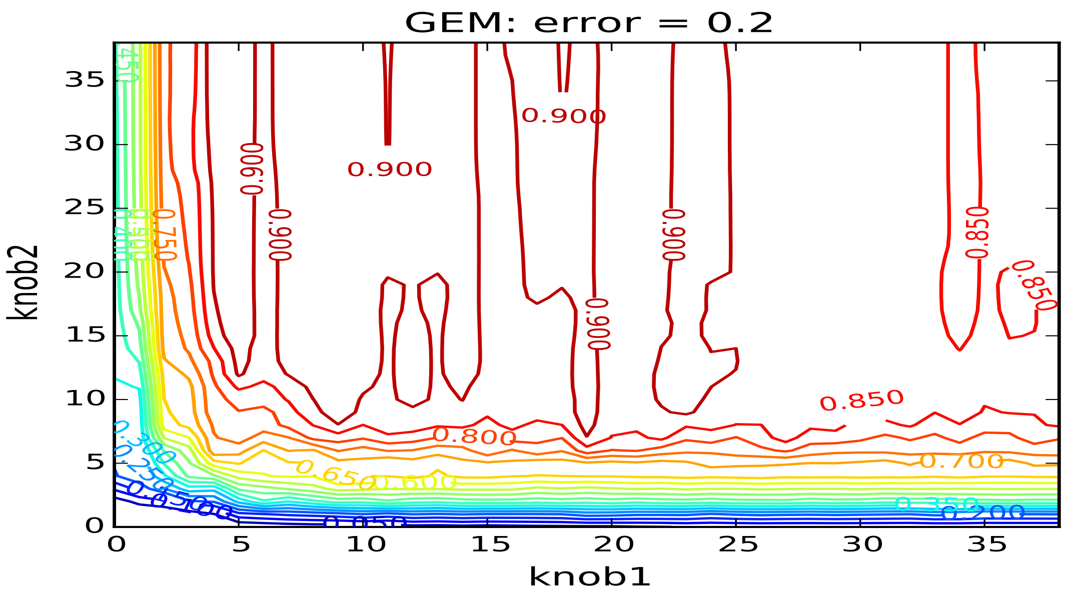


Figure 4.2: Probability contours for GEM. Lines represent the probability of achieving an error of 0.2.

4.5 Summary

In this chapter, we discussed a number of reasonable formulations of the problem of setting knobs optimally given a quality/error bound on the output of a tunable approximate program. These explorations led to propose a formulation based on ideas from Valiant in which the quality/error bound is met probabilistically. We show how to solve this formulation of the control problem next.

Chapter 5

A solution to the proactive control problem

To solve the control problem for a given tunable program, we need the following information.

- I and $p(i)$: the set of possible inputs, and the probability of being presented with input $i \in I$.
- $f_e(i, k_1, k_2)$: error for input i, and knob settings k_1 and k_2.
- $f_c(i, k_1, k_2)$: cost function for input i, and knob settings k_1 and k_2.

Figures in Section 3 and 4.2 show that for the kinds of complex applications considered here, it is difficult if not impossible to derive closed-form analytical expressions for the functions f_e and f_c. Therefore, we use machine learning to learn these functions, given a suitable collection of training inputs.

Figure 5.1 is a pictorial representation of the overall system. For a given program, the system must be provided with a set of training inputs, and metrics for the error/quality of the output and the cost (for example for GEM, the quality metric might be the normalized cut and the cost metric might be the running time or total energy, as explained in Section 1.2). The off-line portion of the system runs the program on these inputs using a variety of knob settings, and learns the functions f_e and f_c. These models are inputs to the controller in the online portion of the system; given an input and values of ϵ and π, the controller solves the control problem to estimate optimal knob settings.

We have implemented our system in a extensible way to permit easy exploration of a variety of machine learning and optimization strategies. In this dissertation, we restrict the discussion here to particular choices of these strategies. For f_e, we use a Bayesian network [55] that captures the relationships between errors from different tunable components; this is described in Section 5.1. For f_c, we consider both running time and energy, and use the tree-based model in the M5 system [58]. This cost model uses a classification of the input based on features provided by the application programmer, and is described in Section 5.2. To solve the control problem in Section 5.4, we uses an simple exhaustive search over the space of knob settings, which is possible when the space of knob settings is small. Scaling to a very large space of knob settings is discussed in Section 6.2.

5.1 Error Model

The error model is used to determine whether or not a knob setting is in the feasible region. Intuitively, Problem Formulation 3 says that a knob setting is in the feasible region if the inputs for which the error is between 0 and ϵ have a combined probability mass greater than or equal to π.

We use Bayesian networks [55] to estimate this. A Bayesian network is a graphical model that represents the probabilistic relationships among random variables. It is a directed acyclic graph (DAG), where each node represents a random variable in the model and an edge represents the dependence relationships between the variables corresponding to the nodes of its end points. Each node or variable may take one of its possible states and the probability

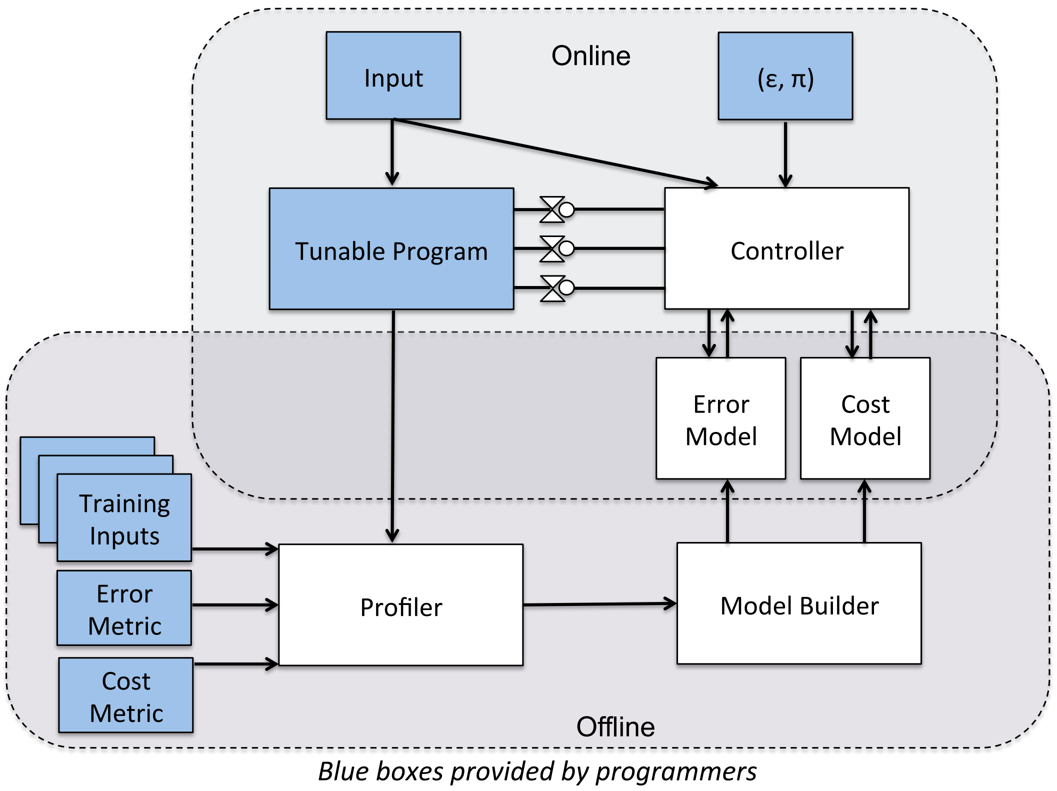
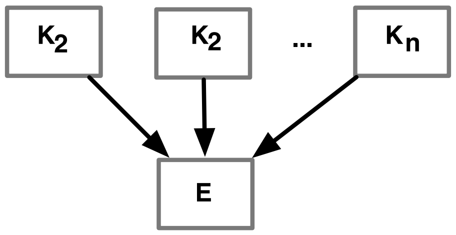


Figure 5.1: Overview of control method

of a state is directly affected by the states of predecessor nodes, if any, in the graph. To quantify this influence, each node is associated with a conditional probability distribution (CPD). The CPD of a node describes the conditional probability distribution of the node's associated variable, given the states of its dependent variables. When a random variable is continuous (as is the case with errors), Bayesian networks require a closed-form distribution to be specified for that variable. The closed-form distribution is generally not available for real-world applications we've observed. Therefore, without making any assumptions about the possible forms of distributions, we discretize the continuous random variables and use a discrete Bayesian network and Bayesian inference to determine the probability that the final error is between 0 and ϵ given a knob setting $(v_1, v_2, ..., v_n)$.

There are several ways to model the error probability distribution using a Bayesian network. A simple model for n knobs is shown in Figure 5.2. In this model, each of the knobs and the error is modeled as a random variable and the error depends on all of the knobs. This simple model works well for the applications we have investigated, but our system allows new models for error to be plugged in easily into the overall framework.



Conditional Probability Distribution(CPD) for E

K1	K2	...	Kn	E	Prob
1	1	...	1	0.1	0.1
1	1	...	2	0.2	0.4
...	...	...	...	...	...

Figure 5.2: Bayesian Network for Error Modeling

5.2 Cost Model

For cost, we model the running time for a given input and knob settings (in Section 5.6.6, we show that this approach can also model energy). The running time can vary substantially depending on the input; after all, even for simple algorithms like matrix multiplication, the running time is a function of the size of the input. For complex irregular algorithms like the ones considered in this paper, running time in general will depend also on other features of the input. For example, the running time of a graph clustering algorithm is affected by the number of vertices and edges in the graph as well as the number of clusters. Therefore, the running time is usually a complex function of input *features* and knob settings.

We use M5 [58], which builds tree-based models, to model the cost function f_c. Input features and knob setting define a multidimensional space, and the tree model corresponds to dividing the space into a set of sub-spaces. In each of the subspace, a linear model is constructed to approximate the function. This model is a multi-dimensional analog of a piecewise linear function in mathematics. Intuitively, this model can approximate cost well because the running time does not usually exhibit sharp discontinuities with respect to knob settings. For example, an iterative algorithm to solve the graph clustering problem may have a knob to control the number of iterations. A small change of the knob value corresponds to the small change of the number of iterations, and it is reasonable to assume the iteration space can be divided into areas where the amount of work per iteration does not change dramatically.

5.3 Learning the Models

The error and cost models are built by learning from the data obtained by executing the programs on a set of representative inputs. The program is profiled exhaustively over the knob space. During each execution of the program, the following data is collected for learning the models.

Learning the Bayesian network model Although the topology of the Bayesian network model is fixed, the conditional probability tables are usually not known in advance. We use a standard Bayesian network learning algorithm, Bayesian posterior estimates [55], to learn the CPD's. The user must provide the error metric for evaluating the error in the program output (Section 3.1 has several examples). During program execution, the knob setting and the corresponding errors are collected.

