View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2014-014 June 23, 2014

Autotuning Algorithmic Choice for Input Sensitivity

Yufei Ding, Jason Ansel, Kalyan Veeramachaneni,
Xipeng Shen, Una-May O Reilly, and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Autotuning Algorithmic Choice for Input Sensitivity

Yufei Ding, Jason Ansel*, Kalyan Veeramachaneni*
Xipeng Shen, Una-May O'Reilly*, Saman Amarasinghe*

The College of William and Mary
*Massachusetts Institute of Technology
{yding,xshen}@cs.wm.edu
*{jansel,kalyan,unamay,saman}@csail.mit.edu

Abstract

Empirical autotuning is increasingly being used in many domains to achieve optimized performance in a variety of different execution environments. A daunting challenge faced by such autotuners is input sensitivity, where the best autotuned configuration may vary with different input sets.

In this paper, we propose a two level solution that: first, clusters to find input sets that are similar in input feature space; then, uses an evolutionary autotuner to build an optimized program for each of these clusters; and, finally, builds an adaptive overhead aware classifier which assigns each input to a specific input optimized program. Our approach addresses the complex trade-off between using expensive features, to accurately characterize an input, and cheaper features, which can be computed with less overhead. Experimental results show that by adapting to different inputs one can obtain up to a 3x speedup over using a single configuration for all inputs.

1. Introduction

The developers of languages and tools have invested a great deal of collective effort into extending the lifetime of software. To a large degree, this effort has succeeded. Millions of lines of code written decades ago are still being used in new programs. A typical example of this can be found in the C++ Standard Template Library (STL) routine `std::stable_sort`, distributed with the current version of GCC and whose implementation dates back to at least the 2001 SGI release of the STL. This legacy code contains a hard coded optimization, a cutoff constant of 15 between merge and insertion sort, that was designed for machines of the time, having $1/100$th the memory of modern machines. Our tests have shown that higher cutoffs (60 to 150) perform much better on current architectures. While this paradigm of write once and run it everywhere is great for productivity, a major sacrifice of these efforts is performance. Write once use everywhere often becomes write once slow everywhere. We need programs with *performance portability*, where programs can re-optimize themselves to achieve the best performance on widely different execution targets.

One of the most promising techniques to achieve performance portability is *program autotuning*. Rather than hardcoding optimizations, that only work for a single microarchitecture, or using fragile heuristics, program autotuning exposes a search space of program optimizations that can be explored automatically. Autotuning is used to search this optimization space to find the best configuration to use for each platform.

A fundamental problem faced by programs and libraries is input sensitivity. For a large class of problems, the best optimization to use depends on the input data being processed. For example, sorting an almost-sorted list can be done most efficiently with a different algorithm than one optimized for sorting random data. This problem is worse in autotuning systems, because there is a danger that the auotuner will create an algorithm specifically optimized for the inputs it is provided during training. Some existing solutions search for the preferable optimization on every training input, and then apply statistical learning algorithms to these data to build a predictive model, which maps input features to the best configurations [14, 20, 22, 27, 30].

But such solutions are not applicable to an important class of autotuning, namely *algorithmic autotuning*. The goal of algorithmic autotuning is to determine the best ways to configure an algorithm or assemble multiple algorithms together for solving a class of problems effectively. Its special challenges for addressing input sensitivity come from its four primary properties.

First, algorithmic autotuning is often sensitive to many input features that are domain-specific and require deep, possibly expensive, analysis to extract. For example, our singular value decomposition benchmark is sensitive to the number of eigenvalues in the input matrix. This is not reflected in a generic feature such as input size. A complete solution to this problem must address the fundamental trade-off between extracting a series of more expensive input features in order to be able to select a better algorithm and the cost of extracting such features overwhelming any performance

gains achieved by the better algorithm. Prior work has not addressed this problem in a systematic way.

Second, algorithmic autotuning often features a large irregular configuration space. In the benchmarks we consider, the autotuner uses algorithmic choices embedded in the program to construct polyalgorithms which process a single input through a hybrid of many different individual techniques. This results in enormous search spaces, ranging from 10^{312} to 10^{1016} possible configurations of a program. For such complex search spaces, techniques based statistical models break down because relations between input features and optimal configurations are not continuous. Even for a single input, modeling based techniques have been shown to be ineffective for these search space [3].

Finally, the case of algorithmic autotuning we consider features multiple optimization objectives. Many algorithms produce outputs of different quality, and the algorithmic autotuner is required to produce configurations that will meet a target quality of service level. For example, our Poisson's equation solver benchmark must produce an output that matches the output of a direct solver to seven digits of precision with at least 90% confidence. Meeting such a requirement is especially difficult, because the difficulty of meeting accuracy requirements can vary with each inputs. For a class of inputs, a very fast poly-algorithm may suffice to achieve seven digits of accuracy, while for different inputs that same solver may not meet the accuracy target. Prior solutions of input-adaptive autotuning have not carefully considered such a complexity.

This work presents a general means of automatically determining what algorithmic optimizations to use for each specific inputs, using selected subset of input features that must be extracted inputs. This work demonstrates that this problem can be solved through a simple *self-refining approach*.

At the center of this approach is a simple idea: In most cases, inputs to a program have some kind of affinity, meaning that many inputs share the same best configuration. So instead of finding the best configuration for every training input—which is impractical in many cases, we only need to find one best configuration for each affine class of training inputs. The difficulty is in how to find out such classes without requiring the best configuration on each input.

Our self-refining approach resolves the difficulty by letting the classes refine themselves in two levels. In the first level, it uses raw input features to cluster training inputs into some primitive classes. It then finds one configuration that fits the centroid of each class. We call these *landmark configurations*. In the second level, it refines the primitive classes by taking the performance of all training inputs on these landmark configurations as the clue—the rationale is that if two inputs are actually affine, they are likely to favor the same configuration. Based on the refined input classes, it can then use supervised statistical learning to distill the input feature sets and build up an input classifier, which makes an algorithmic choice that is sensitive to input variation while managing the input space and search space complexity.

This self-refining design leverages the observation that running a program is usually much faster than finding out the best configuration on an input (which requires a technique like autotuning). That observation makes it possible to avoid finding the best configuration for every input by furnishing evidence of how example inputs and different landmark configurations affect program performance. The design reconciles the stress between accuracy and performance by introducing a programmer-centric scheme and a coherent treatment to the dual objectives at both levels of learning. It seamlessly integrates consideration of the feature extraction overhead into the construction of input classifiers. In addition, we propose a new language keyword that allows the programmer to specify arbitrary domain-specific input features with variable sampling levels.

1.1 Contributions

This paper makes the following contributions:

- Our system is, to the best our knowledge, the first to simultaneously address the interdependent problems of variable accuracy algorithms and input sensitivity.
- Our novel self-refining approach solves the problem of input sensitivity for much larger algorithmic search spaces than would be tractable using prior techniques.
- We offer a principled understanding of the influence of program inputs on algorithmic autotuning, and the relations among the spaces of inputs, algorithmic configurations, performance, and accuracy. We identify a key disparity between input properties, configuration, and execution behavior which makes it impractical to produce a direct mapping from input properties to configurations and motivates our two level approach.
- We show through an experimentally tested model that for many types of search spaces there are rapidly diminishing returns to adding more and more input adaption to a program. A little bit of input adaptation goes a long way, while a large amount is often unnecessary.
- Experimental results show that by dynamically selecting between different optimized configurations for each input one can obtain up to a 3x speedup over using a single configuration for all inputs.

