Proactive Control of Approximate Programs [*]

Xin Sui
Department of Computer Science,
The University of Texas at Austin
xinsui@cs.utexas.edu

Andrew Lenharth
The Institute for Computational Engineering and Science,
The University of Texas at Austin
lenharth@ices.utexas.edu

Donald S. Fussell Keshav Pingali
Department of Computer Science,
The University of Texas at Austin
fussell@cs.utexas.edu,
pingali@cs.utexas.edu

Abstract

Approximate computing trades off accuracy of results for resources such as energy or computing time. There is a large and rapidly growing literature on approximate computing that has focused mostly on showing the benefits of approximate computing. However, we know relatively little about how to control approximation in a disciplined way.

In this paper, we address the problem of controlling approximation for non-streaming programs that have a set of "knobs" that can be dialed up or down to control the level of approximation of different components in the program. We formulate this control problem as a constrained optimization problem, and describe a system called Capri that uses machine learning to learn cost and error models for the program, and uses these models to determine, for a desired level of approximation, knob settings that optimize metrics such as running time or energy usage. Experimental results with complex benchmarks from different problem domains demonstrate the effectiveness of this approach.

1. Introduction

Models should be used, not believed.
Henri Theil

There is growing interest in *approximate computing* as a way of reducing the energy and time required to execute applications [3, 4, 41, 46, 55]. In conventional computing, programs are usually treated as implementations of mathematical functions, so there is a precise output that must computed for a given input. 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 short-cuts if human beings cannot tell the difference in the rendered scene.

In this paper, 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. These knobs might control the number of iterations performed by a loop [7, 36], determine the precision with which floating-point computations are performed [38, 43], or switch between precise and approximate hardware [16]; for the purposes of this paper, the source of approximation does not matter so long as the fidelity of the output is changed by adjusting the knobs.

Perhaps the oldest example of a tunable approximate computation is the method used by Archimedes to estimate the value of π by constructing inscribed and circumscribed regular polygons for circles; by increasing the number of sides of the polygons, more accurate estimates for π can be obtained at the cost of additional computation. Iterative algorithms for solving linear or non-linear systems are tunable algorithms since they can produce more accurate solutions if more iterations are executed. Fast multipole methods for solving n-body problems are tunable by design. Not all approximate algorithms are tunable; for example, Kempe's heuristic for graph coloring [13] 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.

Our concern in this paper is the *control problem* for tunable approximate programs: roughly speaking, given a permissible error for the output, we want to set the knobs to minimize computational costs, such as running time or energy, while meeting the error constraint.

For example, consider the GEM algorithm [52] 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 which is defined in more detail in §4, but roughly speaking, it computes the ra-

[*] The work presented in this paper has been supported by the National Science Foundation grants CNS 1111766, CCF 1218568, XPS 1337281, CNS 1406355, and DARPA BRASS contract FA8750-16-2-0004.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872402

tio 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 output quality, how do we set the knobs for the two components to minimize computational time or energy?

This problem has not been considered in most of the existing literature in this area, which is surveyed in §6. Prior work has focused mostly on the *forward problem* of studying what happens to the output of a tunable approximate program when its knobs are dialed up or down [10, 40, 41, 46], and there is little work on the more difficult *inverse problem* of finding optimal knob settings, given a bound on the output quality. One exception is the Green system [4] but it is targeted for streaming applications in which the system is given a *sequence* of inputs such as a sequence of video frames, and the results from processing an input can be used to adjust knob settings for *succeeding* inputs. This kind of *reactive* control is not useful for applications that are not streaming programs, such as GEM, for which knobs must be set *before* the program is executed; in this paper, we call this *proactive* control.

This paper proposes and evaluates a solution to the proactive control problem for non-streaming programs like GEM that consist of components controlled by one or more knobs and in which the error and cost behaviors are substantially different for different inputs. The rest of the paper is organized as follows.

- We formulate the proactive control problem as a constrained optimization problem, dealing explicitly with the problem of input variability. (§2).
- We describe an error model based on Bayesian networks (§3.1), and a cost model for running time, implemented using the M5 system [34] (§ 3.2).
- We describe the Capri system, which incorporates these models, and discuss two algorithms, implemented in this system, for finding optimal knob settings (§ 3.4).
- We validate the proposed approach using five complex, tunable approximate programs (§ 4 and §5). We also describe the results of a blind test in which we use our system to control a third-party radar application.
- We show that Capri can successfully build models for and tune approximate programs to optimize not just running time but energy as well (§ 5.5).

Related work is described in § 6, and conclusions in § 7.

2. Problem definition

We describe the formulation of the proactive control problem we use in this paper, justifying it 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 can be the running time, energy or other execution metric to be optimized.

In §2.1, 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 § 2.2, we formulate a different 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 flexibility in finding low-cost solutions.

2.1 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 (i) meet the error bound and (ii) minimize the cost. This can be formulated as the following constrained optimization problem.

