Autotuning Algorithmic Choice for Input Sensitivity

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

∗: North Carolina State University, Raleigh, NC, USA
◇: Massachusetts Institute of Technology, Cambridge, MA, USA
∗: {yding8, xshen5}@ncsu.edu
◇:{jansel, kalyan, unamay, saman}@csail.mit.edu

Abstract

A daunting challenge faced by program performance autotuning is input sensitivity, where the best autotuned configuration may vary with different input sets. This paper presents a novel two-level input learning algorithm to tackle the challenge for an important class of autotuning problems, algorithmic autotuning. The new approach uses a two-level input clustering method to automatically refine input grouping, feature selection, and classifier construction. Its design solves a series of open issues that are particularly essential to algorithmic autotuning, including the enormous optimization space, complex influence by deep input features, high cost in feature extraction, and variable accuracy of algorithmic choices. Experimental results show that the new solution yields up to a 3x speedup over using a single configuration for all inputs, and a 34x speedup over a traditional one-level method for addressing input sensitivity in program optimizations.

Categories and Subject Descriptors D3.2 [*Programming Languages*]: Language Classifications – Concurrent, distributed, and parallel languages; D3.4 [*Programming Languages*]: Processors - Compilers.

General Terms Algorithms, Languages, Performance.

Keywords Petabricks, Autotuning, Algorithmic Optimization, Input Adaptive, Input Sensitivity, Two-level Input Learning.

1. Introduction

Application developers have invested a great deal of 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 in an era when machines had $1/100$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 hard-coding 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 in autotuning 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 that contain many repeated values can be done most efficiently with a different algorithm than one optimized for sorting random data. For many problems, no single optimized program exists which can match the performance of a collection of optimized programs autotuned for different subsets of the input space. A common solution is to search for good optimizations on every training input, based on which, it builds a machine learning model that predicts the best optimization to use in a new run according to the features of the new input [22, 30, 33, 39, 42].

Although the exhaustive approach works for certain program optimizations, it is insufficient for *algorithmic autotuning*, a class of autotuning problems, in which, the autotuners try to determine the best ways to configure an algorithm or assemble multiple algorithms into a recursive poly-algorithm to solve a problem [6]. Properties of algorithmic autotuning determines that it is often infeasible to find good optimizations—or algorithmic configuration in this context—for every training input. On one hand, in the benchmarks we consider, the autotuner uses algorithmic choices embedded in the program to construct arbitrary polyalgorithms. The polyalgorithms process a single input through a hybrid of many different individual techniques, which results in enormous search spaces, ranging from 10^{312} to 10^{1016} possible configurations of a programm [6]. Even through intelligent search algorithms [7], finding a desirable configuration at a training input takes hours or days. On the other hand, algorithmic choices are often sensitive to 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. Hundreds or thousands of training inputs are

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2015 held by Owner/Author. Publication Rights Licensed to ACM.
PLDI'15 , June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06…$15.00
DOI: http://dx.doi.org/10.1145/

often prepared such that the complex input feature space could get well covered. The large number of inputs, combined with the long training time per input, means searching for good configurations for every training input may take months or years for a program, which makes the traditional exhaustive approach impractical for algorithmic autotuning.

An intuitive solution is through clustering [11, 33, 39]. It first clusters training inputs into groups with the members in the same group featuring similar values of predefined input features, and then finds the configuration that works the best for the centroid of each group. In a production run, when a new input comes, the program extracts its feature values, based on which, it identifies the cluster that the input resembles the most, and then runs with that cluster's configuration. We call this approach the *one-level learning approach*. It has been used for intelligent program compilations and other autotuning tasks. It avoids finding the best configuration for every training input and hence saves much training time.

The one-level approach however works inferiorly for algorithmic autotuning, resulting in as much as 29x slowdown over using a single configuration for all inputs on our benchmarks (detailed in Section 4). The reason comes from three properties of algorithmic autotuning.