2. Language and Usage

This work is an extension to the PetaBricks language, compiler, and autotuner [3]. This section will briefly describe some of the key features of PetaBricks, and then discuss our extensions to it to support the input sensitive autotuning of algorithms.

```
1 function Sort
2 to out[n]
3 from in[n]
4 input_feature Sortedness, Duplication
5 {
6   either {
7     InsertionSort(out, in);
8   } or {
9     QuickSort(out, in);
10  } or {
11    MergeSort(out, in);
12  } or {
13    RadixSort(out, in);
14  } or {
15    BitonicSort(out, in);
16  }
17 }
18
19 function Sortedness
20 from in[n]
21 to sortedness
22 tunable double level (0.0, 1.0)
23 {
24   int sortedcount = 0;
25   int count = 0;
26   int step = (int)(level*n);
27   for(int i=0; i+step<n; i+=step) {
28     if(in[i] <= in[i+step]) {
29       // increment for correctly ordered
30       // pairs of elements
31       sortedcount += 1;
32     }
33     count += 1;
34   }
35   if(count > 0)
36     sortedness = sortedcount / (double) count;
37   else
38     sortedness = 0.0;
39 }
40
41 function Duplication
42 from in[n]
43 to duplication
44 ...
```

Figure 1. PetaBricks pseudocode for `Sort` with input features

Figure 1 shows a fragment of the PetaBricks Sort benchmark extended for input sensitivity. We will use this as a running example throughout this section.

2.1 Algorithmic Choice

The most distinctive feature of the PetaBricks language is algorithmic choice. Using algorithmic choice, a programmer can define a space of possible polyalgorithms rather than just a single algorithm from a set. There are a number of ways to specify algorithmic choices, but the most simple is the *either...or* statement shown at lines 6 through 16 of Figure 1. The semantics are that when the *ether...or* statement is executed, exactly one of the sub blocks will be executed, and the choice of which sub block to execute is left up to the autotuner.

The *either...or* primitive implies a space of possible polyalgorithms. In our example, many of the sorting routines (QuickSort, MergeSort, and RadixSort) will recursively call Sort again, thus, the *either...or* statement will be executed many times dynamically when sorting a single list. The autotuner uses evolutionary search to construct polyalgorithms which make one decision at some calls to the either...or statement, then different decisions in the recursive calls [4].

These polyalgorithms are realized through *selectors* (sometimes called *decision trees*) which efficiently select which algorithm to use at each recursive invocation of the *either...or* statement. As an example, a selector could create a polyalgorthm that first uses MergeSort to decompose a problem into lists of less than 1420 elements, then uses QuickSort to decompose those lists into lists of less than 600 elements, and finally these lists are sorted with InsertionSort.

2.2 Input Features

In this work, we have extended the PetaBricks language to support input sensitivity by adding the keyword *input_feature*, shown on lines 4 and 5 of Figure 1. The *input_feature* keyword specifies a programmer-defined function, a *feature extractor*, that will measure some domain specific property of the input to the function. A feature extractor must have no side effects, take the same inputs as the function, and output a single scalar value. The autotuner will call this function as necessary.

Feature extractors may have tunable parameters which control their behavior. For example, the `level` tunable on line 23 of Figure 1, is a value that controls the sampling rate of the sortedness feature extractor. Higher values will result in a faster, but less accurate measure of sortedness. *Tunable* is a general language keyword that specifies a variable to be set by the autotuner and two values indicating the allowable range of the tunable (in this example between 0.0 and 1.0).

Section 3 describes how input features are used by the auotuner.

2.3 Variable Accuracy

One of the key features of the PetaBricks programming language is support for variable accuracy algorithms, which can trade output accuracy for computational performance (and vice versa) depending on the needs of the programmer. Approximating ideal program outputs is a common technique used for solving computationally difficult problems, adhering to processing or timing constraints, or optimizing performance in situations where perfect precision is not necessary. Algorithmic methods for producing variable accuracy outputs include approximation algorithms, iterative methods, data resampling, and other heuristics.

At a high level, PetaBricks extends the idea of algorithmic choice to include choices between different accuracies. The programmer specifies a programmer defined *accuracy metric*, to measure the quality of the output and set an *accuracy target*. The autotuner must then consider a two dimensional objective space, where its first objective is to meet the accuracy target (with a given level of confidence) and the second objective is to maximize performance. A detailed description of the variable accuracy features of PetaBricks is given in [5].

2.4 Usage

Figure 2 describes the usage of our system for input sensitive algorithm design. At the first level, there is input aware learning which takes the user's program (containing algorithmic choices), the feature extractors specified by the `input_feature` language keyword and input exemplars as input. Input aware learning is described in Section 3. The output of the learning is an input classifier and a set of input optimized programs, each of which has been optimized for specific class of inputs.

When an arbitrary input is encountered in deployment, the classifier created by learning is used to select an input optimized program which is expected to perform the best on this input. The classifier will use some (possibly variable) subset of the feature extractors available to it to probe the input. Finally, the selected input optimized program will process the input and return the output to the user.

3. Input Aware Learning

To help motivate our self-refining approach, we will first explore existing solutions and their issues. The section will continue to explain the design and implementation of our technique.

3.1 Existing Solutions and Their Issues

A straightforward way to construct an input classifier is via input-based clustering. First, construct feature vectors for every example input set with the input_feature extraction procedures encoded by the programmer. Then, cluster examples based on the feature vectors. Next, find a good algorithmic configuration for each cluster's centroid. For a new input, the classifier first invokes the feature extraction procedures to compute its feature vector, based on which, it finds out what input cluster the new input belongs, and then runs the configuration of that cluster on that new input. This design has been used for addressing input sensitivity in program specialization and others [30]. However, applying it to algorithmic autotuning raises three issues.

First, it fails to acknowledge that two input sets that are similar may not have correspondingly similar configurations. As well, while there may be more than one configuration that suits an input set, but some will perform well on an input set similar to it, while others will not. In other words, there is no direct correspondence between similar input features, similar configurations and/or similar algorithm performance (measured in execution speed and accuracy). Instead the relationships among input properties, configurations and program behavior are non-linear and complex. We call this phenomena a *mapping disparity*. It implies that by assigning configurations based on the differences in input features, the simple design is likely to assign an inferior configuration for new input.

The second issue with the simple design is that it does not consider the overhead in feature extraction on the new input. Due to the complexity in algorithmic choice, some features may take a substantial time to extract. As the feature extraction occurs on the critical path of the program execution, the simple design may end up with a significant slowdown for the introduced extra work.

The third issue is that even if the configuration found by the simple classifier happens to provide the highest performance on that new input, its calculation accuracy may not meet the requirement. It is unclear how the simple design can handle accuracy-performance conflicts, a special complexity in algorithmic autotuning.