Learning the cost model The M5 model is constructed automatically from a set of training data. Each training data point is a vector of input features, knob settings and the running time. The input features are application specific and need to be specified by the programmer. The training data is recursively partitioned to form a tree and in each leaf of the tree, multi-variate linear regression is used to construct a linear model. The details of the learning algorithm are described in the classic M5 paper [58].

5.4 Control Algorithms

To solve optimization Problem 3, the control algorithm must search the space of knob settings to find optimal knob settings, using the error and cost models as estimates of f_e and f_c respectively. Our system is implemented so that new search strategies can be incorporated seamlessly.

If the error and cost models are tractable and each knob has a finite number of settings κ, we can use the algorithm shown in Algorithm 1 to solve optimization Problem 3. We sweep over the space of knob settings, and for each knob setting, use the error model to determine if that knob setting is in the feasible region. The cost model is then used to find a minimal cost point in the feasible region.

The exhaustive search strategy is guaranteed to find the optimal solution. It is useful to show that the effectiveness of the combined error and cost models without introducing extra noise from search approach. However, it cannot scale to large knob space. In Chapter 6, we discuss using heuristic-based search strategy to scale the control algorithm to large knob spaces.

Algorithm 1: The control algorithm based on exhaustive search

```
input  : error bound $\epsilon$ and probability bound $\pi$, input features
output: A knob setting
for $(v_1,v_2,...,v_n)$ in space of knob settings do
    $P \leftarrow BayesModel.P(0 \leq E \leq \epsilon | k_1 = v_1, k_2 = v_2, ..., k_n = v_n)$;
    if $P \geq \pi$ then
        Add $(v_1, v_2, ... ,v_n)$ to feasible region ;
    end
end
return CostModel.Min(input features, FeasibleRegion);
```

5.5 Oracle control

To evaluate the effectiveness of overall control system for given choices of the error model, cost model, and search strategy, it is useful to define an *oracle control* that implements an (i) exhaustive search based control algorithm, using (ii) the actual measured error and (iii) the actual measured cost, for a given input and ϵ and δ values. For most applications and inputs, running the application takes less time than it takes to invoke this oracle control to find optimal knob settings, so the utility of the oracle control is that it permits us to evaluate the effectiveness of the overall system in finding good knob settings.

5.6 Evaluation

We use the applications discussed in Section 3.1 as the benchmarks for evaluating the system. For each benchmark, we collected a set of inputs as shown in Table 5.1. To evaluate the error and cost models, inputs were randomly partitioned into training and testing subsets in the proportions shown.

The input features for modeling the costs for each benchmark are listed as follows.

1. GEM: the number of vertices in the graph, the number of edges and the number of clusters.
2. Ferret: the number of segments in the image.
3. Bullet: the number of triangles in each of the triangle meshes representing the objects.
4. SGD: the number of training instances, the dimension of a training instance and the number of non-zero entries of the training instances.
5. OpenOrd: the number of vertices and the number of edges in the graph.
6. Radar: no input features are used in this application.

All experiments were run on the Texas Advanced Computing Center's Stampede machine. Nodes were equipped with two Xeon E5 processors and 32G memory. We used BNLearn [71] for learning Bayesian networks and gRain [39] for performing inference in Bayesian networks. In addition, we used the Cubist [1] implementation of the M5 Cost Model. We used Python to implement the control algorithm.

Benchmark	#Total	#Train	#Test	Source
GEM	43	26	17	[44], [86]
Ferret	3500	1750	1750	[11]
Bullet	1460	730	730	[14]
SGD	30	18	12	[46], [75], synthetic
OpenOrd	43	26	17	[44], [86]
Radar	128	64	64	synthetic

Table 5.1: Inputs for benchmarks. Inputs are randomly divided into training set and testing set.

	GEM	Bullet	Ferret	SGD	OpenOrd	Radar
Train error model	0.109	0.268	0.927	0.120	0.119	0.072
Train cost model	7.52	49.95	133.37	7.18	15.00	3.80
Control(Exhaustive)	0.034	0.002	0.004	0.020	0.032	0.0009
Max execution time	584.951	1.111	2.59	388.364	221.285	608
Min execution time	0.570	0.010	0.038	0.594	2.362	550
Mean execution time	48.348	0.057	0.341	92.276	76.087	562

Table 5.2: Training time(seconds), running time for Control Algorithm, and Application execution times

We evaluated our system for ϵ ranging from 0.0 to 1.0 and π ranging from 0.1 to 1.0.

5.6.1 Time for Training Models and Executing Control Algorithm

Since training is done offline, training time is not particularly important. Table 5.2 shows the time for training the error and cost models for the five applications. Training time obviously increases with the number of training inputs, but even for Ferret, which has the largest training set, it takes only 0.927 seconds to train the error model and 133.366 seconds (about 2 minutes)

	Bullet						Ferret						GEM					
$\pi \parallel \epsilon$	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.1	NA	NA	1.3	1.4	1.6	1.9
0.9	1.0	1.0	1.0	1.0	1.0	1.4	1.1	1.4	1.4	1.6	1.7	1.9	NA	1.1	1.5	1.9	2.2	2.5
0.8	1.0	1.0	1.0	1.0	1.4	18.3	1.1	1.4	1.6	1.6	1.9	1.9	NA	1.2	1.7	2.1	2.4	2.6
0.7	1.0	1.0	1.0	1.4	18.3	18.3	1.1	1.4	1.6	1.7	1.9	2.0	NA	1.3	1.7	2.1	2.5	2.7
0.6	1.0	1.0	1.4	12.1	18.3	24.3	1.2	1.4	1.6	1.7	1.9	2.0	NA	1.4	2.0	2.2	2.6	2.7
0.5	1.0	1.0	2.2	18.3	24.3	29.0	1.2	1.4	1.6	1.7	2.0	2.0	NA	1.7	2.3	2.5	2.7	3.0

	OpenOrd						Radar						SGD					
$\pi \parallel \epsilon$	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	NA	2.0	2.4	6.3	6.3	5.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	2.5	5.6	14.1	31.3	77.4
0.9	NA	2.9	6.5	6.0	5.9	8.4	1.0	1.0	1.0	1.1	1.1	1.1	NA	11.8	30.3	43.0	56.3	94.9
0.8	NA	5.2	6.4	8.7	8.7	8.5	1.0	1.0	1.0	1.1	1.1	1.1	NA	39.7	52.5	103.7	165.0	184.0
0.7	NA	6.3	6.1	8.5	8.8	8.5	1.0	1.0	1.0	1.1	1.1	1.1	NA	73.3	101.4	139.9	176.1	271.7
0.6	NA	6.0	8.5	8.7	8.5	8.5	1.0	1.0	1.0	1.3	1.3	1.3	1.0	97.5	136.7	207.0	302.0	395.3
0.5	NA	8.5	8.5	8.6	8.4	8.4	1.0	1.0	1.0	1.3	1.3	1.3	1.0	104.6	161.1	259.1	302.0	418.4

Table 5.3: Speedups of the tuned programs for a subset of constraint space.

to train the cost model.

In contrast, the control algorithm is executed online, so it is important for it to be fast. Table 5.2 shows that the time for executing the control algorithm is very small; for example, for SGD, it takes 20 milliseconds. This is an acceptable overhead, particularly for large inputs, as can be seen by comparing these times to the max, min, and mean running times for the applications with maximum quality knob settings shown in the table.

5.6.2 Speedups of Tuned Programs

Speedup is defined as ratio of the running time at a particular knob setting to the running time with the knobs set to maximum quality.

Table 5.3 shows speedups for each application for ϵ values between 0 and 0.5 and π values between 0.5 and 1.0. Each entry gives the average speedup over all test inputs for the knob settings found by our control algorithm given (ϵ, π) constraints in the intervals specified by the row and column indices.

Speedups depend on the application and the (ϵ, π) constraints. For each application, the top-left corner of the constraint space is the "hard" region since the error must be low with high probability. The knob settings must be at or close to maximum, and speedup will be limited. Table entries marked "NA" show where the control system was unable to find any feasible solution for these hard constraints.In contrast, the bottom-right corner of the constraint space is the "easier" region, so one would expect higher speedups. This is seen in all benchmarks. For SGD, significant speedup can be obtained since the asymptotic complexity of the algorithm is proportional to the product of the number of training instances and the number of iterations; therefore, a ten-fold reduction in both these numbers results in a 100-fold speedup.

Overall, we see that controlling the knobs in these applications can yield significant speedups in running time.

5.6.3 Evaluation of control system

While Table 5.3 shows speedups obtained from the knob settings found by the control algorithm in different regions of the constraint space, it does not show how well these constraints were actually met. To provide context, we have evaluated this both for our method and for a similar method using the linear regression model developed in [60] to model both error and running time. The results are shown in Figure 5.3.

For each (ϵ, π) combination, we evaluated the quality of the achieved control as follows:

- For each test input, use the control algorithm to set the knobs. If no knob settings are returned, the tile is colored grey. Otherwise measure

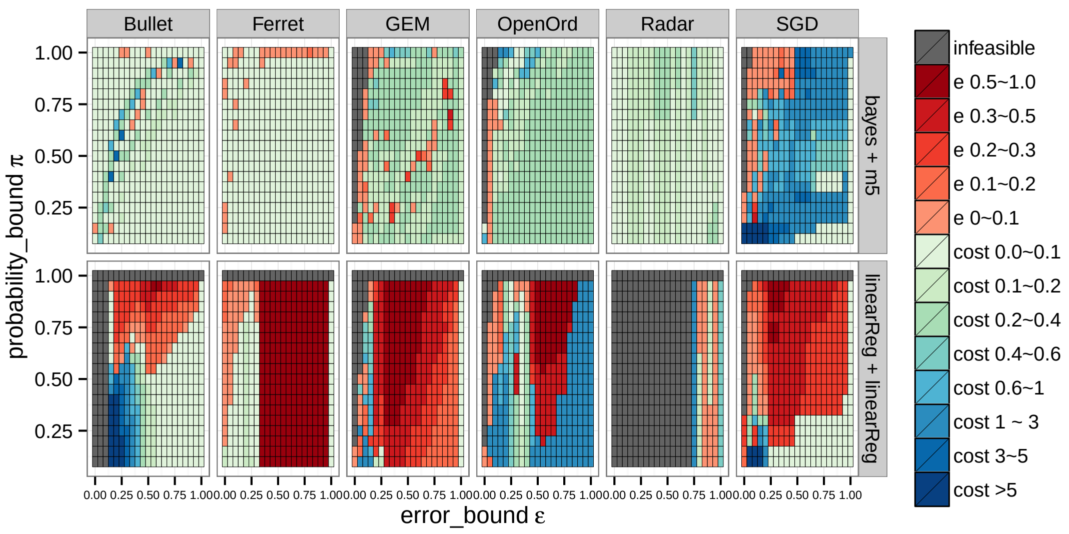


Figure 5.3: Performance of error and cost models for applications

the actual error and the actual running times for this returned knob setting.