- *Costly Features.* Due to the nature of algorithmic selection, algorithmic autotuning often involves extracting complex, and sometimes costly, input features such as matrix eigenvalues. Extraction of too many such features can add a large runtime overhead to production runs.
- *Mapping Disparity.* The relations between input features and algorithmic configurations are often complex. Two inputs having similar values in the predefined features may respond very differently to the same algorithmic configuration. Standard unsupervised feature selection (e.g., PCA) [23] does not solve the problem (more in Section 4), while standard supervised methods (e.g., decision trees) [23] cannot apply since the best configurations for most training inputs are unknown. The disparity between the predefined feature space and the best configurations causes inferior configurations to be selected by the one-level method.
- *Variable Accuracy.* Unlike many compiler optimizations, algorithmic autotuning has close interactions with variable accuracies. A program with different algorithmic configurations often produces outputs of different quality, and the autotuner is required to produce configurations that will meet a user specified quality of service level. For example, a Poisson's equation solver may require its output to match the output of a direct solver to a certain precision with at least 95% confidence. Meeting such a requirement is difficult especially because of the input variations. For a class of inputs, a very fast polyalgorithm may suffice to achieve the accuracy, while for different inputs that same solver may not meet the accuracy target. Without carefully treating the dual objectives of accuracy and performance, the one-level method frequently gives configurations that miss the accuracy target, as Section 4 shows.

This paper presents a new solution that addresses all those challenges through a two-level learning framework. It starts with the traditional one-level learning, and employs a second-level input learning to close the mapping disparity gap between input features and performance space, identify cost-effective input features, and reconcile the stress between accuracy and performance. Specifically, in the second level, after getting the configuration (called a *landmark*) for the centroid of each input cluster, it regroups the training inputs by drawing on the performance of all the landmark configurations on each of the training inputs, based on which, it selects input features and builds up appropriate input classifiers with algorithm performance and computation accuracy simultaneously considered.

This design leverages the observation that running a program once is usually much faster than autotuning the program, which can involve thousands of runs, to find a suitable algorithmic configuration. Its performance-based auto-refinement of input clusters closes the mapping disparity gap between the clustering results and performance, without the need for finding the best configuration for every input. 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. This two-level approach is able to achieve large speedups by making an optimization choice which is sensitive to input variation while managing the input space and search space complexity. In addition, we propose a new language keyword that allows the programmer to specify arbitrary domain-specific input features with variable sampling levels.

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 the sensitivity of algorithmic selection on deep, complex input features.
- Our novel two-level auto-refinement approach solves the problem of input sensitivity for much larger, more complex algorithmic search spaces than would be tractable using state-of-the-art 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 algorithmic configurations that a program can select across inputs. Cross-input adaptation on a few algorithmic configurations goes a long way, while a large number of configurations are often unnecessary.
- Experimental results show that the two-level solution produces up to a 3x speedup over using a single configuration for all inputs in algorithmic selection, and a 34X speedup over prior one-level method.

2. Language and Usage

This work is developed upon the PetaBricks language and its compiler [6]. This section will briefly describe some of the key features of PetaBricks, and then discuss our extensions to it to support the exploration of input sensitive autotuning of algorithms.