Problem Formulation 1. *Given:*

- *a program with knobs* K_1:κ_1 *and* K_2:κ_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

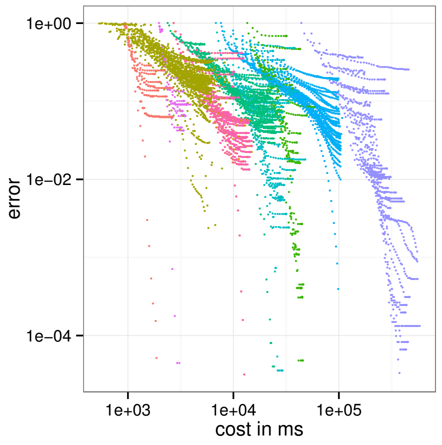


Figure 1. Cost vs. error for GEM benchmark. Each dot represents one knob setting for one input. Different colors represent different inputs.

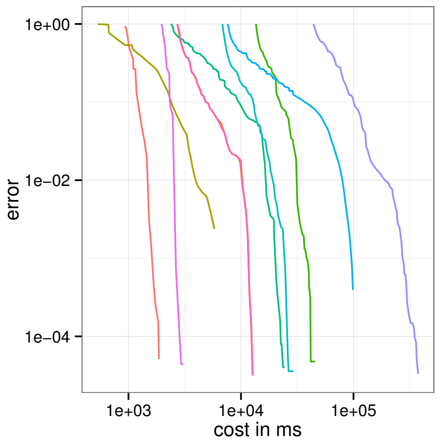


Figure 2. Pareto-optimal curves for GEM benchmark. Different lines represent different inputs.

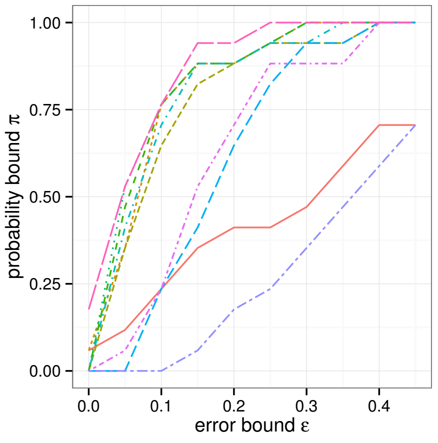


Figure 3. Isoclines for GEM benchmark. Each line represents one knob setting, which is in the feasible region for all (ϵ, π) values below the line.

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. To get a sense of the solution space, we ran the GEM benchmark, discussed in detail in Section 4, with a variety of inputs and different knob settings, and measured both the cost (running time of the program) and the quality degradation of the output of the resulting programs. Figure 1 shows the data. In this graph, each point represents a single input graph and knob settings combination; points that correspond to the same input graph are colored identically. It can be seen that even for a single input graph, there are many knob combinations that produce the same output error, and that these combinations have widely different costs. For a given input graph and output error, we are interested in minimizing cost so only the leftmost point for each such combination is of interest. Figure 2 shows these *Pareto-optimal* points for each input graph.

Since the Pareto-optimal points vary greatly across inputs, 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. This is intractable for non-trivial systems.

2.2 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. 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*
- $\sum_{j \in I} p(j) f_e(j, k_1, k_2) \quad \leq \epsilon$

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.

2.3 Controlling probability of error

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 Valiant's probably approximately correct (PAC) theory of machine learning [51], 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. For a given error bound ϵ and knob setting (k_1, k_2), consider inputs $j \in I$ that satisfy the error bound (that is, $f_e(j, k_1, k_2) \leq \epsilon$). The sum of the probabilities of these inputs is the probability that the knob setting (k_1, k_2) satisfies the error bound for a random input. This can be written formally as follows:

$$P_e(\epsilon, k_1, k_2) = \sum_{(j \in I) \wedge (f_e(j,k_1,k_2) \leq \epsilon)} p(j)$$

If this value is greater than or equal to π, then (k_1, k_2) is in the feasible region for error bound ϵ. For future reference, we call this value the *fitness* of knob setting (k_1, k_2) for error ϵ; intuitively, the greater the fitness of a knob setting, the more likely it is that it satisfies the error bound for the given ensemble of inputs.

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*
- $\sum_{(j \in I) \wedge (f_e(j,k_1,k_2) \leq \epsilon)} p(j) \geq \pi$

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

2.4 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 discussed in this paper apply to such measures but we do not consider them here for lack of space.

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 Section 3 to build models for f_c exploit *features* of the input, and for most problems, these features are the input 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, so we use N, E, and K as the features for cost estimates). In contrast, our experience is that it is difficult to find input features that are strongly correlated with output error or quality, which is why the feasible region in our formulation depends only on the values of π and ϵ, and not on features of the input. A deeper understanding of how error is related to input features might permit the development of alternative solutions.

Difficulties in solving the control problem: To gain insight into the control problem, it is useful to study the feasible region for given values of the parameters ϵ and π. Notice that this 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 3 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).