3.2 Self-Refining Approach

Our self-refining approach is divided into two levels. The first level is shown in Figure 3. In its first step it clusters and groups the input space into a finite number of input classes and then uses the autotuner to identify a optimized algorithmic configuration for each cluster's centroid. We call these autotuned configurations landmarks. Next, it executes every training input using every landmark configuration. This is order of magnitude faster than autotuning for every training input. These results will be used at the next level.

The second level is shown in Figure 4. It refines the primitive classes by interpreting the mapping evidence previously collected on the inputs and their performance on a small set of landmark configurations. It builds a large number of classifiers each different by which input features it references and/or different by the algorithm used to derive the classifier. It then computes an objective score, incorporating both feature extraction costs and predicted algorithm execution time, for every classifier and selects the best one as the production classifier.

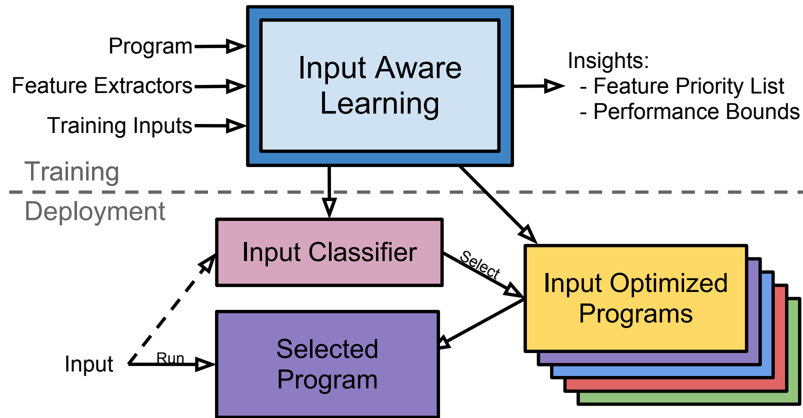


Figure 2. Usage of the system both at training time and deployment time. At deployment time the input classifier selects the input optimized program to handle each input. Input Aware Learning is described in Section 3.

Together, these two levels create an approach that is able to achieve large speedups on benchmarks sensitivity to input variation. The next two subsections will provide details on the design of each of these levels.

3.3 Level 1

The main objective of Level one is to identify a set of configurations for each class of inputs. We call these configurations "landmarks".

Specifically, there are four steps in this level of learning.

- **Step 1: Feature Extraction** We assemble a feature vector for each training input using the values computed by the *input_feature* procedures of the program. For each property, by using a tunable parameter such as `level` in the *input_feature* procedure in the program, we have collected values at z different costs which are what we call features.
- **Step 2: Input Clustering** We first normalize the input feature vectors to avoid biases imposed by the different value scales in different dimensions. We then group the inputs into a number of clusters (one hundred in our experiments) by running a standard clustering algorithm (e.g., K-means) on the feature vectors. For each cluster, we compute its centroid. Note that the programmer is only required to provide a `input_feature` functions.
- **Step 3: Landmark Creation** We autotune the program using the PetaBricks evolutionary autotuner *multiple times*, once for each input cluster, using its centroid as the presumed inputs. While the default evolutionary search in autotuner generates random inputs at each step of search, we use a flag which indicates it should use the centroid instead. We call each configuration for each cluster's centroid as input data to the program, a *landmark*. The stochasticity of the autotuner means we may get different configurations albeit perhaps equal performing ones each time.
- **Step 4: Performance Measurement** We run each landmark configuration on *every* training input and record both the execution time and accuracy (if applicable) as its performance information.

We note that there is an alternative way to accomplish Steps 1 through 3 and identify landmarks. We could find the best configuration for each training input, group these configurations based on their similarity, and use the centroids of the groups as landmark configurations. This unfortunately is infeasible because it is too time consuming to autotune (a process than can take many hours) for every training input example. Additionally, for the problem instances explored in this paper, modeling is not as effective as empirical search, and search, evolutionary in the case of PetaBricks, involves the composition and evaluation of hundreds of thousands configurations, taking hours or even days to finish [3]. This also has the problem we mention previously: similar configurations do not have matching algorithm performance. For these reasons, we cluster in the space of inputs', determine an inputs centroid for each cluster and then autotune the centroid to get a landmark. This process will cost some extra time depending the number of landmarks we want to obtain, but it is a one time only cost to programmer.

3.4 Level 2

The main objective of Level two is to refine the primitive classes produced by level 1. The design of this step stems from the observations that finding the best configuration for every input takes much longer time than directly running the problem. Besides, two inputs tend to perform well on the same configuration if they are affine. Based on these observations, level 2 rearrange the classes by taking the performance of all training inputs and these landmark configurations. And by learning the input features of the training inputs and their assignments to the configurations, a best adaptive overhead aware classifier would be identified.

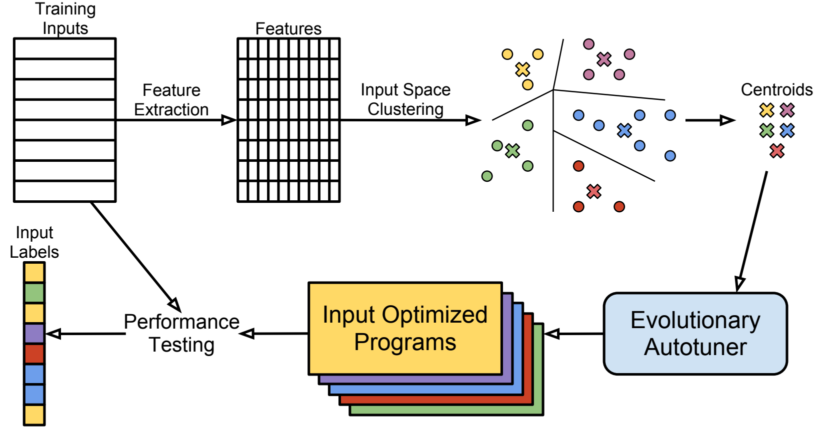


Figure 3. Selecting representative inputs to train input optimized programs.

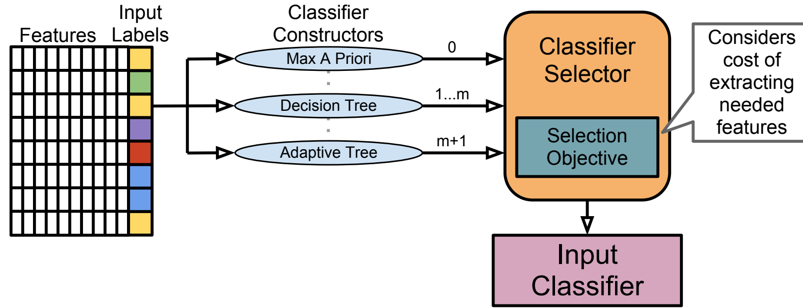


Figure 4. Constructing and selecting the input classifier.

There are two main challenges to get the best classifier: first is to determine which input features are good predictors of a high performing classifier. If this was directly known, these features could be used to learn one classifier which would directly be used in production. Because it is not, the first sub-goal is to generate a candidate set of classifiers *each with a unique set of features* and all using the data that provides evidence of the relationship between inputs, configurations and algorithm performance. The other is setting up cost function for classifier learning and computing an objective score for every classifier to identify the best candidate classifier for production, where both should be able to reconcile the stress between performance and accuracy.