- Using the measured actual error for each input, find the probability that the error bound ϵ has actually been met (if all inputs are equally likely, this is just the fraction of inputs for which the actual error is less than or equal to ϵ). If this probability is greater than π, the control algorithm has succeeded in meeting the (ϵ, π) constraint. Give the tile a color between green and blue, depending on how close the average cost (over all inputs) is to the cost of the solution found by the oracle described in Chapter 5.4.

- If the (ϵ, π) constraint has not actually been met, color the tile red. The shade of red indicates how far the achieved average error is from the error of the solution found by the oracle control.

In short, grey tiles indicate that the algorithm was unable to find a solution. Red tiles indicate that the algorithm found a feasible solution but failed to satisfy the (ϵ, π) constraint and the shade of red indicates the degree of failure in the average error. Green-blue tiles indicate that the algorithm succeeded, and the color indicates how close the cost is to the optimal cost solution found by the oracle. Ideally, all tiles would be colored grey to light red, indicating that the error bound was met with low cost.

The top set of squares shows the results from the control system proposed in this paper, while the bottom set of squares shows the results from the control system based on linear regression. Overall, the control system using the Bayes model for error and the m5 model for cost performs quite well for all inputs and regions of the constraint space. The only noticeable problem is in SGD: the cost model seems to be somewhat conservative for this application, which is why there are a lot of blue squares. In contrast, the control system based on linear regression performs quite poorly. No solutions are found in most parts of the space, and even when solutions are found, the cost of the solutions is very sub-optimal.

To further evaluate the quality of the results produced by our method, we performed the same study using an oracle method in which our error and cost models are replaced by actual measurements of the error and running times of the algorithm for each input.

In Figure 5.4 we compare the results produced by this oracle with our method. Even the oracle is sometimes unable to find a solution: this happens in the hard region of the (ϵ, π) space. Even for the oracle, a few points are colored red because the control algorithm of Figure 1 may return different knob settings for different inputs. Therefore, when the entire set of test inputs is

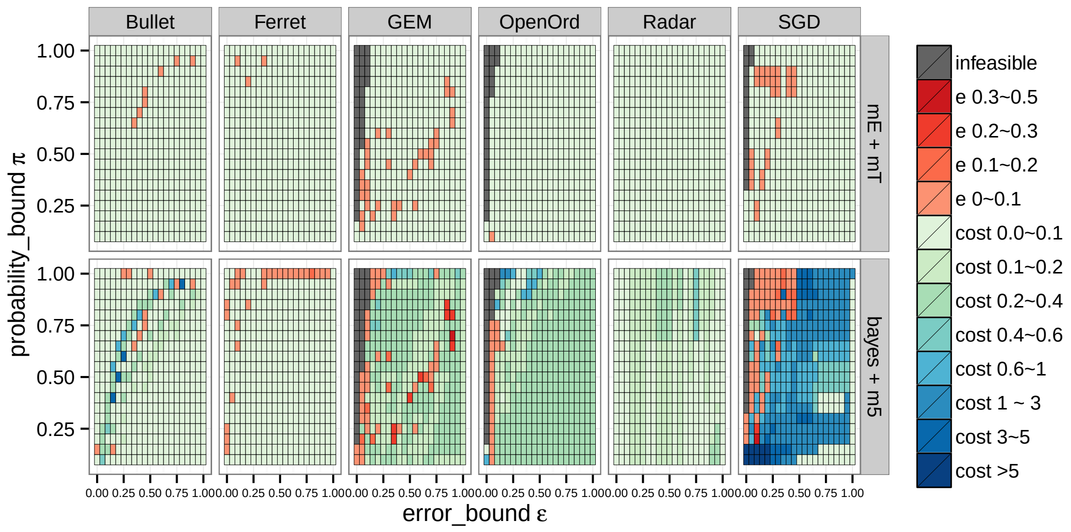


Figure 5.4: Comparing bayes + m5 with oracle on performance of error and cost models

looked at, we may find that the (ϵ, π) constraint is not met even though there are particular knob settings for which the constraint would have been satisfied had they been used for all inputs. Using the Bayesian network to model error and m5 to model cost is fairly successful across the constraint space: for most points, it finds solutions and the cost difference from the oracle's solution is within 40%.

5.6.4 Average Accuracy of the Error and Cost Models

In this section, we estimate how good the different error and cost models are on the average. These estimates can be misleading because a model that is very accurate in an uninteresting portion of the constraint space may be accurate on the average but it is still not very useful if it is inaccurate in the interesting portion on the constraint space! Nevertheless, we present some estimates for the sake of completeness.

	measuredE		bayes		linearReg	
App	mT	m5	mT	m5	mT	linearReg
GEM	0.908	0.842	0.914	0.927	0.124	0.156
Bullet	0.982	1.000	0.980	0.987	0.710	0.679
Ferret	0.992	0.982	0.947	0.935	0.296	0.259
SGD	0.922	0.710	0.801	0.854	0.276	0.276
Radar	1.000	1.000	1.000	1.000	0.333	0.546
OpenOrd	0.997	0.997	0.945	0.945	0.823	0.579
Mean	0.967	0.922	0.931	0.941	0.427	0.416

Table 5.4: Accuracy of the error models. mT represents measuredT.

	measuredE		bayes		linearReg	
App	mT	m5	mT	m5	mT	linearReg
GEM	0.006	0.010	0.006	0.006	0.429	0.416
Bullet	0.0001	0.000	0.0001	0.0001	0.045	0.056
Ferret	0.0001	0.0001	0.0002	0.003	0.271	0.346
SGD	0.003	0.017	0.014	0.011	0.256	0.256
Radar	0.000	0.000	0.000	0.000	0.350	0.198
OpenOrd	0.0001	0.0001	0.004	0.004	0.047	0.134
Mean	0.002	0.005	0.004	0.004	0.233	0.234

Table 5.5: The average gap between the fraction of satisfied inputs and the probability bound(π) for unsatisfied constraints. mT represents measuredT.

Table 5.4 shows the accuracy of the error model. The accuracy is defined as follows.

$$\text{Accuracy} = \frac{\text{number of sastisfied constraints}}{\text{number of total constraints}}$$

Recall that a constraint(ϵ, π) is satisfied when $n_i/t_i \leq \pi$, where n_i is the number of test inputs on which the predicted knob setting achieves error less than ϵ and t_i is the total number of testing inputs. From Table 5.4, we can see both **measuredE** and **bayes** consistently achieve high accuracy across the four benchmarks while **linearReg** achieves very poor accuracy. On average, **bayes** with **m5** achieves 94.1%, which is very close to the accuracy 96.7% achieved by **MeasureE** + **MeasureT**. **LinearReg** only achieves 31.2% accuracy on average.

The reason why **measuredE** does not achieve 100% accuracy is that different knob settings may be used for different inputs, as explained in Chapter 5.6.3.

Table 5.5 shows the average gap between the fraction of satisfied inputs (n_1/t_i) and the probability bound(π) for unsatisfied constraints. Here we see that the gap is very small for **measuredE** and **bayes** across benchmarks. On average, the gap for both models is less than 0.3% for both **measuredT and m5**. As a result, **measuredE** and **bayes** are very close to meeting the unsatisfied constraints. On the other hand, **linearReg** has a significant gap of 23.3% and 23.4% on the average.

Figure 5.5 shows the root mean square error(RMSE) to evaluate the cost models. RMSE is a standard statistical way to evaluate predictions.

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(actual_i - predicted_i)^2}$$

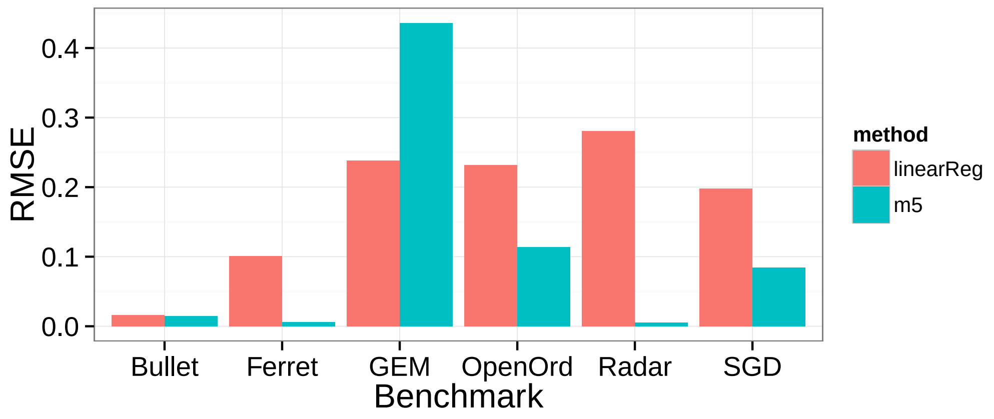


Figure 5.5: RMSE of Cost Models.

The RMSE is computed on all of the testing inputs for all the knob settings. Since **linearReg** predicted relative cost, in order to compare with it, we normalizes the predicted and measured cost. On five benchmarks, **m5** performs better than **linearReg**. On Ferret, **linearReg** is significantly worse than **m5**.

5.6.5 Effect of tuning on individual inputs

While the previous charts showed average behavior over all inputs, it is also interesting to see how responsive the system is to tuning for a given input. Figure 5.6 and Figure 5.7 show how the measured error and measured cost for a number of inputs to GEM evolve with changing probability and error bounds. We see that when we increase the probability bound, **bayes** + **m5** can effectively pick the knob setting to lower errors for different inputs. On the other hand, **linearReg** + **linearReg** does not respond at all π changes.

When we look at each column in Figure 5.6, we see that with the increase of error bound ϵ, **bayes** + **m5** is effective in relaxing the error. The

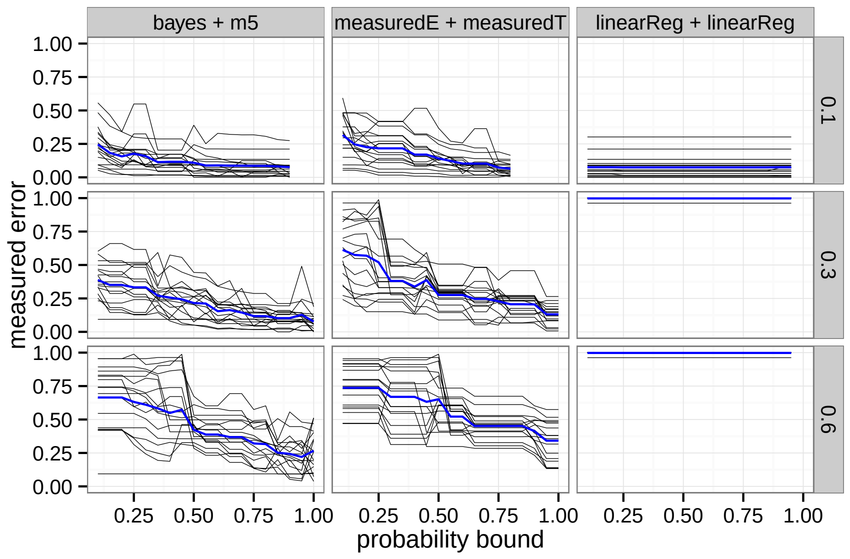


Figure 5.6: Per-input evolution of errors for GEM. Blue line represents the mean of errors across different test inputs.

corresponding row in the Figure 5.7 shows the cost decreases as the error bound ϵ increases, as expected. On the other hand, **linearReg + linearReg** over-relaxes the errors.