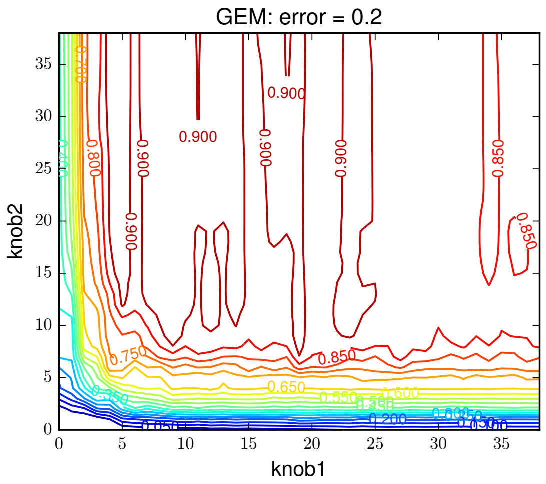


Figure 4. Probability contours for GEM. Lines represent the probability of achieving an error of 0.2.

Figure 4 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. 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.

3. 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 function.
- $f_c(i, k_1, k_2)$: cost function.

Figures 1 through 4 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 is a pictorial representation of the Capri 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). The off-line portion of the Capri 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 Capri in a extensible way to permit easy exploration of different machine learning and optimization strategies. For lack of space, we restrict the discussion to particular choices of these strategies. For f_e, we use a Bayesian network [31]; this is described in §3.1. For f_c, we consider both running time and energy, and use the tree-based model in the M5 system [34]. This cost model uses a classification of the input based on features provided by the application programmer, and is described in §3.2. We explore two solutions to the control problem in §3.4: one uses exhaustive search over the space of knob settings, which is possible when the space of knob settings is small, and the other uses the Precimonious heuristic search strategy [38], which can be used when the space of knob settings is very large. Exhaustive search permits us to evaluate the effects of *model inaccuracy* independently of *algorithmic inaccuracy* introduced by heuristic search methods.

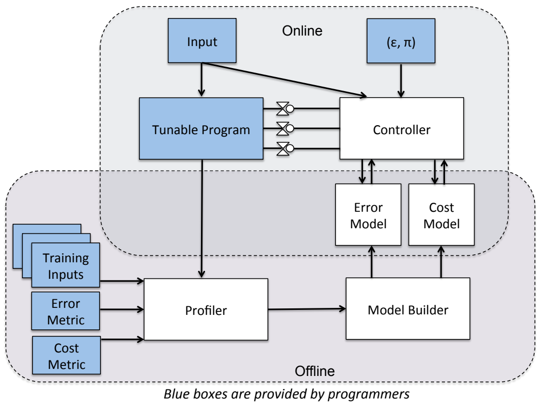
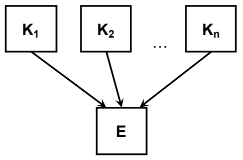


Figure 5. Overview of Capri

3.1 Error Model

An error model can be viewed abstractly as a proxy for the fitness function $P_e(\epsilon, k_1, k_2)$ defined in Section 2.3 (recall that if $P_e(\epsilon, k_1, k_2) \geq \pi$, then (k_1, k_2) is in the feasible region for a given (ϵ, π)).

In Capri , the error model is implemented using a Bayesian network [31]. A Bayesian network is a directed acyclic graph (DAG) in which each node represents a random variable in the model and each edge represents the dependence relationship between the variables corresponding to the nodes of its end points. Each node or variable may take one of many possible states and the probability 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. 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 $(k_1, k_2, ..., k_n)$.



Conditional Probability Distribution(CPD) for E

K_1	K_2	...	K_n	E	Prob
1	1	...	1	0.1	0.1
1	1	...	2	0.2	0.3
...	...	...	...	...	...

Figure 6. Bayesian Network for Error Modeling

There are many ways to model the error probability distribution using a Bayesian network. A simple model for n knobs is shown in Figure 6. In this model, the output error E depends directly on the settings of all of the knobs. Although this is simple, the size of the table for the output error is exponential in the number of knobs. This is not a problem for the applications studied in this paper, but Capri allows different error models to be plugged in easily if desired. For example, one way to reduce the size of this table is to model the errors of individual components and build a network that models the way in which these errors compose to produce the output error. This leads to more complex Bayesian networks that reflect the interconnection structure of the components of the program. We leave this for future work.

3.2 Cost Model

We model both the running time and total energy. For most algorithms, the running time can vary substantially for different inputs; after all, even for simple algorithms like matrix multiplication, the running time is a function of the input size. For complex irregular algorithms like the ones considered in this paper, running time will depend not only the input size but 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 [34], which builds tree-based models, to model the cost function f_c. Input features and knob settings define a multidimensional space; the tree model divides this space into a set of subspaces, and constructs a linear model in each subspace. The division into sub-spaces is done automatically by M5, which is a major advantage of using this system. 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 or energy per iteration does not change dramatically.

3.3 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 [31], to learn the CPD's. The user must provide the error metric for evaluating the error in the program output (§4 has several examples).

Learning the cost model The M5 model is constructed automatically from the 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 details of the learning algorithm are described in the classic M5 paper [34].

Since these models are learnt during the training phase, the time for learning these models is of secondary importance compared to ensuring that the training data sets are representative of the test inputs that will be given to the system when it is deployed.