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 *either...or* statement is executed, exactly one of the sub blocks will

```
 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

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 [8].

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. Figure 2 shows an example: The selector creates 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.

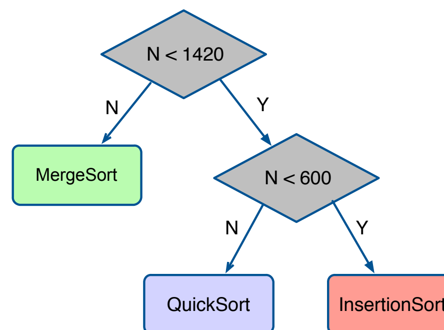


Figure 2: A selector that realizes a polyalgorithm for sorting.

2.2 Input Features

To support exploration of input sensitivity, in this work we introduce a new keyword *input_feature* into the PetaBricks language, 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 automatically set by the autotuner, and two values indicating the allowable range of the tunable (in this example between 0.0 and 1.0).

Users may use *input_feature* keyword to define all the potentially relevant features based on her domain knowledge; some of the features could be more costly or less important than others. Effectively selecting the subset of features that are critical and beneficial is part of the key support offered by our two-level approach. Suitable values of the tunable parameters are also part of the algorithmic configurations that our enhanced autotuner is designed to help determine adaptively across program inputs.

After autotuner selects the right subset of input features and their tunable parameter values, every execution of the program of interest will call the corresponding feature extraction functions at the beginning of the execution. The collected input features will be used to select the configuration of the algorithm that best suits the current input. Section 3 will elaborate on how input features are used by the auotuner.

2.3 Variable Accuracy

One of the key features of the PetaBricks programming language is its support for variable accuracy algorithms, which can trade output accuracy for computational performance (and vice versa) depending on the quality of service requirements or program specification. 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 sets an *accuracy target*. The autotuner must then consider a two dimensional objective space, where its

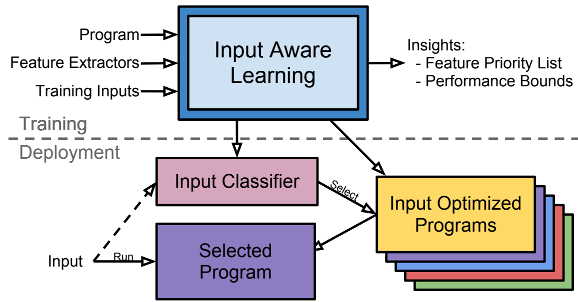


Figure 3: Usage of the system both at training time and deployment time. At deployment time the input classifier selects the input optimized program to handle each input. Input Aware Learning is described in Section 3.

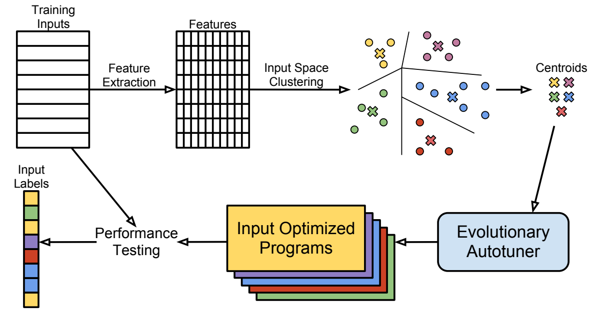


Figure 4: Selecting representative inputs to train input optimized programs.

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 3 describes the usage of our system for input sensitive algorithm design. The input aware learning, described in Section 3, takes the user's program (containing algorithmic choices), the feature extractors specified by the `input_feature` language keyword and input exemplars as input. The output of the learning is an input classifier, a subset of input features, 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 the subset of the feature extractors to probe the input. Finally, the selected input optimized program will process the input and return the output.

3. Input Aware Learning

Motivated by the particular complexities of algorithmic autotuning, we develop a two-level learning framework. The first level is shown in Figure 4. 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 good algorithmic configuration for each cluster's centroid. In addition however, to provide data on the mappings among inputs, configuration and performance, it executes every exemplar (i.e., training input) using the configuration of each cluster. These results will be used at the next level.

The second level is shown in Figure 5. It first refines the input clustering results by drawing on the previously collected evidence on the inputs and their performance on a small set of "landmark" configurations. It then builds a large number of classifiers each different by which input features it references and/or the algorithm used to derive the classifier. It then computes an objective score for every classifier and selects the best one as the production classifier. Together, these two levels create an approach that is able to achieve large speedups by its sensitivity to input variation and configuration influence on program performance. We provide detail on each level.