3.4.1 Data Conditioning Before Classifier Learning

We use machine learning to generate our classifiers. Per machine learning, we make each set of example inputs, their features, feature extraction costs, execution times and accuracy scores for each landmark configuration, a row of a dataset. We append to each row a label which represents its new assignment. Such labels will be deployed to derive classifier together with other inputs later.

More formally, we create a datatable of 4-tuples where each 4-tuple is $< \mathbf{F}, \mathbf{T}, \mathbf{A}, \mathbf{E} >$, where $\mathbf{F}$ is a M-dimensional feature vector for this input, $\mathbf{T}$ and $\mathbf{A}$ are vectors of length $1 \times K_1$ where i^{th} entry represents the execution time and accuracy achieved for this input when i^{th} *landmark* configuration is applied, and $\mathbf{E}$ is a M-dimensional vector giving us the values for time taken for extraction of the features. We first generate labels $L \in \{1, 2, 3, \ldots, K_1\}$ for each input. Label l_i represents the best configuration for the ith input, decided by its runtime performance on all landmark configures. Depending on the number of optimization objectives, we have different labeling schemes. For problems where only minimizing the execution time is an objective (for example *sorting*) the label for i^{th} input is simply $\arg\min_j T_i^j$. While for problems where both accuracy and execution time are objectives, we first choose a threshold for the accuracy and then select the subset of configurations that meet the accuracy threshold and among them pick the one that has the minimum execution time. For the case, in which none of the configs achieve desired accuracy threshold, we pick the configuration that gives the maximum accuracy.

3.4.2 Setting up the cost matrix

Before we show the various classifiers we derive, we want to first demonstrate a preeminent component, cost matrix, for classifier learning. Because we have 100 *landmark* configurations, we have a *100-class* problem. For such *extreme-class* classification problem, it is paramount that we set up a cost matrix. The cost matrix C_{ij} represents the cost incurred by misclassifying a feature vector that is labeled i

as class j. The classifier learning algorithms uses the cost matrix to evaluate the classifier. To construct the cost matrix for this problem we followed this procedure. For every input, labeled i, we calculate the performance difference between the i landmark and the jth landmark. We assign a cost that is inverse of average performance difference for all inputs that are labeled i. This corresponds to $C^p_{i,j}$. For those benchmarks that have accuracy as a requirement, we also include a accuracy penalty matrix $C^a_{i,j}$, which is the ratio of inputs for which accuracy constraint is not met by j^{th} configuration. We construct the final cost matrix as $C_{ij} = \beta \cdot C^a_{i,j} \cdot \max_t(C^p_{i,t}) + C^p_{i,j}$. The cost is a combination of accuracy penalty and performance penalty where the former is a leading factor, while the latter acts as a tuning factor. We tried different settings for β ranging from 0.001 to 1, to get the best performance. We found 0.5 to be the best and use that.

We then divide our inputs into two sets, one set is used for training the classifier, the other for testing. We next pass the training set portion of the dataset to different classification methods which either reference different features or compute a classifier in a different way. Formally a method derives classifier C referencing a feature set $f_c \subset \mathbf{F}$ to predict the label, i.e., $C(F_i) \rightarrow L_i$.

3.4.3 Classifier Learning

We now describe the classifiers we derive and the methods we use to derive them.

Max-apriori classifier: This classifier evaluates the empirical priors for each configuration label by looking at the training data. It chooses the configuration with the maximum prior (maximum number of inputs in the training data had this label) as the configuration for the entire test data. There is no training involved in this other than counting the exemplars of each label in the training data. As long as the inputs follow the same distribution in the test data as in the training data there is minimal mismatch between its performance on training and testing. It should be noted that this classifier does not have to extract any features of the input data.

Advantages: No training complexity, no cost incurred for extraction of features.

Disadvantages: Potentially highly inaccurate, vulnerable to error in estimates of prior.

Exhaustive Feature Subsets Classifiers: Even though we have M features in the machine learning dataset, we only have $\frac{M}{z}$ input properties. The bottom level feature takes minimal time to extract and only looks at the partial input, while the top level takes significant amount of time as it looks at the entire input. However, features extracted for the same input could be highly correlated. Hence, as a logical progression, we select a subset of features size of which ranges between $1 \dots \frac{M}{z}$ where each entry is for a property. For each property we allow only one of its level to be part of the subset, and also allow it to be absent altogether. So for 4 properties with $z = 3$ levels we can generate 4^4 unique subsets. For each of these 256 subsets we then build a decision tree classifier [26] yielding a total of 256 classifiers.

The feature extraction time associated with each classifier is dependent on the subset of features used by the classifier, ie. for an input i, it is the summation $\sum_j E^j_i$. The decision tree algorithm references the label and features and tries minimize its label prediction error. It does not reference the feature extraction times, execution times or accuracy information. These will be referenced in accomplishing the second sub-goal of classifier selection.

Because we wanted to avoid any "learning to the data", we divided the data set into 10 folds and trained 10 times on different sets of nine folds while holding out the 10th fold for testing ("10 fold cross validation"). Of the 10 classifiers we obtained for each feature set, we chose the one that on average performed better.

Advantages: Feature selection could simply lead to higher accuracy and allow us to save feature extraction time.

Disadvantages: More training time, not scalable should the number of properties increase.

All features Classifier: This classifier is one of the 256 Exhaustive Feature Subsets classifiers which we call out because it uses all the M/z features at their highest level.

Advantages: Can lead to higher accuracy classification.

Disadvantages: More training time, higher feature extraction time, no feature selection.

Incremental Feature Examination classifier: Finally, we designed a classifier which works on an input in a sequential fashion. First, for every feature $f_m \in \mathbb{R}$, we divide it into multiple decision regions $\{d^m_1 \dots d^m_j\}$ where $j \geq K_1$. We then model the probability distributions under each class for each feature under each decision region $P_{m,j,k}(f_m = d^m_j | L_i = k)$. Given a pre-specified order of features it classifies in the following manner when deployed:

Step 1: Calculate the feature: Evaluate the mth feature for the input and apply the thresholds to identify the decision region it belongs to.

Step 2: Calculate posterior probabilities: The posterior for a configuration (class label) k, given all the features $\{1 \dots m\}$ acquired so far and let $d^1_1 \dots d^i_j$ be the decision regions they belong to, is given by:

$$P(L_i = k | f_{1 \dots m}) = \frac{\Pi_m P_{m,j,k}(f_m = d^m_j | L_i = k) P(L = k)}{\sum_k \Pi_m P_{m,j,k}(f_m = d^m_j | L_i = k) P(L = k)} \quad (1)$$

Step 3: Compare and decide: We pre-specify a threshold on the posterior Δ and we declare the configuration (class label) as k if its posterior $P(L_i = k | f_{1 \dots m}) > \Delta$. If none of the posteriors are greater than this threshold, we return to step 1 to acquire more features.