3.4 Control Algorithms

The control algorithm must search the space of knob settings to find optimal knob settings, using the error and cost models as proxies for f_e and f_c respectively. Our system is implemented so that new search strategies can be incorporated seamlessly. We evaluated two search algorithms: *exhaustive search* and *Precimonious search* [38]. This lets us evaluate model accuracy separately from search accuracy.

- *Exhaustive search:* If the error and cost models are not expensive to evaluate and each knob has a finite number of settings, we can use exhaustive search. We sweep over the entire 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.
- *Precimonious search [38]:* In a large search space, heuristics-based search is an effective way to trade-off search cost for quality of the result. Precimonious search, which is based on the delta-debugging algorithm [54], is one such strategy. The algorithm is organized around a data structure called the *change set*; initially, this set consists of all the knobs set to the highest values, and the algorithm attempts to lower these settings iteratively as follows. In each iteration, the settings of all knobs in the change set are lowered by one level. If this new setting satisfies the accuracy constraint, the algorithm continues to the next iteration. Otherwise, the algorithm heuristically tries to find subsets of knobs whose settings can be lowered while still satisfying the accuracy constraint, and then chooses the lowest cost subset from these. Then the algorithm continue to next iteration with this subset as the new change set. Precimonious can quickly prune the search space but the solution it finds may be a local minimum.

3.5 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*, which solves Problem Formulation 3, for a given input and (ϵ, δ) values, by performing exhaustive search over the knob space and using the actual error and cost at each point in the knob space. In other words, oracle control solves the optimization problem in Problem Formulation 3 by using the actual functions f_e and f_c, instead of using proxies for these functions as Capri does. For most applications and inputs, running the application takes less time than it takes to invoke the oracle control to find optimal knob settings, so the utility of oracle control is only that it permits us to evaluate the effectiveness of the overall system in finding good knob settings.

3.6 Discussion

The approach used in Capri to controlling approximation is called *open-loop control* in classical control theory [24]. Open-loop control uses a model of the system to determine how knobs should be set for a given input, and does not consider whether the control objectives were actually met. Therefore open-loop control has no way to correct for model inaccuracy.

Closed-loop control, also known as *feedback control*, can be used when the input is a function of time; in the classical setting, the input is a continuous function of time, while in many computer applications, the input is a sequence of values, one per time step, such as a stream of video frames. In this setting, the result of controlling the system for one input can be used to adaptively control the system for succeeding inputs, as is done in the Green system [4]. Closed-loop control can be less sensitive than open-loop control to model inaccuracy, which is an advantage. However, in our setting, programs take one input and produce one output (for example, GEM takes one graph as input and produces a partitioning of the graph), so there is no obvious way to use closed-loop feedback control.

Much of the theory of closed-loop control is concerned with stability[1], and powerful techniques based on frequency-domain analysis and Lyapunov's methods have developed for proving the stability of linear and non-linear systems respectively [47]. One advantage of open-loop control is that stability is not an issue since there is no feedback.

4. Applications

We evaluated the Capri system on five complex applications: (i) GEM, the graph partitioner for social networks which was introduced earlier, (ii) Ferret [6], a content-similarity based image search engine, (iii) ApproxBullet (Bullet) [32], a 3D physics game engine, (iv) SGDSVM (SGD) [7], a library for support vector machines and (v) OpenOrd [28], a library for layout graphs. The code for these applications was modified by us to permit control of approximation. In addition, we did a blind test of the system using an unmodified radar processing application [21], written by researchers at the University of Chicago, which was already instrumented for control.

Error/Quality Definition To compute the error/quality of the output, we require the user to provide a distance function that quantifies the difference between an approximate execution and a reference execution for a given input. The reference execution can be the exact execution if such a thing exists or the best execution in the knob space for that input. The error is defined as a normalized version of this distance

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

where d, d_{max} and d_{min} represent the distance for a execution, the maximum distance and the minimum distance over the knob space for the same input. The distance function is application-specific.

[1] Roughly speaking, stability ensures that if the input is bounded, the output is bounded as well (BIBO).

4.1 GEM

As described in §1, GEM [52] is a graph clustering algorithm for social networks.

Knobs: There are two components; both use a weighted kernel k-means algorithm and have a knob controlling the number of iterations. Each knob can be set to one of 40 levels. All input graphs are partitioned into 100 clusters in our experiments.

Error metric: The output of GEM is the cluster assignment 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:

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

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 distance function computes the difference of the normalized cut given two clustering assignments. The reference execution is the execution achieving the smallest normalized cut.

Input features for modeling cost: the number of vertices in the graph, the number of edges and the number of clusters.

4.2 Ferret

Ferret [6] performs content-similarity based image search from an 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, an algorithm called Multi-probe LSH is used to approximately find $2K$ nearest neighbors of the query image. This algorithm uses multiple hash tables, 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 by an iterative optimization algorithm.

Knobs: There are three knobs in this application. In 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. EMD is computed by an iterative algorithm; executing more iterations trades 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.

Error Metric: We adopt a distance function that is widely used in the comparing search engine results [5]. Given two image lists L_1 and L_2 returned by two executions, the distance function is:

$$2\times(k-z)(k+1)+\sum_{i\in Z}|rank_1(i)-rank_2(i)| \\ -\sum_{i\in S} rank_1(i)-\sum_{i\in T} rank_2(i) \quad (1)$$

where Z is the set of images appears in both L_1 and L_2, S and T are the sets of images appearing only 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 ranks of the image i in L_1 and L_2.