5.6.6 Optimizing Energy Consumption

Finally, we note that a major advantage of our approach is that it can be used to optimize not just running time but any metric for which a reasonable cost model can be constructed. Therefore, in principle, the system can be used to optimize metrics such as energy consumption, bandwidth or memory consumption. In this section, we show the results of applying the system to optimizing energy consumption for the same benchmarks.

We measured energy on a Intel Xeon E5-2630 CPU with 16Gb of memory. We used the Intel RAPL (Running Average Power Limit) interface and

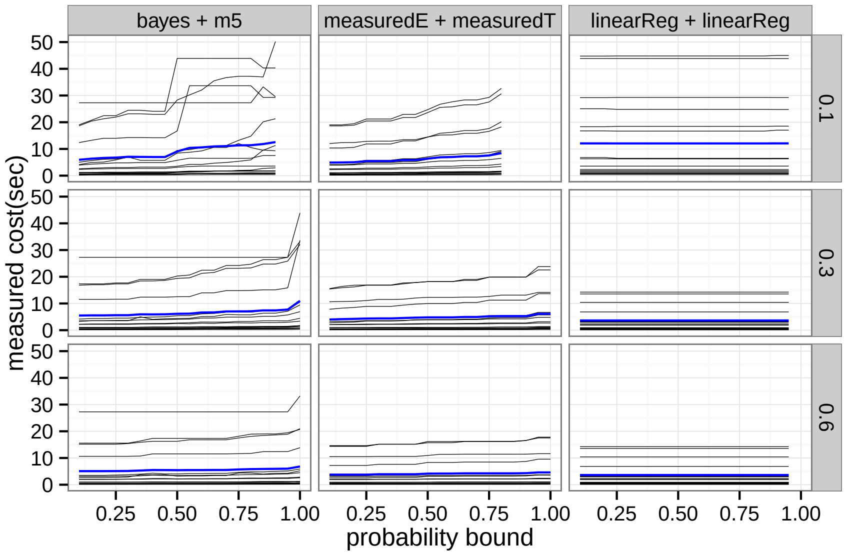


Figure 5.7: Per-input evolution of costs for GEM. Blue line represents the mean of costs across different test inputs.

PAPI to measure the energy consumption. Since this machine does not support DRAM counters, we measured the whole CPU package energy consumption.

Table 5.6 shows the power savings obtained for our benchmarks for ϵ values between 0 and 0.5 and π values between 0.5 and 1.0. Each entry gives the average power savings over all test inputs for the knob settings found by our control algorithm given (ϵ, π) constraints in the intervals specified by the row and column indices. As expected, savings are greater when the constraints are looser.

Figure 5.8 shows the performance of our system compared to the linear regression approach on the four applications. We can see that our approach performs very close to the oracle **measuredE + measureP**, while **linearReg + linearReg** does cannot satisfy a great portion of the constraints.

	Bullet						Ferret						GEM					
π || ϵ	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	1.0	1.0	1.0	1.0	1.0	1.0	NA	1.0	1.0	1.0	1.1	1.1	NA	NA	1.3	1.5	1.7	1.9
0.9	1.0	1.0	1.1	1.0	1.0	1.0	1.0	1.5	1.7	1.8	1.8	1.8	NA	NA	1.7	2.0	2.1	2.3
0.8	1.0	1.1	1.0	1.0	1.0	1.0	1.1	1.6	1.8	1.8	1.8	1.8	NA	1.1	1.8	2.1	2.3	2.5
0.7	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.7	1.8	1.8	1.8	1.8	NA	1.2	1.8	2.3	2.5	2.5
0.6	1.0	1.0	1.0	1.0	1.0	1.2	1.4	1.8	1.8	1.8	1.8	1.8	NA	1.5	2.1	2.3	2.5	2.8
0.5	1.0	1.0	1.0	1.0	1.2	1.3	1.4	1.8	1.8	1.8	1.8	1.8	NA	1.7	2.3	2.5	2.8	2.8

	OpenOrd						Radar						SGD					
π || ϵ	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	NA	2.4	6.1	7.2	7.2	8.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	21.6	59.5	83.3	108.3	107.3
0.9	NA	6.0	7.1	7.2	8.9	8.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	51.0	98.0	149.2	168.7	262.7
0.8	NA	6.0	7.2	8.9	8.9	8.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	91.0	192.5	266.0	265.0	319.0
0.7	NA	7.1	7.2	8.9	8.9	8.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	112.7	193.6	265.0	338.2	319.0
0.6	NA	7.2	8.9	8.9	8.9	8.9	1.0	1.0	1.0	1.3	1.3	1.3	1.0	110.2	193.6	345.1	341.8	410.2
0.5	NA	8.9	8.9	8.9	8.9	8.9	1.0	1.0	1.0	1.3	1.3	1.3	1.0	129.9	254.2	345.1	420.2	410.2

Table 5.6: Energy Savings of the tuned programs for a subset of constraint space.

5.7 Summary

In this chapter, we presented a system using a machine learning based approach to solve the control problem defined in the Formulation 3. Through experiments on a set of complex applications, we showed this system performs well for different error constraints and different cost metrics.

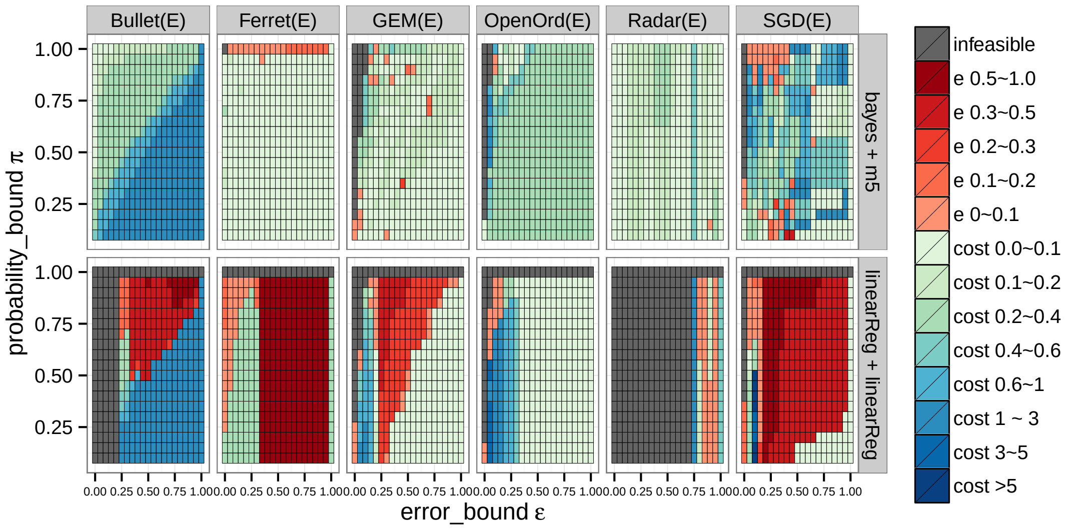


Figure 5.8: Performance of error and energy cost models for all applications

Chapter 6

Extending The Approach

This chapter describes a number of extensions to the basic control system described in Chapter 5.

6.1 Multi-Criteria Error Constraints

6.1.1 Problem Formulation

In Problem Formulation 3, the error constraint is that the probability of achieving an error bound ϵ must be larger than a given bound π. In some circumstances, this may not be adequate. For example, an application may achieve an error bound of less than 0.3 with 0.9 probability, so it may fail to meet this error bound with a probability of 0.1.

However if the application cannot tolerate large errors, we may need to restrict the probability of high errors occurring; for example, we may need to require that the probability of the error being larger than 0.8 must be less than 0.05. Problem Formulation 3 can be easily extended to handle such a requirement by adding additional error constraints. As shown in Problem Formulation 4, multiple constraints can be specified. Each of the constraints specifies the allowed probability lower bound and upper bound for an error range. The above example will have two error constraints:

$$0.9 \geq \sum_{(j \in I) \wedge (0.0 \leq f_e(j,k_1,k_2) \leq 0.3)} p(j) \quad \geq 1.0$$

and

$$0.05 \geq \sum_{(j \in I) \wedge (0.8 \leq f_e(j,k_1,k_2) \leq 1.0)} p(j) \quad \geq 0.0$$

Problem Formulation 4. Given:

- a program with knobs $K_1 : \kappa_1$ and $K_2 : \kappa_2$,
- a set of possible inputs I, and
- a probability function p such that for any $i \in I$, $p(i)$ is the probability of getting input i.

For an input $i \in I$, error bound $\epsilon > 0$, and a probability $1 \geq \pi > 0$ with which this error bound must be met, find $k_1 \in \kappa_1$, $k_2 \in \kappa_2$ such that

- $f_c(i, k_1, k_2)$ is minimized
- $\pi_{l1} \geq \sum_{(j \in I) \wedge (\epsilon_{l1} \leq f_e(j,k_1,k_2) \leq \epsilon_{h1})} p(j) \quad \geq \pi_{h1}$
- $\pi_{l2} \geq \sum_{(j \in I) \wedge (\epsilon_{l2} \leq f_e(j,k_1,k_2) \leq \epsilon_{h2})} p(j) \quad \geq \pi_{h2}$
- $\ldots$
- $\pi_{ln} \geq \sum_{(j \in I) \wedge (\epsilon_{ln} \leq f_e(j,k_1,k_2) \leq \epsilon_{hn})} p(j) \quad \geq \pi_{hn}$

Figure 6.1 shows a pictorial view of the multiple error constraints. The multiple error constraints in Problem Formulation 4 specify the shape of the desired error probability density function. In Figure 6.1, the probability density function of a knob setting is presented with three error constraints. An error constraint $\pi_{li} \geq \sum_{(j \in I) \wedge (\epsilon_{li} \leq f_e(j,k_1,k_2) \leq \epsilon_{hi})} p(j) \quad \geq \pi_{hi}$ specifies that the area under the density function between the error range $[\epsilon_{li}, \epsilon_{hi}]$ is within the