3.1 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 (100 in our experiments; sensitivity of the number discussed in Section 4) 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 an `input_feature` function.
- **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.

Another simple way to select the landmarks is based on random selection. However, the quality of the landmarks selected is worse than that obtained with clustering algorithm based on input features, especially when the number of landmarks (i.e., samples) is small. For example, with 5 configurations, uniformly picked landmarks result in 41% degradation of performance than selection with kmeans. As the number of configurations increases, the gap shrinks. But because finding the configuration on one sample takes a long time, clustering algorithms (e.g., K-means) have their advantages.

Besides, 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 find a suitable configuration for every input example. Modeling is not as effective as 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 [6]. This also has the problem we mention previously: similar configurations do not have matching algorithm

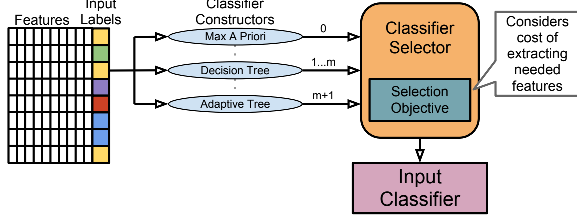


Figure 5: Constructing and selecting the input classifier.

performance. For these reasons, we cluster in the space of inputs, and then use the second-level learning to automatically refine the clusters.

3.2 Level 2

The main objective of this level is to identify a production classifier that is able to efficiently identify the best landmark configuration to use for a new input. The challenge is to determine which input features are good predictors of a high performing classifier. Because it is unknown, 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. Incorporating this step and its wealth of evidence drastically improves results. The second sub-goal follows: choose among the candidates to identify the best one for production. Through the following discussion, we assume that there are u input feature extractors with each having z possible levels of collection cost; the whole feature set is hence as large as $M = u * z$. We use K_1 to represent the number of landmarks.

Cluster Refinement before Classifier Learning An important step before achieving both sub-goals is to regroup the training inputs to reflect their performance on the landmark configurations collected in the first level. In this step, 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 is the best configuration for the input.

More formally, we create a datatable of 4-tuples. Each 4-tuple is $< \mathbf{F}, \mathbf{T}, \mathbf{A}, \mathbf{E} >$, where $\mathbf{F}$ is an M-dimensional feature vector for this input, $\mathbf{T}$ and $\mathbf{A}$ are vectors of length $1 \times K_1$ where the ith entry represents the execution time and accuracy achieved for this input when the ith *landmark* configuration is applied. $\mathbf{E}$ is an M-dimensional vector giving us the values for time taken for extraction of the features. We first generate labels $L \in \{1 \dots K_1\}$ for each input. Label l_i represents the best configuration for the ith input. For problems where only minimizing the execution time is an objective (for example *sorting*) the label for the ith input is simply $\arg\min_j T_i^j$. For problems where both accuracy and execution time are objectives, we use a user-provided threshold (per the requirement of the particular problem and usage context) 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. These labels regroup the inputs into K_1 clusters that directly reflect the performance of various configurations on those inputs.

This step and the clustering in the first level together forms our two-level clustering strategy. By enabling an auto-refinement of input clusters, the strategy helps close the mapping disparity gap between the predefined input features and the performance space that the one-level learning approach suffers from.

Setting Up the Cost Matrix Because we have K_1 *landmark* configurations, we have a K_1-*class* problem; K_1=100 in our experiments. 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 use the cost matrix to evaluate the classifiers. To construct the cost matrix for this problem we followed this procedure. For every input, labeled i, we calculate the performance difference between the ith landmark and the jth landmark. We assign a cost that is the 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., $C(F_i) \rightarrow L_i$.

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

(1) 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.