The reference execution is obtained by setting all knobs to their maximum levels.

Input features for modeling cost: the number of segments in the image.

4.3 ApproxBullet(Bullet)

Bullet is a 3D physics game engine for approximately simulating the rigid body dynamics of objects. The input of Bullet is a set of objects represented by triangular meshes. Bullet simulates the behaviors of the objects in frames. In each frame, Bullet performs two major computations: approximate collision detection and sequential impulse-based constraint solving. The approximate collision algorithm is a simplified version of [32]. It first builds a multi-resolution representation for each object by coarsening its triangular mesh repeatedly [19]. 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 which is based on the 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 constraint solver, the number of iterations can trade-off the accuracy of the solution for cost. 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 up to 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 distance 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.

Input features for modeling cost: The number of triangles in each of the triangle meshes representing the objects.

4.4 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 [7] 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 distance function computes the difference of the misclassification rates between the two models resulting from the two executions. The reference execution is the one achieving the minimum misclassification rate for the same input.

Features for modeling cost: The number of training instances, the dimension of a training instance and the number of non-zero entries of the training instances.

4.5 OpenOrd

OpenOrd [28] 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 otherwise 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. The iterations are divided into five phases: liquid, expansion, cooldown, 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 to 3, 6, 6, 3 and 4 levels respectively.

Error metrics: The objective function minimized by the algorithm is used as the quality metric to measure how well the optimization problem is solved. The reference execution is the one achieving the minimum objective value.

Input features for modeling cost: The number of vertices and the number of edges in the graph.

4.6 Radar processing

We also performed a blind test of the system using a radar processing application [21] developed by Hank Hoffmann 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) is 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 beam former. The fourth knob changes the range resolution. The application can have 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.

Input features for modeling cost: No input features are used in this application.

5. Evaluation

For each benchmark, we collected a set of inputs as shown in Table 1. We would have liked to have more training inputs for GEM, OpenOrd and SGD. To evaluate the error and cost models, inputs were randomly partitioned into training and testing subsets in the proportions shown.

All experiments were run on a machine with two Xeon E5 processors and 32G memory. We used BNLearn [44] for learning Bayesian networks and gRain [23] 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 Capri system.

Benchmark	#Total	#Train	#Test	Source
Bullet	1783	881	882	[8, 50]
Ferret	3500	1750	1750	[6]
GEM	43	26	17	[25, 53]
OpenOrd	43	26	17	[25, 53]
SGD	30	18	12	[26, 48], synthetic
Radar	128	64	64	synthetic

Table 1. Inputs for benchmarks. Inputs are randomly divided into training set and testing set.

	Bullet	Ferret	GEM	OpenOrd	SGD	Radar
Training time (error)	0.239	0.927	0.109	0.119	0.120	0.072
Training time (cost)	34.44	133.37	7.52	15.00	7.18	3.80
Control (exhaustive)	0.0016	0.004	0.034	0.032	0.020	0.0009
Control (Precimonious)	0.0003	0.0010	0.0008	0.0016	0.0018	0.0006
Max execution time	2.365	2.590	584.951	221.285	388.364	608
Min execution time	0.008	0.038	0.570	2.362	0.594	550
Mean execution time	0.090	0.341	48.348	76.087	92.276	562

Table 2. Training time for error and cost (application execution time) models, running time for Control Algorithm, and Application execution times (all times are seconds)

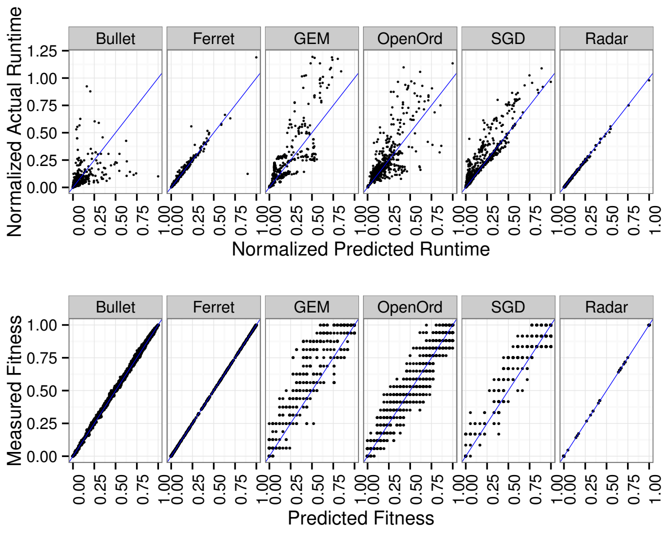


Figure 7. Accuracy of cost and error models

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

5.1 Evaluation of the cost and error models

Training is done offline, so training time is not as important as the accuracy of the cost and error models. Table 2 shows the time for training the error and cost models for the five applications (for these experiments, the cost metric was the running time of the application, and the time for running the applications on the training inputs with different knob settings is excluded). Training time obviously increases with the number of training inputs, but even for Ferret, which has