In this method, we incrementally acquire features for a input point i based on judgement as to whether there is enough evidence (assessed via posteriors) for them to indicate one

configuration. This results in a variable feature extraction time for different inputs thus providing potential further reductions in feature extraction time at the time the production classifier is applied to new inputs. For all the classifiers preceding this one, we extracted the same number of features for all the inputs.

This method can be applied after the previous method has found the best subset to further save on feature extraction time.

To train this classifier, we need to find the optimal decision regions for each feature and the threshold on posterior Δ such that the performance measurements mentioned above are minimized. This could be done via a simple continuous parameter search algorithm. Usually, more decision regions per feature help increase the performance and in some cases search over orders could help. This is the only classifier in our system where we introduce the domain specific cost function into the inner loop of training of a classifier.

Advantages: Reduced feature extraction time, scalable as the number of attributes increase.

Disadvantages: One possible drawback for this classifier is that it requires a storage of $i \times j \times k$ number of probabilities in a look up table. Training time.

3.4.4 Candidate Selection of Production Classifier

After we generate a set of classifiers based on different inputs or methods, we next need to select one as the production classifier. We start by applying every classifier on the test set and measuring the performance (execution time and accuracy, if required) of the algorithm when executing with its predicted configuration. We can compare this performance to that of the rest configuration. There are three objectives for the production classifier: 1) minimize execution time; 2) meet accuracy requirements ; and, 3) minimize the feature extraction time.

Let β_i be the minimum execution time for the input i by all the representative polyalgorithms. Let $\Psi(i, L_i)$ be the execution time of i when its class label is L_i, given by classifier C and $g_i = \sum_j T_j, j \in f_c$ be the feature extraction time associated with this classification.

Given a classifier we measure its efficacy for our problem as follows:

For time only:The cost incurred (represented by r_i) for classifying a data point to configuration c_i will be $r_i = \Psi(i, c_i) + g_i$. The cost function (represented by R) for all the data will be the average of all their costs, that is, $R = \sum_i (r_i)/N$, where N is the total number of data lists. We refer to R as performance cost in the following description.

For time and accuracy: Let H be the accuracy threshold, that is, only when the accuracy of the computation (e.g., binpacking) result at a data list exceeds H, the result is useful. The value of H can be prefixed by programmer.

Suppose the fraction of data lists whose computation results are inaccurate (ie. accuracy is less than H) is s when classifier C is applied to our data set. We set a target on the s. If a classifier does not meet this target, it is considered invalid (or incurring a huge cost). If a classifier meets this target then the cost of this classifier is calculated as defined above.

3.5 Discussion of the Self-Refining Approach

This two level learning has several important properties.

First, it uses a two level design to systematically address *mapping disparity*. Its first phase takes advantage of the properties of inputs to identify landmark configurations. It then furnishes evidence of how example inputs and different landmarks affect program performance (execution time and accuracy). Its second phase uses this evidence to (indirectly) learn a production classifier. By classifying based upon best landmark configuration it avoids misusing similarity of inputs. The means by which it evaluates each candidate classifier (trained to identify the best landmark) to determine the production classifier takes into account the performance of the configurations both in terms of execution time and accuracy.

Second, this two level learning reconciles the stress between accuracy and performance by introducing a programmer-centric scheme and a coherent treatment to the dual objectives at both levels of learning. The scheme allows programmers to provide two thresholds. One is an *accuracy threshold*, which determines whether the computation result is considered as accurate; the other is a *satisfaction threshold*, which determines whether the statistical accuracy guarantee (e.g., the calculation is accurate in 95% time) offered by a configuration meets the programmer's needs. The scheme is consistently followed by both levels of learning.

Third, it seamlessly integrates consideration of the feature extraction overhead into the construction of input classifiers. Expensive feature extractions may provide more accurate feature values but cost more time than cheap ones do. The key question is to select the feature extractions that can strike a good tradeoff between the usefulness of the features and the overhead. Our two level learning framework contains two approaches to finding the sweet point. One uses exhaustive feature selection, the other uses adaptive feature selection. Both strive to maximize the performance while maintaining the accuracy target.

Fourth, our learning framework maintains an open design, allowing easy integration of other types of classifiers. Any other classification algorithm could be integrated into our system without loss of generality. Plus it takes advantage of the PetaBricks autotuner to intelligently search through the vast configuration space.

4. Evaluation

To measure the efficacy of our system we tested it on a suite of 6 parallel PetaBricks benchmarks [5]. Of these benchmarks 1 requires fixed accuracy and 5 require variable accuracy. Each of these benchmarks was modified to add feature extractors for their inputs and a richer set of input generators to exercise these features. Each feature extractor was set to 3 different sampling levels providing more accurate measurements at increasing costs. Tests were conducted on a 32-core (8×4-sockets) Xeon X7550 system running GNU/Linux (Debian 6.0.6).

We use two primary baselines to provide both a lower bound of performance without input adaptation and an upper bound of the limits of input adaption. Neither baseline includes (or requires) any feature extraction costs.

- *Static oracle* uses a single configuration for all inputs. This configuration is selected by trying each input optimized program configuration and picking the one with the best performance. The static oracle is the performance that would be obtained by not using our system and instead using an autotuner without input adaptation. In practice the static oracle may be better than some offline autotuners, because such autotuners may train on non-representative sets of inputs.
- *Dynamic oracle* uses *the best* configuration for each input. It is the lower bound of the best possible performance that can be obtained by our input classifier. It is equivalent to a classifier that always picks the best optimized program and requires no features to do so. We allow the dynamic oracle to miss the accuracy target on up to 10% of the inputs, to match the selection criteria of the input classifier.

4.1 Benchmarks

We use the following 6 benchmarks to evaluate our results.

Sort The sort benchmark sorts a list of doubles using either InsertionSort, QuickSort, MergeSort, or BitonicSort. The merge sort has a variable number of ways and has choices to construct a parallel merge sort polyalgorithm. Sort is the only non-variable accuracy benchmark shown. Input variability comes from different algorithms having fast and slow inputs, for example QuickSort has pathological input cases and InsertionSort is good on mostly-sorted lists. For input features we use standard deviation, duplication, sortedness, and the performance of a test sort on a subsequence of the list.

Sort1 results are sorting real-world inputs taken from the Central Contractor Registration (CCR) FOIA Extract, which lists all government contractors available under FOIA from data.gov. *Sort2* results are sorting synthetic inputs generated from a collection of input generators meant to span the space of features.

Clustering The clustering benchmark assigns points in 2D coordinate space to clusters. It uses a variant of the kmeans algorithm with choices of either random, prefix, or centerplus initial conditions. The number of clusters (k) and the number of iterations of the kmeans algorithm are both set by the autotuner. The accuracy metric compares the sum of distances squared to cluster centers to a canonical clustering algorithm. Clustering uses input the features: radius, centers, density, and range.

Clustering1 results are clustering real-world inputs taken from the Poker Hand Data Set from UCI machine learning repository. *Clustering2* results are clustering synthetic inputs generated from a collection of input generators meant to span the space of features.