(2) 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 [38] 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 $\sum_j E_i^j$. The decision tree algorithm references the label and features and tries to 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 increases.

(3) All features Classifier: This classifier is one of the 256 Exhaustive Feature Subsets classifiers which we call out because it uses all the u unique features.

Advantages: Sometimes lead to higher accuracy classification.

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

(4) 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 $\{d_1^m \dots d_j^m\}$ where $j \geq K_1$. We then model the probability distributions under each class for each feature under each decision region $P_{m,j,k}(f_m = d_j^m | L_i = k)$. 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 $d_1^1 \dots d_j^i$ be the decision regions they belong to, is given by:

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

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 an 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.

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 other configurations. 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: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 inputs. We refer to R as performance cost in the following description.

For time and accuracy: For the case of variable accuracy, the efficacy is calculated with two thresholds the user defines according to the requirements of the particular problem and usage context. The first, H_1, is the accuracy threshold, that is, only when the accuracy of the computation (e.g., binpacking) result at a data list exceeds H_1, the result is useful. The second, H_2, is the satisfaction threshold. Suppose the fraction of inputs whose computation results are inaccurate (ie. accuracy is less than H) is s when classifier C is applied to our data set. If $s < H_2$, the classifier is considered invalid (or incurring a huge cost). If the classifier meets this target, the performance cost of this classifier is calculated as defined above.

3.3 Discussion of the Two Level Learning

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. For variable-accuracy algorithms, absolute accuracy guarantees on arbitrary inputs are difficult to have (just like in most software testing or machine learning tasks). The scheme we use 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 former indicates the desired accuracy target and the latter indicates the desired proportion of inputs that hit that target. The accuracy thresholds are benchmark-specific, given by the authors or users of each benchmark. For example, for a Bin Packing benchmark, the accuracy can be set as the average of the occupied fractions of all bins; for a clustering benchmark, it is the difference in the average point-to-center distances between the clustering result from the benchmark and the result from a canonical

clustering algorithm. The scheme is consistently followed by both levels of learning. The two-level method offers a certain kind of statistical guarantee of the accuracy.

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 may not be amortized by the time savings brought by them. 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. Moreover, 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]. These benchmarks are up to thousands of lines long and have search spaces with 10^{312} to 10^{1016} possible configurations. Of these benchmarks, five 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). There are 50,000 to 60,000 inputs per program. Around half of them are used for training and the half for testing. All reported results are based on cross validation, which ensures the exclusion of training inputs from the testing inputs.

Section 4.1 describes the definition of the accuracy metric for each benchmark along with the accuracy thresholds used in the experiments. We set the satisfaction threshold to 95% for all experiments, which means that the accuracy threshold should be satisfied on at least 95% of the inputs.

Three primary baselines are used for comparison.

- *Static Oracle.* It 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 (and meeting the satisfying accuracy threshold when applicable). 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.
- *One Level.* It is the aforementioned traditional one-level learning approach. It clusters inputs based on the predefined features. It fails to treat the overhead of extracting features and the disparity between input feature and performance space, and is oblivious to variable accuracy.