	Bullet						Ferret						GEM						OpenOrd						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	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	NA	2.0	2.4	6.3	6.3	5.9	NA	2.5	5.6	14.1	31.3	77.4
0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.4	1.4	1.6	1.7	1.9	NA	1.1	1.5	1.9	2.2	2.5	NA	2.9	6.5	6.0	5.9	8.4	NA	11.8	30.3	43.0	56.3	94.9
0.8	1.0	1.0	1.0	1.0	1.0	24.3	1.1	1.4	1.6	1.6	1.9	1.9	NA	1.2	1.7	2.1	2.4	2.6	NA	5.2	6.4	8.7	8.7	8.5	NA	39.7	52.5	103.7	165.0	184.0
0.7	1.0	1.0	1.0	1.0	9.0	96.6	1.1	1.4	1.6	1.7	1.9	2.0	NA	1.3	1.7	2.1	2.5	2.7	NA	6.3	6.1	8.5	8.8	8.5	NA	73.3	101.4	139.9	176.1	271.7
0.6	1.0	1.0	1.0	1.7	96.6	141.3	1.2	1.4	1.6	1.7	1.9	2.0	NA	1.4	2.0	2.2	2.6	2.7	NA	6.0	8.5	8.7	8.5	8.5	1.0	97.5	136.7	207.0	302.0	395.3
0.5	1.0	1.0	1.0	39.9	115.4	204.6	1.2	1.4	1.6	1.7	2.0	2.0	NA	1.7	2.3	2.5	2.7	3.0	NA	8.5	8.5	8.6	8.4	8.4	1.0	104.6	161.1	259.1	302.0	418.4

Table 3. Speedups of the tuned programs for a subset of constraint space.

the largest training set, it takes only 0.927 seconds to train the error model and 133.366 seconds (about 2 minutes) to train the cost model.

How accurate are the cost and error models constructed from the training inputs? One way to determine this is to use them to make predictions for the test inputs, and evaluate the accuracy of these predictions against empirical measurements.

Evaluating the accuracy of the cost model for a given application is straightforward: we sweep the space of test inputs and knob settings, and for each point in this space, we compare the running time predicted by the cost model with the actual execution time. The top charts in Figure 7 show the results for the applications in our test suite. In each graph, the x-axis is the predicted running time and the y-axis is the measured running time. If the cost model is perfect, all points should lie on the $y{=}x$ line. Figure 7 shows that this is more or less true for Ferret and Radar. For GEM and SGD, the predicted time is usually less than the actual execution time, and for Bullet and OpenORD, the over-predictions and under-predictions are more or less evenly distributed. Radar implements a regular algorithm in which running time depends on the size of the input. In contrast, GEM and OpenORD implement complex graph algorithms, so they are more irregular in their behavior.

Estimating the accuracy of the error model has to be more indirect since the model does not make error predictions for individual inputs but only for an ensemble of inputs. The error model is a proxy for the fitness function

$$P_e(\epsilon, k_1, k_2) = \sum_{(j \in I) \wedge (f_e(j,k_1,k_2) \leq \epsilon)} p(j)$$

This proxy is constructed during the training phase by letting I be the set of training inputs. One way to evaluate the accuracy of this proxy is to construct another proxy by letting I be the set of test inputs. If the model is accurate, these two proxy functions, which we call the predicted fitness and measured fitness, will be equal.

The bottom charts of Figure 7 show the results of this experiment. The x and y axes in each graph are the predicted and measured fitness respectively. We sweep over the space of (discretized) error values ϵ and knob settings, and for each point in this space, we evaluate the two proxy functions and plot the point in the graph. We see that the error model is very accurate for Bullet, Ferret and Radar, and less so for the other three benchmarks. For GEM and SGD, most of the points lie above the $y{=}x$ line, which means that the predicted fitness is usually less than the actual fitness. Therefore, the feasible region determined by using the model may be smaller than the actual feasible region.

It is important to note that although model accuracy can be improved in many ways, more accurate models do not necessarily enable a system like Capri to produce better solutions to the control problem. For example, the cost model is used only to rank knob settings in the feasible region, so the actual value of the predicted running time for a knob setting is not important; if the predicted running time for every knob setting is a thousand times more than the actual running time, the model is very inaccurate but it will nevertheless rank knob settings perfectly. Similarly, as long as the feasible region returned by an error model contains the optimal knob setting and does not contain spurious knob settings ranked higher by the cost model, that error model is as good as a perfectly accurate model as far as its use in Capri is concerned.

5.2 End-to-end evaluation of the control system

The control algorithm is executed online, so it is important for it to be fast. Table 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. Using the Precimonious algorithm for control is faster than using exhaustive search, as expected.

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

Table 3 shows speedups for each application for ϵ values between 0 and 0.5 and π values between 0.5 and 1.0 (for lack of space, we show only a portion of the overall constraint space). Each entry gives the average speedup over all test inputs for the knob settings found by the control algorithm based on exhaustive search, 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.