Bin Packing Bin packing is a classic NP-hard problem where the goal of the algorithm is to find an assignment of items to unit sized bins such that the number of bins used is minimized, no bin is above capacity, and all items are assigned to a bin. The bin packing benchmark contains choices for 13 individual approximation algorithms: AlmostWorstFit, AlmostWorstFitDecreasing, BestFit, BestFitDecreasing, FirstFit, FirstFitDecreasing, LastFit, LastFitDecreasing, ModifiedFirstFitDecreasing, NextFit, NextFitDecreasing, WorstFit, and WorstFitDecreasing. Bin packing contains 4 input feature extractors: average, standard deviation, value range, and sortedness.

Singular Value Decomposition The SVD benchmark attempts to approximate a matrix using less space through Singular Value Decomposition (SVD). For any $m \times n$ real matrix A with $m \geq n$, the SVD of A is $A = U\Sigma V^T$. The columns u_i of the matrix U, the columns v_i of V, and the diagonal values σ_i of Σ (singular values) form the best rank-k approximation of A, given by $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$. Only the first k columns of U and V and the first k singular values σ_i need to be stored to reconstruct the matrix approximately. The choices for the benchmark include varying the number of eigenvalues used and changing the techniques used to find these eigenvalues. The accuracy metric used is the ratio between the RMS error of the initial guess (the zero matrix) to the RMS error of the output compared with the input matrix A, converted to log-scale. For input features we used range, the standard deviation of the input, and a count of zeros in the input.

Poisson 2D The 2D Poisson's equation is an elliptic partial differential equation that describes heat transfer, electrostatics, fluid dynamics, and various other engineering disciplines. The choices in this benchmark are multigrid, where cycle shapes are determined by the autotuner, and a number of iterative and direct solvers. As an accuracy metric, we used the ratio between the root mean squared (RMS) error of the initial guess fed into the algorithm and the RMS error of the guess afterwards. For input features we used the residual

Benchmark Name	Dynamic Oracle	Classifier (w/o feature extraction)	Classifier (w/ feature extraction)
sort1	5.104×	2.946×	2.905×
sort2	6.622×	3.054×	3.016×
clustering1	3.696×	2.378×	2.370×
clustering2	1.674×	1.446×	1.180×
binpacking	1.094×	1.093×	1.081×
svd	1.164×	1.108×	1.105×
poisson2d	1.121×	1.086×	1.086×
helmholtz3d	1.111×	1.046×	1.044×

Figure 5. Mean speedup over the static oracle of the generated input classifier (with and without feature extraction costs included) and the dynamic oracle, using 100 landmark configurations. Static oracle uses the best *single* configuration for all inputs, and is the best performance achievable without using input adaption. Dynamic oracle uses the best configuration for each input, and is the upper bound speedup one would get with a "perfect" input classifier.

measure of the input, the standard deviation of the input, and a count of zeros in the input.

Helmholtz 3D The variable coefficient 3D Helmholtz equation is a partial differential equation that describes physical systems that vary through time and space. Examples of its use are in the modeling of vibration, combustion, wave propagation, and climate simulation. The choices in this benchmark are multigrid, where cycle shapes are determined by the autotuner, and a number of iterative and direct solvers. As an accuracy metric, we used the ratio between the root mean squared (RMS) error of the initial guess fed into the algorithm and the RMS error of the guess afterwards. For input features we used the residual measure of the input, the standard deviation of the input, and a count of zeros in the input.

4.2 Experimental Results

Figure 5 shows the overall performance of our system on an isolated testing data set. Overall results range 1.046x speedup for helmholtz3d, to 3.054x speedup for sorting. Both of these results are close to the dynamic oracle performance of 1.111x and 6.622x for these same two benchmarks. Different speedups across benchmarks are caused by the diversity and distribution of supplied inputs, as well as program's differing sensitivity to input variations. Applications whose performance vary more across inputs have more room for speedup by adapting to specific inputs. This is confirmed by the performance of the dynamic oracle, which is the upper bound of speedup from adapting to different inputs with our technique. Generally, the less expensive features (in terms of feature extraction time) were sufficient to meet the best performance. Additionally, we note that most of the features we extracted had orders of magnitude smaller extraction time when compared to the execution time.

In the sort benchmark, we also tried both real world inputs (sort1) and inputs from our own generator (sort2). For real world input, the best classifier used the sorted list and sortedness features at its intermediate sampling level and the duplication and deviation at the cheapest level. 2.946x speedup was achieved compared to a dynamic oracle speedup of 5.104x. For inputs from our own generator, the best classifier used the sorted list and sortedness features at its intermediate sampling level, achieving our largest speedup of 3.054x compared to a dynamic oracle of 6.622x.

In the clustering benchmark, we tried real world inputs and those from our own generator. For real world input, the best classifier used the density feature at its cheapest level, and algorithms selected by the classifier causes a 2.378x shorter execution time (compared to the dynamic oracle speedup of 3.696x), For our own generator, the best classifier used the centers feature at its cheapest sampling level, achieving a 1.446x speedup compared to a dynamic oracle of 1.674x. However, centers feature is the most expensive feature relative to execution time, which lowers the effective speedup to just 1.180x.

In the binpacking benchmark, the best classifier used the deviation and sortedness features at the intermediate level. The classifier is selecting algorithms that cause a 1.093x faster execution time (close to the dynamic oracle speedup of 1.094x).

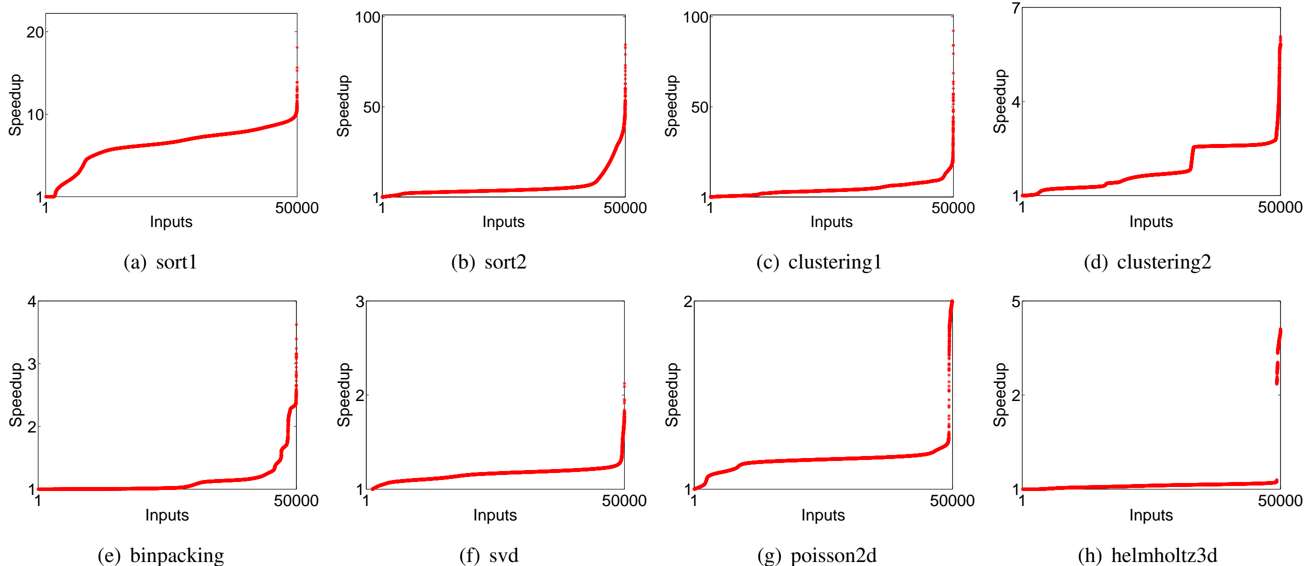
In the svd benchmark, the best classifier used only zeros input feature at the intermediate level and achieved 1.108x speedup compared to a dynamic oracle of 1.164x. It is known that svd is sensitive to the number of eigenvalues, but this feature is expensive to measure. The features we included are cheaper and tend to reflect that factor indirectly. For instance, it is likely, although not necessarily, that a matrix with many 0s has fewer eigenvalues than a matrix with only a few 0s.