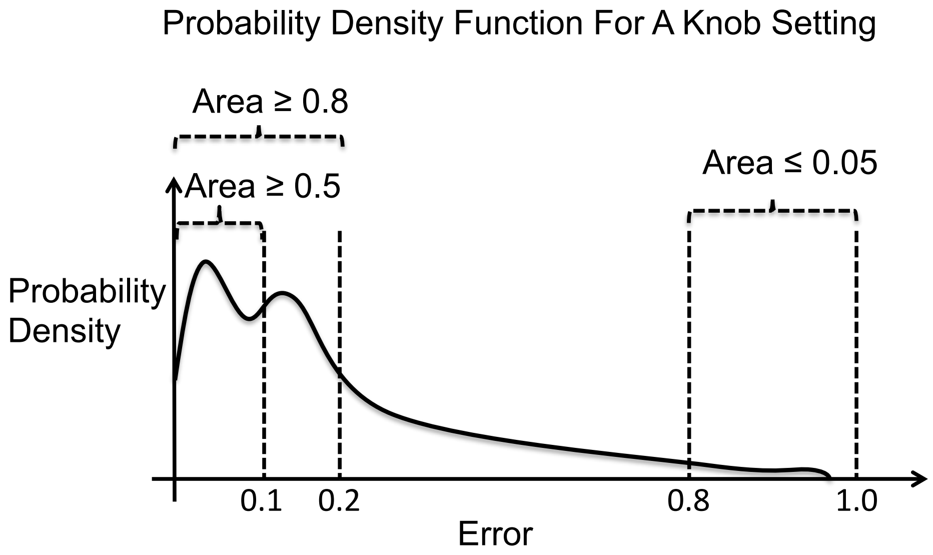


Figure 6.1: Error constraints specify the shape of the probability density function of a knob setting

$[\pi_{li}, \pi_{hi}]$. For example, the third constraint requires that the probability that the error is larger than 0.8 must be less than 0.05.

To solve Problem Formulation 4, only the control algorithm 1 needs to be changed; the error model and cost model remain the same. The modified control algorithm is shown in Algorithm 2: the knob space is swept and all the constraints are checked for each knob setting to find the feasible region.

6.1.2 Evaluation

In the experiments, besides the error constraints used in Chapter 4, we bound the maximum error by the following constraint:

$$0.05 \geq \sum_{(j \in I) \wedge (0.8 \leq f_e(j,k_1,k_2) \leq 1.0)} p(j) \quad \geq 0.0$$

.

Algorithm 2: The control algorithm based for Multiple Constraints

```
input  : error bound $\epsilon$ and probability bound $\pi$
output: A knob setting
for $(v_1, v_2, ..., v_n)$ in space of knob settings do
    $Satisfied = False$ ;
    for error constraint $i$ do
        $P \leftarrow BayesModel.P(\epsilon_{li} \leq E \leq \epsilon_{hi} | k_1 = v_1, k_2 = v_2, ..., k_n = v_n)$ ;
        if $P \geq \pi_{li}$ and $P \leq \pi_{hi}$ then
            $Satisfied = True$ ;
        end
    end
    if $Satisfied$ then
        Add $(v_1, v_2, ... , v_n)$ to feasible region ;
    end
end
return CostModel.Min(FeasibleRegion);
```

Table 6.1 shows the results from adding the extra constraint. We can see that by adding the extra constraint, the speedups decrease for some applications in comparison to the speedups shown in Table 5.3 in Chapter 4. This is because the system chooses more conservative knob settings in order to satisfy the extra error constraint. By keeping the same maximum error constraint, tuning π and ϵ achieves similar behaviors as before: relaxing either π or ϵ can increase the speedups.

Figure 6.2 shows the performance of the error and cost model compared to the oracle. We can see that the knob settings the system chooses satisfy the error constraints for most of error constraint spaces. For applications GEM, OpenOrd and Radar, the cost achieved by the system is close to the cost achieved by the oracle. For SGD and Bullet, the system chooses more

	Bullet						Ferret						GEM					
π/ϵ	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.1	NA	NA	1.3	1.4	1.6	1.9
0.9	1.0	1.0	1.0	1.0	1.0	1.5	1.1	1.5	1.5	1.7	1.8	2.0	NA	1.1	1.6	1.9	2.2	2.5
0.8	1.0	1.0	1.0	1.0	2.3	2.3	1.1	1.5	1.7	1.8	2.0	2.1	NA	1.3	1.7	2.1	2.4	2.6
0.7	1.0	1.0	1.0	1.5	2.3	2.3	1.2	1.5	1.7	1.8	2.0	2.2	NA	1.3	1.7	2.1	2.5	2.7
0.6	1.0	1.0	1.5	2.3	2.3	2.3	1.2	1.5	1.7	1.9	2.1	2.2	NA	1.5	2.0	2.3	2.6	2.8
0.5	1.0	1.0	2.3	2.3	2.3	2.3	1.2	1.5	1.7	1.9	2.1	2.3	NA	1.7	2.3	2.5	2.7	2.9

	OpenOrd						Radar						SGD					
π/ϵ	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.1	0.2	0.3	0.4	0.5
1.0	NA	2.0	2.5	5.9	6.1	5.9	1.0	1.0	1.0	1.1	1.1	1.1	NA	2.6	5.7	14.4	30.4	78.1
0.9	NA	2.9	6.1	5.8	5.9	8.4	1.0	1.0	1.0	1.1	1.1	1.1	NA	11.6	29.9	41.2	50.5	94.8
0.8	NA	4.7	6.2	8.5	8.4	8.0	1.0	1.0	1.0	1.1	1.1	1.1	NA	39.8	52.8	94.3	112.0	169.2
0.7	NA	6.1	5.9	8.2	8.2	8.0	1.0	1.0	1.0	1.1	1.1	1.1	NA	58.0	77.6	130.7	170.1	256.6
0.6	NA	6.1	8.2	8.2	8.0	8.0	1.0	1.0	1.0	1.3	1.3	1.3	1.0	76.0	81.3	162.2	221.8	256.6
0.5	NA	8.4	8.2	8.0	8.0	8.0	1.0	1.0	1.0	1.3	1.3	1.3	1.0	85.3	131.7	201.0	221.8	256.6

Table 6.1: Speedups of the tuned programs for a subset of constraint space when bounding the maximum error: $0.05 \geq \sum_{(j \in I) \wedge (0.8 \leq f_e(j,k_1,k_2) \leq 1.0)} p(j) \geq 0.0$.

conservative knob settings than the oracle. The performance of SGD is similar to the performance shown in Figure 5.3.

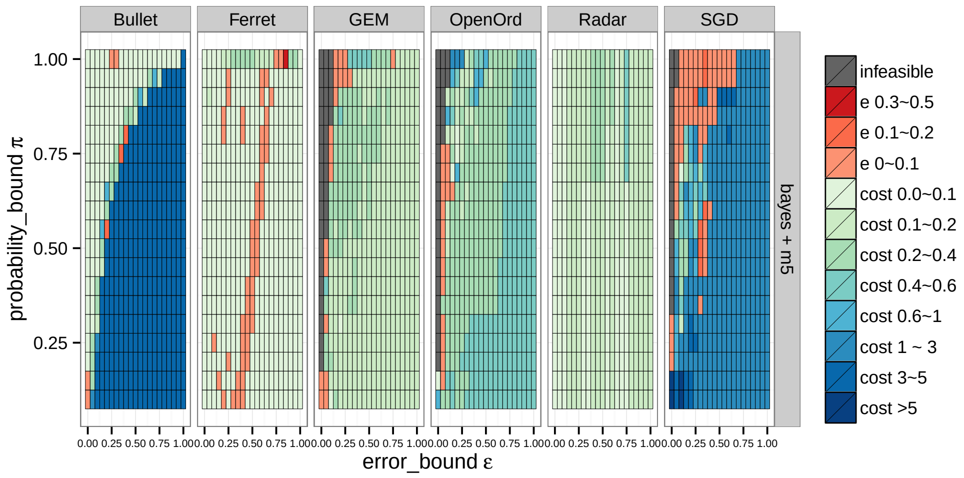


Figure 6.2: Bounding the maximum error: $0.05 \geq \sum_{(j \in I) \wedge (0.8 \leq f_e(j,k_1,k_2) \leq 1.0)} p(j) \geq 0.0$

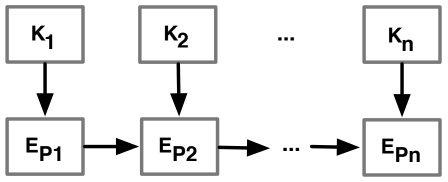
6.2 Scaling to Large Knob Spaces

The basic approach described in the Chapter 4 may not scale to large number of knobs. In this section, we discuss scaling bottlenecks in the basic approach and the corresponding approaches to overcome them.

6.2.1 Scaling Error Models

The conditional probability distribution(CPD) in the simple Bayesian error model discussed in Section 5.2 is a discrete table, requiring $|V_1| \times |V_2| \times ... \times |V_n| \times |E|$ entries, where $|V_i|$ is the number of discrete values for the knob k_i and $|E|$ is the number of discrete values for the error. This number of entries increases exponentially with the number of knobs. There are several ways to make the error model scale to large number of knob spaces.

The number of entries in the CPD in a node in a Bayesian network depends on the number of incoming edges of that node. In the simple Bayesian error model, the number of incoming edges of the error node is equal to the number of knobs. One way to scale this model is to introduce intermediate nodes between the error node and the knob nodes. The introduction of intermediate nodes can be done by decomposing an approximate program into components and modeling the error propagation among the components. For example, a large class of approximate programs can be modeled as a chain of components and each component is associated with a knob to vary the fidelity of the component. In a chain of n tunable components, the error introduced by varying the fidelity of a component is independent of the error introduced by components downstream from it. This independence can be exploited to reduce the complexity of the error model. Figure 6.3 shows a possible Bayesian network for such a program of this sort. The final error can be thought of



Conditional Probability Distribution(CPD)

CPD for E_{Pi} (i>1)

K_i	E_{Pi-1}	E_{Pi}	Prob
1	0.1	0.1	0.1
1	0.1	0.2	0.4
...	...	...	...

CPD for E_{P1}

K_1	E_{P1}	Prob
1	0.1	0.1
1	0.2	0.3
...	...	...

Figure 6.3: The Chain Model

as being accumulated over the chain of components. Each component of the chain model receives a propagated error from its previous component. The error propagated to its output is a combination of its received error and the local error generated by its knob setting. The propagated error of the last component is the final error. In this way, each node in the Bayesian network has two incoming edges, so the CPD does not increase with the number of knobs. This chain model can be easily extended to a DAG(Directed Acyclic Graph) model which will allows more general program structures. The disadvantages of the chain model is that it is hard to deal with programs in which the components form a loop structure. In addition, it requires the users to define the intermediate error metrics, which in certain cases could be hard to define.