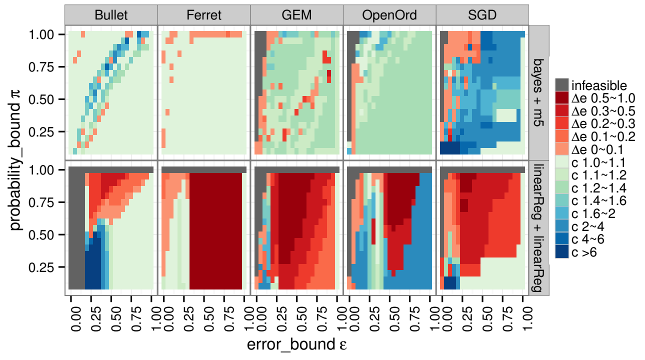


Figure 8. Effectiveness of Capri for constrained optimization of running time. Regions are colored red if error constraint is not met; shade of red encodes difference between achieved and target errors. Regions are colored green to blue if error constraint is met; shade of green-blue encodes ratio of achieved cost to cost of oracle solution, averaged over all test inputs.

.

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 tenfold 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.

Effectiveness in finding optimal knob settings While Table 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 linear regression to model both error and running time (linear regression can be seen as the simplest non-trivial model one can build for these values). The results are shown in Figure 8.

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 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 on average (over all test inputs) the cost is to the cost of the solution found by the oracle described in §3.4. The number associated with the color in each cell represents the average (over all test inputs) of the ratio of the cost of the error and cost method combination to the cost of the oracle.
- If the (ϵ, π) constraint has not actually been met, color the tile red. The shade of red indicates the difference between the achieved error and the error of the solution found by the oracle control.

In short, grey tiles indicate that the control 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 light green to light red, indicating that the error bound was met with low cost.

The top set of squares shows the results from using Capri , 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, which has a lot of blue squares. A closer study showed that the feasible region found by the Bayes error model is smaller than it should be and did not contain some low-cost points found by the oracle control. This can be attributed to the fact that the predicted fitness function for SGD is somewhat conservative, as seen in Figure 7.

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.

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%.

5.3 Performance of Radar Application

	Radar					
π || ϵ	0.0	0.1	0.2	0.3	0.4	0.5
1.0	1.0	1.0	1.0	1.1	1.1	1.1
0.9	1.0	1.0	1.0	1.1	1.1	1.1
0.8	1.0	1.0	1.0	1.1	1.1	1.1
0.7	1.0	1.0	1.0	1.1	1.1	1.1
0.6	1.0	1.0	1.0	1.3	1.3	1.3
0.5	1.0	1.0	1.0	1.3	1.3	1.3

Figure 9. Speedup of Radar Application

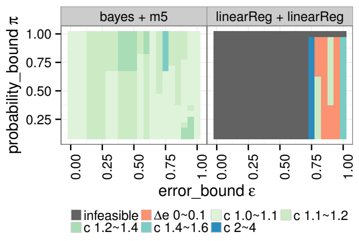


Figure 10. Effectiveness of Capri for Radar application

Table 9 and Figure 10 shows the speedups and the performance of the error and cost models for the radar application for different values of ϵ and π. These results are competitive

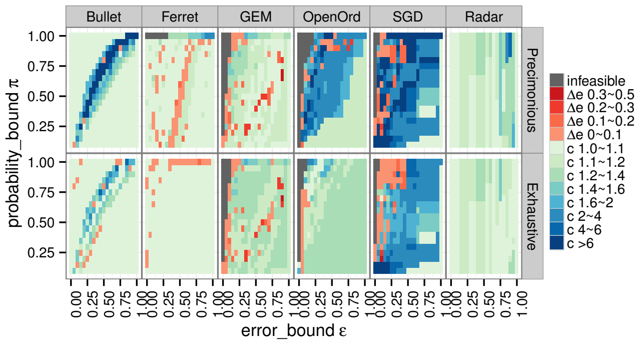


Figure 11. Precimonious search vs. exhaustive search

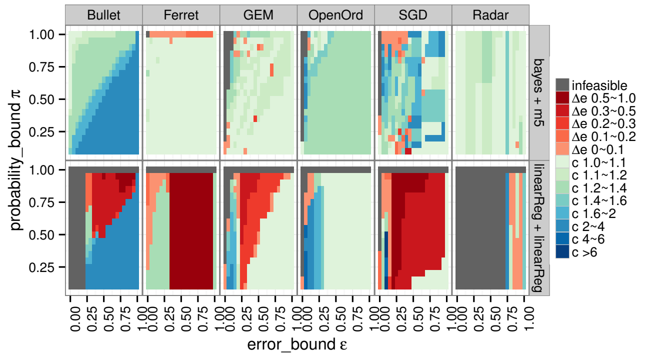


Figure 12. Effectiveness of Capri for constrained optimization of energy usage using exhaustive search

with the knob settings obtained by manual tuning for this application, which demonstrates the effectiveness of using machine learning to solve the control problem. The models using linear regression were unable to find solutions in most of the constraint space.

5.4 Using Precimonious Search

Table 2 shows that using Precimonious to solve the control problem is significantly faster than using exhaustive search. On the other hand, Figure 11 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 can 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. 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.