In the poisson2d benchmark, the best classifier employed the input features zeros at the intermediate level, achieving a 1.086x speedup compared to a dynamic oracle of 1.121x.

In the helmholtz3d benchmark, the best classifier used the residue, zeros and deviation input features at the intermediate level and the range feature at the cheapest level. This benchmark showed a 1.046x speedup, compared to a dynamic oracle speedup of 1.111x.

4.3 Input Generation and Distribution

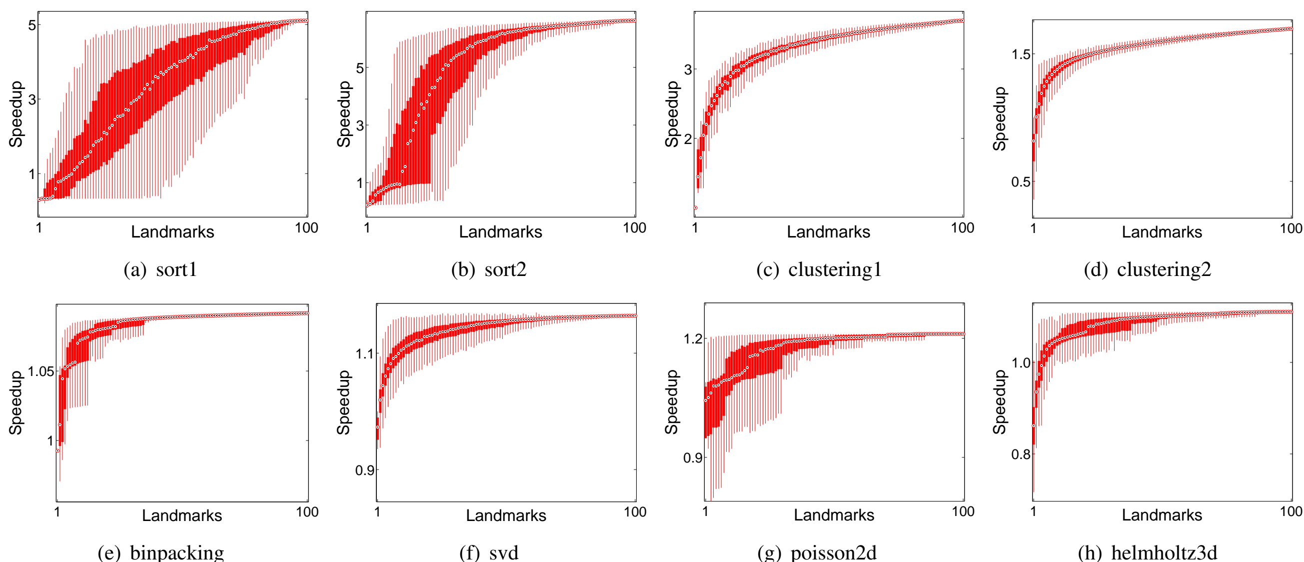
For *sort1* and *clustering1* we used real world input data taken from production systems. The performance for this real world data can be compared to *sort2*, *clustering2*, and other benchmarks where we used synthetic input generators designed to span the feature space. For sort, real world inputs provides similar mean speedup to synthetic inputs. For clustering, real world inputs saw much larger speedups than synthetic inputs. Interestingly, the classifier for synthetic in-



(a) sort1 (b) sort2 (c) clustering1 (d) clustering2

(e) binpacking (f) svd (g) poisson2d (h) helmholtz3d

Figure 6. Distribution of speedups over static oracle for each individual input. For each problem, some individual inputs get much larger speedups than the mean.



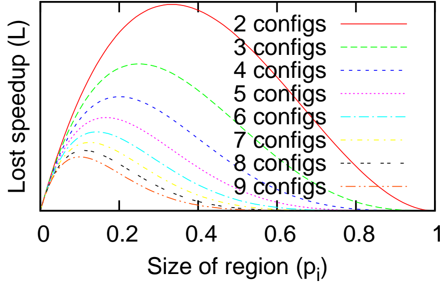
(a) sort1 (b) sort2 (c) clustering1 (d) clustering2

(e) binpacking (f) svd (g) poisson2d (h) helmholtz3d

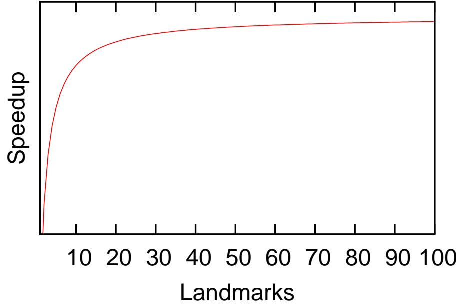
Figure 8. Measured speedup over static oracle as the number of landmark configurations changes, using 1000 random subsets of the 100 landmarks used in other results. Error bars show median, first quartiles, third quartiles, min, and max.

puts in clustering needs to use a much more expensive set of input features because the classes of inputs are harder to distinguish.

To have a better idea of the origin of different speedup, not only among benchmarks, but also for the same benchmark, but different input sources. We further investigate the distribution of speedups for individual inputs to each program, sorted such that the largest speedup is on the right, showed in Figure 6. What is interesting here is the speedups are not uniform. For each benchmark there exist small sets of inputs with very large speedups, in some cases up to $90x$. This shows that way inputs are chosen can have a large effect on mean speedup observed. If one had a real world input distribution that favored these types of inputs the overall speedup of this technique would be much larger. In other words, the relative benefits of using of input adaptation techniques can vary drastically depending on your input data distribution.



(a) Predicted loss in speedup contributed by input space regions of different sizes.



(b) Predicted speedup with a worst-case region size with different numbers of sampled landmark configurations.

Figure 7. Model predicted speedup compared to sampling all input points as the number of landmarks are increated. Y-axis units are omitted because the problem-specific scaling term in the model is unknown.

4.4 Theoretical Model Showing Diminishing Returns with More Landmark Configurations

In addition to the evaluation of our classifier performance it is important to evaluate if our methodology of clustering and using 100 landmark configurations is sufficient. To help gain insight into this question we created a theoretical model where we consider the input search space of a program where some finite number of optimal program configurations dominate different subsets of the input space. For each of these dominate configurations, we define the values p_i and s_i, where p_i is fraction of the inputs in the search space where this configuration dominates and s_i is the speedup on these configurations obtained by training a configuration for any of the inputs where this configuration dominates. The model assumes that no speedup is obtained if one of these points is not sampled. We also assume that all inputs have equal cost before the speedups are applied, to avoid the need for weighting terms.