Instead of using Bayesian networks, we can use other machine learning techniques to model the error to avoid the exponential increase of the size of CPD. To solve Problem formulation 3, we need an error model that is a function $f(v_1, v_2, ..., v_n, eb)$ which maps a knob setting $(v_1, v_2, ..., v_n)$ and an error bound eb to the corresponding probability defined in the Problem formulation 3. Regression techniques can be applied to learn the function. To train the regression model, we must extract training instances from profiling data. The following steps can be used to generate the training data from the profiling data.

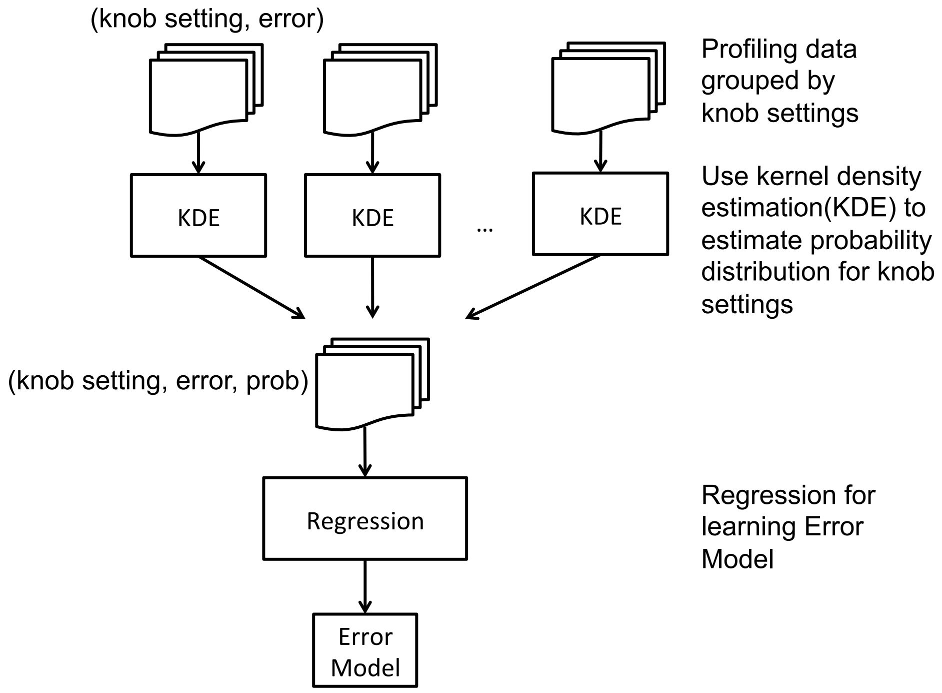
- Kernel density estimation can be used to learn the error probability density function for each knob setting, using the profiling data. Kernel density estimation is a non-parametric approach to estimating the probability density function of a random variable. Non-parametric means it does not assume any closed form distribution for the variable. The kernel density estimator is defined as

  $$\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{x - x_i}{h})$$

  . where K is a kernel which is a non-negative function that integrates to one and has mean zero, and h is a smoothing parameter called the bandwidth. Intuitively, the density at x is estimated by summing up the values of n kernel functions at x.

- For each knob setting in the profiling data, query its probability density function and obtain the probabilities for a set of sampled error values.

The overall procedure is shown in Figure 6.4. To gain more insight into this model, we can look at Figure 6.5 and Figure 6.6. The error model can be thought as a space of one dimensional probability density functions, as shown in Figure 6.5, where the knobs form a space and each knob is associated with a 1-dimensional probability density function. The simple model of Figure 5.2 actually stores the density functions for all the knob settings. Figure6.6 shows how the probability density function changes across different knob settings in the knob space for the application GEM. We can see that from a knob setting to a nearby knob setting in the knob space, the probability density



(knob setting, error)
Profiling data grouped by knob settings
KDE
KDE
...
KDE
Use kernel density estimation(KDE) to estimate probability distribution for knob settings
(knob setting, error, prob)
Regression
Regression for learning Error Model
Error Model

Figure 6.4: Learning the Error Model using KDE and Regression

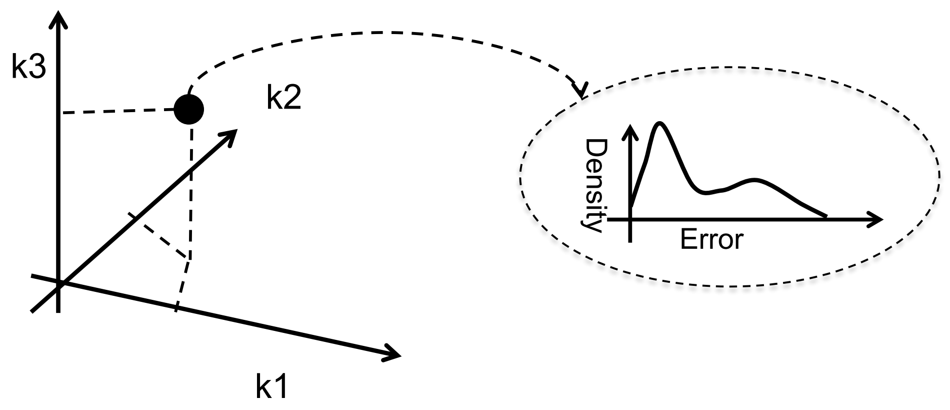


Figure 6.5: A Space of One Dimensional Probability Density Functions

function in many cases changes only slightly. Instead of saving the density functions explicitly for all the knob settings in the knob space, we can save a set of representative density functions and other density functions can be generated from the representative density functions. This is the reason why the regression approach can avoid the exponential space problem in the simple model. For example, we can use M5 model for the regression. Intuitively, we can think the knob space as being partitioned into subspace and in each sub-space, a linear model is used from one density function to generate other density functions.

6.2.1.1 Evaluation

In the experiments, we use M5 for the regression in Figure 6.4 and we use kernel density estimator in Scipy package.

Figure 6.7 shows the result of replacing the Bayesian error model with the model shown in Figure 6.4. For GEM, Bullet, OpenOrd and SGD, the result is a little worse than using Bayesian network error model. In Radar,

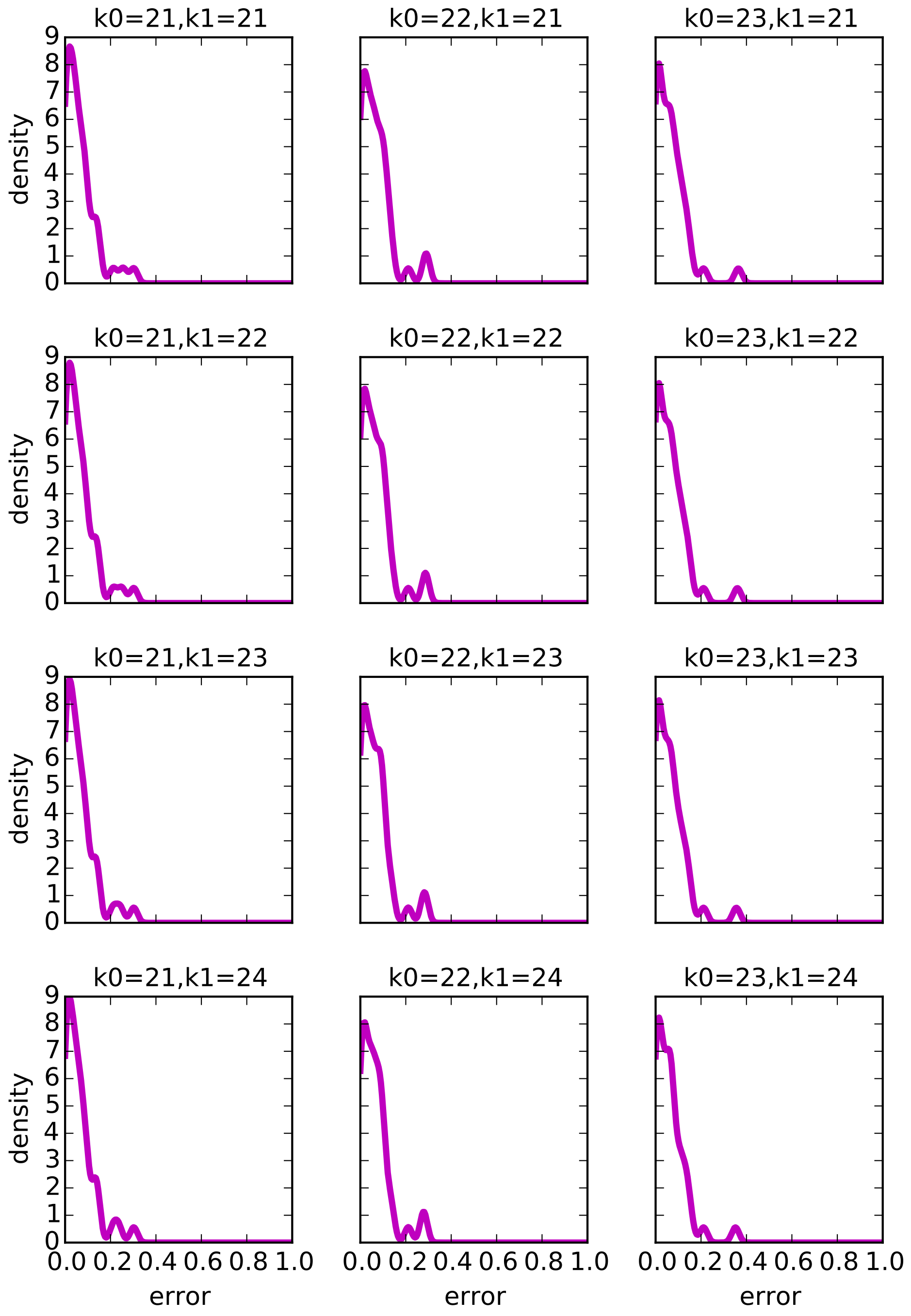


Figure 6.6: the Dynamics of Probability Density of Error for GEM

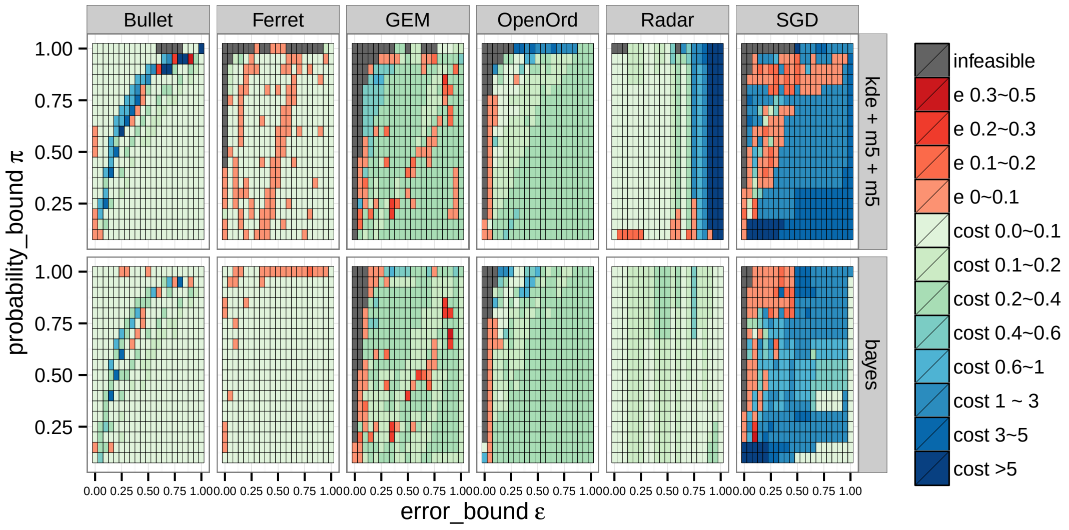


Figure 6.7: Performance of using kernel density estimation based error model for applications

the system chooses similar quality knob settings as the simple model for most of the constraints. However when the error bound is between 0.75 and 1.0, the system chooses very conservative knob settings so the cost is much higher than using the Bayesian network error model.