5.5 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 PAPI to measure the energy consumption. This machine does not support DRAM counters, so what is being measured is the CPU package energy consumption.

Table 4 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 12 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.

6. Related work

Approximation opportunities in software and hardware Loop perforation [46] explores skipping iterations during loop execution. [35] explores randomly discarding tasks in parallel applications. [36] and [9] explore relaxing synchronization in parallel applications. [49] explores different algorithmic level approximation schemes on a video summarization algorithm. In [39], 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.

A distortion model using linear regression was used in [35] to demonstrate the feasibility of their approximation techniques. The results in this paper show that linear regression is not useful for modeling quality and cost.

Researchers have proposed several hardware designs for exploiting approximate computing [15, 16, 29, 33, 42, 45]. Our techniques can be useful in choosing how to most efficiently to map programs onto such hardware and thus increase the effectiveness of such approaches.

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 [4] periodically monitors QoS values and recalibrates using heuristics whenever the QoS is lower than a specified level. PowerDial [22] leverages feed-back control theory for recalibration. Argo [18] is an autotuning system for adapting application performance to changes in multicore resources. SAGE [40] exploits this approach on GPU platforms. In [17], the authors use simulated annealing to adjust

	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	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						SGD						Radar					
$\pi \parallel \epsilon$	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	2.4	6.1	7.2	7.2	8.9	NA	21.6	59.5	83.3	108.3	107.3	1.0	1.0	1.0	1.1	1.1	1.1
0.9	NA	6.0	7.1	7.2	8.9	8.9	NA	51.0	98.0	149.2	168.7	262.7	1.0	1.0	1.0	1.1	1.1	1.1
0.8	NA	6.0	7.2	8.9	8.9	8.9	NA	91.0	192.5	266.0	265.0	319.0	1.0	1.0	1.0	1.1	1.1	1.1
0.7	NA	7.1	7.2	8.9	8.9	8.9	NA	112.7	193.6	265.0	338.2	319.0	1.0	1.0	1.0	1.1	1.1	1.1
0.6	NA	7.2	8.9	8.9	8.9	8.9	1.0	110.2	193.6	345.1	341.8	410.2	1.0	1.0	1.0	1.3	1.3	1.3
0.5	NA	8.9	8.9	8.9	8.9	8.9	1.0	129.9	254.2	345.1	420.2	410.2	1.0	1.0	1.0	1.3	1.3	1.3

Table 4. Energy savings of the tuned programs for a subset of constraint space.

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.

Auto-tuning Auto-tuning explores a space of exact implementations to optimize a cost metric like running time; in contrast, the control problem defined in this paper deals with both error and cost dimensions. [3, 14] have extended the PetaBricks [2] auto-tuning system to include an error bound. In [14], training inputs are grouped into clusters based on user-provided features, and auto-tuning is used to find optimal knob settings for each cluster for given error bounds. For a new input, optimal knob settings for the same error bounds are determined by classifying the input into one of the clusters and using the predetermined knob settings for that cluster. Auto-tuning is used by Precimonious [38] to lower precision of floating point types to improve performance for a particular accuracy constraint.

The main difference between our approach and auto-tuning approaches is that our approach builds error and cost models that can be used to control knobs for any error constraint presented during the online phase, without requiring re-training. Since auto-tuning approaches do not build models, they do not have the ability to generalize their results from the constraints they were trained for to other constraints. Note that the clustering-classification approach can be combined with our approach by clustering the training inputs and building a different model for each cluster.

Programming language support EnerJ [41] proposes a type system to separate exact and approximate data in the program. Rely [10] uses static analysis techniques to quantify the errors in programs on approximate hardware. [37] developed tools for debugging approximate programs. None of these deals with controlling the tradeoff of error versus cost.

Error Guarantees In [55], 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 [30] extends Rely [10] 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 [20] 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. [27] uses neural networks to predict whether to invoke approximate accelerators or executing the precise code for a quality constraint.

Analytic properties of programs Several techniques exist to verify whether a program is Lipschitz-continuous [12]. Smooth interpretation [11] 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.

7. Conclusions

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.

In this paper, we addressed the problem of controlling tunable approximate programs, which have one or more knobs that can be changed to vary the fidelity of the output of the approximate computation. 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. Our experimental results show that this approach performs well on controlling tunable approximate programs.

References

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

[2] 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.

[3] 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.

[4] 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.

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

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[17] 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.

[18] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. Application autotuning to support runtime adaptivity in multicorearchitectures. In *SAMOS XV*, 2015.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] Benjamin Kuo. *Automatic control systems*. Prentice-Hall Publishers, 1991.

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

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

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

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

[29] 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.

[30] 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 and Applications*, OOPSLA '14, 2014.

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

[32] 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.

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

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

[35] 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.

[36] 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.

[37] 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 Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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.

[47] Jean-Jacques Slotine and Weiping Li. *Applied Nonlinear control*. Prenctice-Hall Publishers, 1991.

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

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

[50] TF3DM. 3d graphics models. `http://tf3dm.com`.

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

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

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

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

[55] 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.