If we assume the k landmark configurations are sampled uniform randomly (which is likely a worse technique than our actual clustering) the total expected loss in speedup, L, compared to a perfect method that sampled all points would be:

$$L = \sum_i (1 - p_i)^k p_i s_i$$

Where $(1 - p_i)^k$ represents the chance of "missing" the region of the search space where configuration i is optimal and $p_i s_i$ represents the cost of missing that region of the search space in terms of speedup.

Figure 7(a) shows the value of this function for a single region as p_i changes. One can see that on the extremes $p_i = 0$ and $p_i = 1$ there is no loss in speedup, because either the region is so small a speedup in that region does not matter or the region is so large that random sampling is likely to find it. For each number of configs, there exists a worst-case region size where the expected loss in speedup is maximized. We can find this worst-case region size by solving for p_i in $\frac{dL}{dp_i} = 0$ which results in a worst-case $p_i = \frac{1}{k+1}$. Using this worst-case region size, Figure 7(b) shows the diminishing returns predicted by our model as more landmark configurations are sampled. Figure 8 validates this theoretical model by running each benchmark with varying numbers of landmark configurations. This experiment takes random subsets of the 100 landmarks used in other results and measures that speedup over the static oracle. Real benchmarks show a similar trend of diminishing returns with more landmarks that is predicted by our model. We believe that this is strong evidence that using a fixed number of landmark configurations (e.g., 10 to 30 for the benchmarks we tested) suffices in practice, however correct number of landmarks needed may vary between benchmarks.

5. Related Work

A number of studies have considered program inputs in library constructions [9, 12, 19, 24, 29, 35]. They concentrate on some specific library functions (e.g., FFT, sorting) while the algorithmic choices in these studies are limited. Tian and others have proposed an input-centric framework [30] for dynamic program optimizations and showed the benefits in enhancing Just-In-Time compilation. Jung and others have considered inputs when selecting the appropriate data structures to use [20]. Several recent studies have explored the influence of program inputs on GPU program optimizations [22, 27].

This current study is unique in focusing on input sensitivity to complex algorithmic autotuning, which features vast algorithmic configuration spaces, sensitivity to deep input features, variable accuracy, and complex relations between inputs and configurations. These special challenges prompt the development of the novel solutions described in this paper.

A number of offline empirical autotuning frameworks have been developed for building efficient, portable libraries in specific domains. ATLAS [34] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LA-

PACK. FFTW [13] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [18] for sparse matrix computations, SPIRAL [25] for digital signal processing, and OSKI [33] for sparse matrix kernels.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [1, 2, 14, 23]. These projects change both the order that compiler passes are applied and the types of passes that are applied. PetaBricks [3] offers a language support to better leverage the power of autotuning for complex algorithmic choices. However, none of these projects have systematically explored the influence of program inputs beyond data size or dimension.

In the dynamic autotuning space, there have been a number of systems developed [7, 8, 10, 16, 17, 21, 28] that focus on creating applications that can monitor and automatically tune themselves to optimize a particular objective. Many of these systems employ a control systems based autotuner that operates on a linear model of the application being tuned. For example, PowerDial [17] converts static configuration parameters that already exist in a program into dynamic knobs that can be tuned at runtime, with the goal of trading QoS guarantees for meeting performance and power usage goals. The system uses an offline learning stage to construct a linear model of the choice configuration space which can be subsequently tuned using a linear control system. The system employs the heartbeat framework [15] to provide feedback to the control system. A similar technique is employed in [16], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance. The principle theme of these studies is to react to dynamic changes in the system behavior rather than proactively adapt algorithm configurations based on the characteristics of program inputs.

Additionally, there has been a large amount of work [6, 11, 31, 32] in the dynamic optimization space, where information available at runtime is used combined with static compilation techniques to generate higher performing code. Such dynamic optimizations differ from dynamic autotuning because each of the optimizations is hand crafted in a way that makes it likely lead to an improvement of performance when applied. Conversely, autotuning searches the space of many available program variations without a priori knowledge of which configurations will perform better.

6. Conclusions

We have shown a self-refining solution to the problem of input sensitivity in autotuning that, first, clusters to find input sets that are similar in the multi-dimensional property space and uses an evolutionary autotuner to build an optimized program each of these clusters, and then builds an adaptive overhead aware classifier which assigns each input to a specific input optimized program. This provides a general means of automatically determining what algorithmic optimization to use when different optimization strategies suit different inputs. Though this work, we are able to extract principles for understanding the performance and accuracy space of a program across a variety of inputs, and achieve speedups of up to 3x.

While at first input sensitivity seems to be excessively complicated issue where one must deal with large optimization spaces and complex input spaces, we show that input sensitivity can be handled with simple extensions to an existing autotuning system. We also showed that there are fundamental diminishing returns as more and more input adaptation is added to a system and that a little bit of input adaption can go a long way.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization*, pages 295–305, 2006.

[2] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES'04*, pages 231–239, 2004.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *PLDI*, Dublin, Ireland, Jun 2009.

[4] J. Ansel, M. Pacula, S. Amarasinghe, and U.-M. O'Reilly. An efficient evolutionary algorithm for solving bottom up problems. In *Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, July 2011.

[5] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for autotuning variable-accuracy algorithms. In *CGO*, Chamonix, France, Apr 2011.

[6] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *PLDI*, 1996.

[7] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, June 2010.

[8] V. Bhat, M. Parashar, . Hua Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed. Enabling self-managing applications using model-based online control strategies. In *International Conference on Autonomic Computing*, Washington, DC, 2006.

[9] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*, pages 340–347, 1997.

[10] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4, March 2001.

[11] P. C. Diniz and M. C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *PLDI*, New York, NY, 1997.

[12] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[13] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *IEEE*, 93(2), February 2005. Invited paper, special issue on "Program Generation, Optimization, and Platform Adaptation".

[14] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, Jul 2008.

[15] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*, New York, NY, 2010.

[16] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2209-042, Massachusetts Institute of Technology, Sep 2009.

[17] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Power-aware computing with dynamic knobs. In *ASPLOS*, 2011.

[18] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *International Conference on Computational Science*, 2001.

[19] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[20] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[21] G. Karsai, A. Ledeczi, J. Sztipanovits, G. Peceli, G. Simon, and T. Kovacshazy. An approach to self-adaptive software based on supervisory control. In *International Workshop in Self-adaptive software*, 2001.

[22] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.

[23] E. Park, L.-N. Pouche, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 119 –129, April 2011.

[24] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[25] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1), 2004.

[26] J. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[27] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, 2012.

[28] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.

[29] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2005.

[30] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[31] M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, 2000.

[32] M. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. *ACM SIGPLAN Notices*, 36(7), 2001.

[33] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing Conference*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[34] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing*, Washington, DC, 1998.

[35] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.