6.2.2 Scaling Control Algorithms

The exhaustive search in Algorithm 1 cannot scale to large number of knobs. There is a vast literature on non-linear optimization methods and among them, there are many heuristics that can be used to speed up the search, possibly at the cost of finding sub-optimal solutions. Our system is implemented so that new search strategies can be incorporated seamlessly.

As an example, we applied the search strategy used in the Precimonious system [63]. *Precimonious search* is a heuristic search strategy, based on the delta-debugging algorithm [87], to trade off search time for solution quality. The search algorithm is initialized with a change set consisting of all the knobs set to the highest values and it lowers the values of knobs in the change sets iteratively. In each iteration, all the knobs in the change set are lowered to their next levels. If this chosen setting satisfies the accuracy constraint, it continues to the next iteration. Otherwise, the algorithm partitions the change set into subsets and check which subset can be lowered to satisfy the accuracy constraint and achieves the lowest cost. Then the algorithm continues to the next iteration with the chosen subset as the new change set. If none of the subsets can be lowered, a finer partitioning of the change set is checked until the change set cannot be further partitioned. Precimonious can quickly prune the space to find a local minimum solution. There is of course no guarantee that this solution is the global optimum.

6.2.2.1 Evaluation

Table 6.2 shows that using Precimonious to solve the control problem is significantly faster than using exhaustive search. On the other hand, Figure 6.8 shows that the knob settings it finds may be worse than the ones found by the exhaustive search, as one might expect. For OpenOrd, we see that the Precimonious-based system is unable to find knob settings in many more cells, and finds much higher cost knob settings in the top left cells. The main reason for this is that Precimonious uses a greedy search strategy that may get stuck in a local minimum, which in the case of OpenOrd is not the global minimum. However, this experiment shows that the search strategy in our system is not restricted to exhaustive search, and that other search strategies can be used easily.

	GEM	Bullet	Ferret	SGD	OpenOrd	Radar
Control(Exhaustive)	0.034	0.002	0.004	0.020	0.032	0.0009
Control(Precimonious)	0.0008	0.0003	0.0010	0.0018	0.0016	0.0006

Table 6.2: Running Time for Control Algorithm

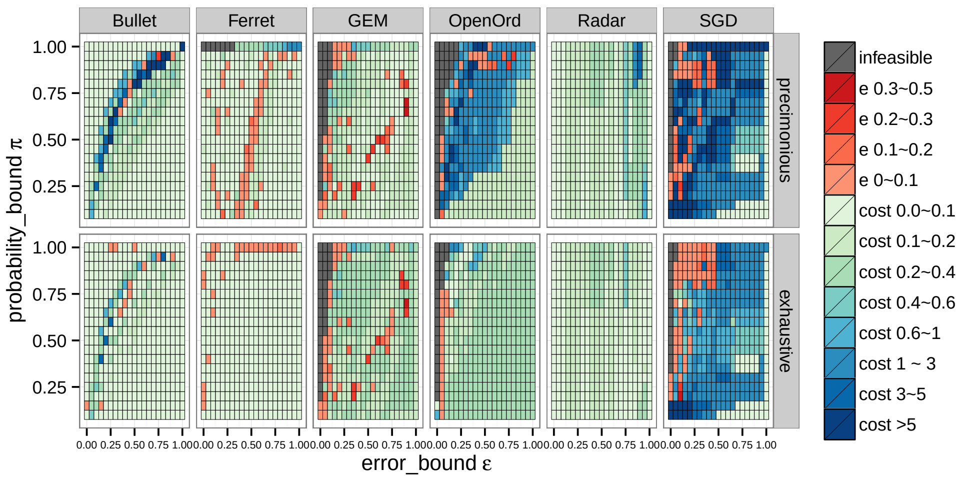


Figure 6.8: Control using precimonious search vs using exhaustive search

6.3 Summary

In this chapter, we discussed extending the control system presented in Chapter 5 to handle multi-criteria constraints. The experiments showed that the extended control system can perform well for multi-criteria constraints. In addition, we discussed ways to scale the system to a large number of knobs by scaling the error model and the search algorithm. Through experiments, we showed the new error model performs a little worse than the simple Bayesian error model, and that the Precimonious search algorithm is much faster than the exhaustive search algorithm although the knob settings chosen are not quite as good.

Chapter 7

Related Work

There is a large body of existing work, much of it produced in the last few years, on exploiting approximation to reduce the running time or energy required to execute a program. This chapter surveys some of this work, and describes how it relates to the work described in this dissertation.

7.1 Exploiting approximation in software

A large body of existing work has focused on making modifications to existing applications to produce approximate versions of these applications. For example, loop perforation [73] explores skipping iterations during loop execution. [60] explores randomly discarding tasks in parallel applications. [61] and [15] explores relaxing synchronization in parallel applications. [78] explores different algorithmic level approximation schemes on a video summarization algorithm. In [52] and [65], methods are developed to recognize patterns in programs that provide approximation opportunities. These techniques could be used to provide knobs automatically and thus complement our work.

7.2 Approximation opportunities in hardware

Researchers have proposed several hardware designs for exploiting approximate computing [27, 28, 50, 57, 68, 72, 76]. [68], [50], [72] and [27] explore introducing approximations into different parts of computer hardware such as arithmetic operations, registers and memory; [28] replaces code segments with hardware-implemented neural networks that approximate the output generated by the code segments. Our techniques can be useful in choosing how to most efficiently map programs onto such hardware and increase the effectiveness of such approaches.

7.3 Reactive control of streaming applications

In this problem, the system is presented with a stream of inputs in which successive inputs are assumed to be correlated with each other, and results from processing one input can be used to tune the computation for succeeding inputs. The Green System [6] periodically monitors QoS values and recalibrates using heuristics whenever the QoS is lower than a specified level. PowerDial [38] leverages feed-back control theory for recalibration. SAGE [66] exploits this approach on GPU platforms. In [29], the authors use simulated annealing to adjust the knob settings. The problem considered in this paper is fundamentally different since it involves proactive control of an application with a single input rather than reactive control for a stream of inputs. However, the techniques described in this paper may be applicable to reactive control as well.

7.4 Auto-tuning

Auto-tuning [3, 84] explores a space of accurate implementations to optimize cost, while the control problem has to deal with both error and cost dimensions. [4, 26] have extended PetaBricks [3] to include an error bound.

[63, 70] use search to lower the precision of floating point types to improve performance for a particular accuracy constraint.

The main difference between our work and auto-tuning is that we build error and cost models that are input-sensitive, so optimal knob settings can be chosen in an input-sensitive way. Since auto-tuning approaches do not build models, they do not have the ability to generalize their results from the inputs they were trained for to other inputs.

7.5 Programming language support

EnerJ [67] proposes a type system to separate exact and approximate data in the program. Rely [17] uses static analysis techniques to quantify the errors in programs on approximate hardware. The work in [16] provides programming language support to help programmers verify approximations. The work in [69] uses Bayesian networks to verify probabilistic assertions. [62] developed tools for debugging approximate programs. None of these deals with controlling the tradeoff of error versus cost.

7.6 Error Guarantees

In [88], the author formulated a randomized program transformation which trades off expected error versus performance as an optimization problem. However, their formulation assumes very small variations of errors across inputs, an assumption violated in all of our complex real-world benchmarks applications. They also assume the existence of an *a priori* error bound for each approximation in the program and that the error propagation is bounded by a linear function. These assumptions make it hard to apply this approach to real-world applications. For example, we know of no non-trivial error bounds for our benchmarks.Chisel [51] extends Rely [17] to use integer linear programming(ILP) to optimize the selection of instructions/data executed/stored in approximate hardware. The ILP constraints are generated by static analysis, which propagates errors through the program. While they consider input reliability, i.e. the probability that an input contains errors, they do not deal with input sensitivity of the error function. Moreover, their error propagation method requires that the error function be differentiable and their static analysis technique cannot deal with input-dependent loops, which are common in our benchmarks and many other applications. ApproxHadoop [34] applies statistical sampling theory to Hadoop tasks for controlling input sampling and task dropping. While statistical sampling theory gives nice error guarantees, the application of this technique is restricted to very special scenarios. [48] uses neural networks to predict whether to invoke approximate accelerators or executing the precise code for a quality constraint.

7.7 Analytic properties of programs

Researchers have developed techniques to verify whether a program is Lipschitz-continuous [19]. Smooth interpretation [18] can smooth out irregular features of a program. Given the input variability exhibited in our applications, analytic properties usually provide very loose error bounds and are not helpful for setting knobs.

Chapter 8

Conclusion and Future Work

Although there is a large body of work on using approximate computing to reduce computation time as well as power and energy requirements, little is known about how to control approximate programs in a principled way. Previous work on approximate computing has focused either on showing the feasibility of approximate computing or on controlling streaming programs in which error estimates for one input can be used to reactively control error for subsequent inputs.

We found most of programs that are suitable for approximate computing have tunable approximate algorithms in them, which have one or more knobs that can be changed to vary the fidelity of the output of the computation. Such tunable approximate algorithms are designed by algorithm experts in the corresponding application domain and their knobs can be used reliably for tuning the fidelity of the output. We investigated a variety of such tunable approximate algorithms. By analyzing the computation patterns in such algorithms, we proposed a classification scheme that captures the approximation patterns in a systematic way. The classification is useful for understanding tunable approximate algorithms from the perspective of approximate computing and help identify appropriate knobs in tunable approximate programs.

We addressed the problem of controlling tunable approximate programs which contain tunable approximate algorithms. We proposed controlling tunable approximate programs in a proactive way. The proactive control problem is suitable for general tunable approximate programs without assuming a stream of inputs or that the fidelity of the output can be estimated quickly. We showed how the proactive control problem for tunable programs can be formulated as an optimization problem, and then gave an algorithm for solving this control problem by using error and cost models generated using machine learning techniques. The machine-learning based approach eliminates the need for programmers to build analytic error and cost models. We validated the approach using a set of complex tunable approximate programs. Our experimental results show that this approach performs well on controlling tunable approximate programs. We believe our techniques are suitable for other tunable approximate programs such as tunable approximate programs with knobs switching between exact hardware or approximate hardware and with knobs changing precision of floating point operations.