- *Dynamic Oracle.* It uses *the best* configuration for each input. Due to the size of the search space, an oracle that tries all possible configurations is not possible. The dynamic oracle we use represents the best that is possible in the second level given the landmarks available. It gives the upper bound of the possible performance that can be obtained by our input classifier (and reaches the accuracy satisfaction threshold). It is equivalent to a classifier that always picks the best optimized program (given the landmarks available) and requires no features to do so.

4.1 Benchmarks

We use the following 6 benchmarks to evaluate our results.

Sort This benchmark sorts a list of doubles using 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 [1]. *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 is $\sum \hat{d}_i / \sum d_i$, where, $\hat{d}_i$ and d_i are the distance between a point and its cluster center in a canonical clustering algorithm and in our experiment respectively. The accuracy threshold is set to 0.8. 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 [2]. *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. Its accuracy metric is defined as the average of the occupied fractions of all bins. The accuracy threshold is set to 0.95.

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 logrithm of the ratio of the root mean squared (RMS) error of the initial guess (the zero matrix) to the RMS error of the output. The RMS errors are relative to the input matrix A. The accuracy threshold is set to 0.7. 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 logrithm of the ratio of the RMS error of the initial guess to the RMS error of the guess afterwards. The RMS errors are relative to the accurate solution of the equation. The accuracy threshold is set to 7. 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.

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. The accuracy metric is the same as used in *Poisson 2D*. The accuracy threshold is also set to 7. 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

Table 1 reports the overall performance. In this section, all speedup numbers use the performance of static oracle as the baseline unless noted otherwise. Results from all methods exceed the accuracy satisfaction threshold except for the one-level method, the satisfaction rates of which are reported in the rightmost column of Table 1. Compared to the performance by the static oracle, the one-level approach performs better on only two out of the eight tests (with one of them missing the accuracy target). On the other tests, the slowdown by that approach is as much as 29 times. The comparison between columns 5 and 6 in Table 1 indicates that the large feature extraction cost is one of the reasons for the poor performance. But even without considering that cost, column 5 shows that the one-level method still brings much less speedups on sort1 and sort2, and much lower accuracy on all other benchmarks, compared to what the two-level method brings. The reason is its poor classifier construction for the lack of consideration of the space disparity. Note that during the search for the best configuration for a landmark in the 1-level method, accuracy is considered. The problem is that despite that the configuration works well for the landmark input, it does not work well on other inputs that belong to the same cluster as the landmark, because of the poor clustering results by the 1-level method, which causes the poor accuracy.

On the other hand, the second level clustering and the follow-up classifier construction in the two-level method help find the refined input feature set that better correlates with the performance of configurations on inputs. The second-level clustering is critical. Take Kmeans as an example, 73.4% of the data points changed their clusters when the second-level clustering is applied. The better classifiers, plus better input features it selects, make the two-level method significantly outperform the one-level method.

Table 1 shows that the one-level method, when its overhead in input feature collection is not counted, gives a higher speedup than the two-level method on clustering1. Unlike the one-level approach, the two-level approach considers not only performance but also cost and the computation accuracy of resulting program. On clustering1, it uses a subset of input features on which it could provide good performance and satisfying accuracy (>95%), while the one-level approach uses all input features, which, although it gives higher performance when cost is not counted, has an inferior overall performance and a poor accuracy (54.56%).