8.1 Future Work

In our work, we discover algorithm-level knobs for tuning output fidelity of approximate programs by analyzing the internal algorithms of the approximate programs whereas hardware researchers design approximate hardware to expose hardware-level knobs. It is interesting to experimentally compare the advantages and disadvantages of algorithm-level knobs and hardware-level knobs, e.g the situations in which each kind of knob is more effective for saving energy as well as reducing the quality degradation. In addition, it will be useful to explore how to combine algorithm-level knobs with hardware-level knobs.

As introduced in Section 7.1 and Section 7.1, many existing researches uses other kind of knobs both in software and hardware, instead of using algorithm-level knobs. A direction for future research is to evaluate how well our control approach applies to such knobs. In addition, we evaluated our approach using the cost metrics of executing time and energy consumption. Future work could evaluate other cost metrics such as power, bandwidth and memory consumption.

In Chapter 5, we do not use any input feature for the error model since it is not easy to find input features strongly correlated to errors. However such features may exist. For example, for GEM, the structure of the social network graph structures may be strongly correlated with the errors. There are different ways to characterize social network graph structures such as network diameter and degree distributions. Future work might explore whether such features can enhance the error model or not and how to incorporate such input features into the error model when such features are available. It will be expected that inputs having similar features have similar error behaviors. So one candidate approach to incorporate such features into the error model is clustering the inputs into a set of classes according to the input features and building one error model for each class. Within one class, the error model is expected to be simpler than building one model over all the inputs since the input variance is reduced.

Our control approach can also be applied to streaming applications by treating each input in the input stream as an independent input. Since it does not utilize the feedback from previous inputs, it might set knob settings too conservatively. A direct extension is using the classical feedback control theory in our system to replace the control algorithm. Classical feedback control theory requires a model of the controlled object (called control plant). In our setting, the control plant is the program and the model of the control plant is the error model and cost model. Intuitively, the control algorithm works as follows: 1) it maintains the current estimated probability of achieving the required error bound; 2) when it sees the error for an input, it will re-estimate the probability; 3) if the estimated probability is lower than the probability bound, it will set the knob setting for next input with higher probability of achieving the error bound than the probability of the knob setting for the current input; 4) if the estimated probability is higher than the probability bound, it will set the knob setting with lower probability of achieving the error bound. Future work might implement this approach and compare against the current state of the art reactive control approaches in approximate computing such as Green [6] and SAGE [66].

The Green system [6] builds a simple model for each knob in the program independently without considering the interactions among different knobs in the offline phase. In the online phase, if an input does not achieve the error bound, it will adjust the knob setting by assuming the models for different knobs are additive. If feedback shows the assumption is violated, it uses heuristic search to further adjust the knob settings. We can see that the models built in Green play very limited role when the knobs do not have an additive effect on error. In comparison to the existing reactive approaches, we expect that the models in our system play a more important role in adjusting the knobs when feedback is received.

SAGE [66] does not build error and cost models, but in the offline phase, by profiling a set of inputs, it uses hill-climbing techniques to search for a set of directions for optimal knob setting. Along each direction, the quality of the knob settings decreases for the profiled inputs. In the online phase, SAGE switches between those settings. These directions can be thought of as simple version of the error model.

This dissertation is an attempt to provide a clean foundation for principled control of tunable approximate programs. We hope that other researchers build on this foundation to advance the state of the art in this important area of systems research.

Bibliography

[1] Cubist. `https://www.rulequest.com/cubist-info.html`.

[2] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.

[3] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2011.

[5] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM J. Numer. Anal.*, 45(2):890–904, February 2007.

[6] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation.

In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[7] Judit Bar-Ilan, Mazlita Mat-Hassan, and Mark Levene. Methods for comparing rankings of search engine results. *Comput. Netw.*, 2006.

[8] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[9] Kevin Beason. smallpt: Global illumination in 99 lines of c++. `http://www.kevinbeason.com/smallpt/`.

[10] Pavel Berkhin. A survey on pagerank computing. *Internet Mathematics*, 2:73–120, 2005.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[12] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, 2010.

[13] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, September 2005.

[14] John Burkardt. 3d graphics models. `http://people.sc.fsu.edu/~jburkardt/data/ply/ply.html`.

[15] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In

Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '15, 2015.

[16] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[17] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 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 and Applications*, OOPSLA '13, 2013.

[18] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, August 2012.

[19] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[20] Kumar Chellapilla, Michael Shilman, and Patrice Simard. Optimally combining a cascade of classifiers. volume 6067. SPIE, 2006.

[21] Radha Chitta, Rong Jin, and Anil K. Jain. Efficient kernel clustering using random fourier features. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 161–170, Washington, DC, USA, 2012. IEEE Computer Society.

[22] Arthur Choi, Trevor Standley, and Adnan Darwiche. Approximating weighted max-sat problems by compensating for relaxations. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 211–225, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] Chris Chu and Yiu-Chung Wong. Fast and accurate rectilinear steiner minimal tree algorithm for vlsi design. In *Proceedings of the 2005 international symposium on Physical design*, ISPD '05, pages 28–35, New York, NY, USA, 2005. ACM.

[24] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. MIT Press, 2001.

[25] Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *J. ACM*, 50(2):107–153, March 2003.

[26] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015.

[27] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[28] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceed-*

ings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45, 2012.

[29] Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. Performance portability across heterogeneous socs using a generalized library-based approach. *ACM Trans. Archit. Code Optim.*, 2014.

[30] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 247–254, New York, NY, USA, 1993. ACM.

[31] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[32] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, 1997.

[33] James Glanz. Landlords double as energy brokers. New York Times online article, May 2013.

[34] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[35] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 491–500, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[36] Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[37] Henry Hoffmann, Anant Agarwal, and Srinivas Devadas. Selecting spatiotemporal patterns for development of parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1970–1982, 2012.

[38] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 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, 2011.

[39] Sren Hjsgaard. Graphical independence networks with the grain package for r. *Journal of Statistical Software*, 46, 2012.

[40] I. R. Ismaeil, A. Docef, F. Kossentini, and R. K. Ward. A computation-distortion optimized framework for efficient dct-based video coding. *Trans. Multi.*, 3(3):298–310, September 2001.

[41] Sean Keely. Reduced precision for hardware ray tracing in gpus. In *Proceedings of High-Performance Graphics 2014 (HPG14)*, June 2014.

[42] Sean Keely and Donald Fussell. A reduced-precision architecture for real-time ray tracing. In *Computer Science Technical Report TR-14-05*. The University of Texas at Austin, January 2014.

[43] Jan Klein and Gabriel Zachmann. Time-critical collision detection using an average-case approach. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '03, pages 22–31, New York, NY, USA, 2003. ACM.

[44] Jure Leskovec. Stanford large network dataset collection(snap). `http://snap.stanford.edu/data/`.

[45] Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM.

[46] Chih-Jen Lin. Libsvm classification dataset. `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

[47] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, I3D '95, New York, NY, USA, 1995. ACM.

[48] Divya Mahajan, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, and Hadi Esmaeilzadeh. Prediction-based qualitycontrol for approximate accelerators. In *Second Workshop on Approximate Computing Across the System Stack*, WACAS, 2015.

[49] Shawn Martin, W. Michael Brown, Richard Klavans, and Kevin W. Boyack. Openord: an open-source toolbox for large graph layout. volume 7868, 2011.

[50] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[51] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '14, 2014.

[52] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, 2011.

[53] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in fpga placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, FPGA '01, pages 29–36, New York, NY, USA, 2001. ACM.

[54] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *In Proceedings of Uncertainty in AI*, pages 467–475, 1999.

[55] R. E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003.

[56] Miguel A. Otaduy and Ming C. Lin. Clods: Dual hierarchies for multiresolution collision detection. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, 2003.

[57] Krishna V. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Trans. Comput.*, 2005.

[58] J. R. Quinlan. Learning with continuous classes. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, pages 343–348. World Scientific, 1992.

[59] Laks Raghupathi, Vincent Cantin, Francois Faure, and Marie-Paule Cani. Real-time simulation of self-collisions for virtual intestinal surgery. In *Proceedings of the 2003 international conference on Surgery simulation and soft tissue modeling*, IS4TM'03, pages 15–26, Berlin, Heidelberg, 2003. Springer-Verlag.

[60] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, 2006.

[61] Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, 2007.

[62] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *Proceedings of the Twentieth International Con-*

ference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, 2015.

[63] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[64] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[65] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

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

[67] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 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, 2011.

[68] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[69] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[70] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[71] M. Scutari. Learning Bayesian Networks with the bnlearn R Package. *Journal of Statistical Software*, 35, 2010.

[72] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt. Exploiting partially-forgetful memories for approximate computing. *Embedded Systems Letters, IEEE*, March 2015.

[73] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, 2011.

[74] Erik Sintorn and Ulf Assarsson. Real-time approximate sorting for self shadowing and transparency in hair rendering. In *Proceedings of the*

2008 symposium on Interactive 3D graphics and games, I3D '08, pages 157–162, New York, NY, USA, 2008. ACM.

[75] Soeren Sonnenburg, Vojtech Franc, Elad Yom-Tov, and Michele Sebag. Pascal large scale learning challenge. `http://largescale.ml.tu-berlin.de`.

[76] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, 2014.

[77] Xin Sui, Tsung-Hsien Lee, Joyce Jiyoung Whang, Berkant Savas, Saral Jain, Keshav Pingali, and Inderjit Dhillon. Parallel clustered low-rank approximation of graphs and its application to link prediction. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, LCPC '12, 2012.

[78] Karthik Swaminathan, Chung-Ching Lin, Augusto Vega, Alper Buyuktosunoglu, Pradip Bose, and Sharathchandra Pankanti. A case for approximatecomputing in real-time mobile cognition. In *Second Workshop on Approximate Computing Across the System Stack*, WACAS, 2015.

[79] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[80] L. G. Valiant. A theory of the learnable. *CACM*, 27(11), 1984.

[81] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Gestures as point clouds: A $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction*, ICMI '12, 2012.

[82] P. Waldemar and T. Ramstad. Hybrid klt-svd image compression. In *Proceedings of the 1997 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '97) -Volume 4 - Volume 4*, ICASSP '97, Washington, DC, USA, 1997. IEEE Computer Society.

[83] P. Wesseling. *An introduction to multigrid methods*. Pure and applied mathematics. J. Wiley, 1992.

[84] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98.

[85] Joyce Jiyoung Whang, Xin Sui, and Inderjit S. Dhillon. Scalable and memory-efficient clustering of large-scale social networks. In *ICDM*, 2012.

[86] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009. `http://socialcomputing.asu.edu`.

[87] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[88] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, 2012.