The dynamic oracle results in Table 1 shows a spectrum of potential among the tests, reflecting the different sensitivity of the problems to input variations. Applications whose performance vary more across inputs have more room for speedup by adapting to specific inputs. Correspondingly, the speedups by the two level approach show a range as well, from 1.04x to over 3x. The speedups are close to what the dynamic oracle produces except on the sort benchmark, which may require some more advanced feature extractors to be provided.

Feature extraction time barely changes the speedups by the two-level method, indicating that only a small amount of time was spent on feature extraction. Generally, the less expensive features (in terms of feature extraction time) were sufficient for the two level approach to meet the best performance. The low extraction cost of the selected features largely obviated the need for the adaptive classifier in the current scenario.

In the sort benchmark, we 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, and achieved 2.95x speedup. For inputs from our own generator, the best classifier used the sorted list and sortedness features at its intermediate sampling level, achieving speedup of 3.05x.

In the clustering benchmark, we also 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 2.38x shorter execution time. For our own generator, the best classifier used the centers feature at its cheapest sampling level, achieving a 1.45x speedup. However, centers feature is the most expensive feature relative to execution time, which lowers the effective speedup to just 1.18x.

In the binpacking benchmark, the best classifier used the deviation and sortedness features at the intermediate level. The feature extraction time is negligible compared to the sorting time, and hence it gives a speedup of 1.08x, almost the same as that by the dynamic oracle.

In the svd benchmark, the best classifier used only zeros input feature at the intermediate level and achieved 1.11x speedup compared to a dynamic oracle of 1.16x. 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 1.09x speedup compared to a dynamic oracle of 1.12x.

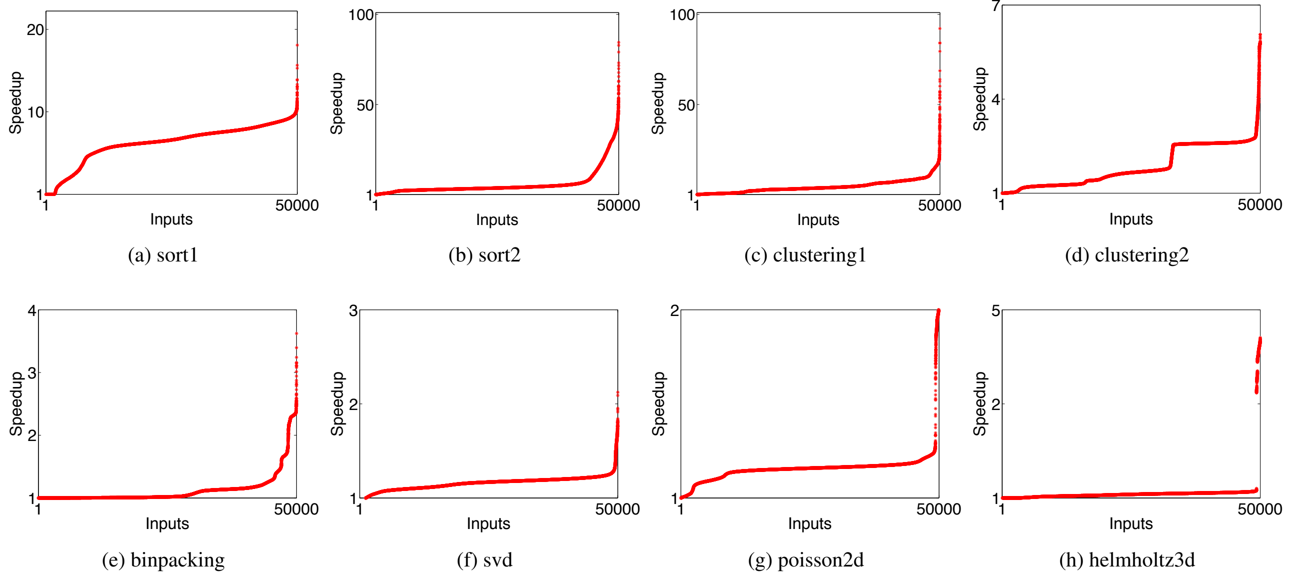
In the helmholtz3d benchmark, the best classifier used the residue, zeros and deviation input features at the intermediate level and the range feature at the cheapest level. This benchmark showed a 1.05x speedup, compared to a dynamic oracle speedup of 1.11x.

Figure 6 shows the distribution of speedups for individual inputs to each program, sorted such that the largest speedup is on the right. 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 the 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.

Training Time The training time of the two-level method is dominated by the time for the autotuner to find the 100 landmark configurations. The time ranges from 2 hours to 2.7 days for our benchmarks. An exhaustive method that finds the best configuration for every input would take over 200 times longer, given the 20,000 to 30,000 training inputs a benchmark has in our experiments.

Table 1: Mean speedup (slowdown if less than 1) over the performance by the static oracle. The rightmost column reports the percentage of inputs on which the one-level approach reaches accuracy threshold; the percentages by all other methods are greater than 95%, the satisfaction threshold.

Benchmark Name	**Dynamic Oracle**	**Two-level (w/o feature extr. time)**	**Two-level (w/ feature extr. time)**	**One-level (w/o feature extr. time)**	**One-level (w feature extr. time)**	**One-level accuracy**
sort1	5.10×	2.95×	2.91×	1.23×	1.14×	-
sort2	6.62×	3.05×	3.01×	0.22×	0.22×	-
clustering1	3.70×	2.38×	2.37×	3.09×	0.095×	54.56%
clustering2	1.67×	1.45×	1.18×	1.15×	0.034×	77.03%
binpacking	1.10×	1.09×	1.08×	1.07×	0.99×	97.83%
svd	1.16×	1.11×	1.11×	1.04×	1.03×	72.85%
poisson2d	1.12×	1.09×	1.09×	0.89×	0.84×	64.32%
helmholtz3d	1.11×	1.05×	1.04×	0.85×	0.84×	96.97%



(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.

4.3 Theoretical Model Showing Diminishing Returns with More Landmark Configurations

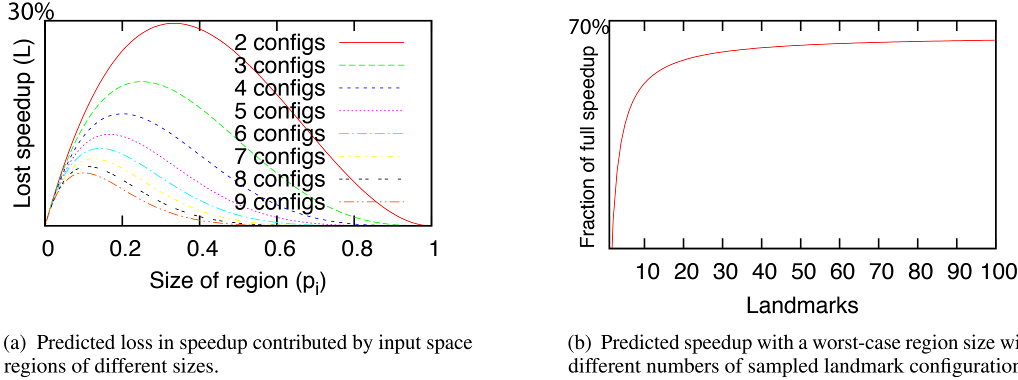
In addition to the evaluation of our classifier performance, it is important to evaluate if our methodology of clustering and using 100 landmark configurations is sufficient. To help gain insight into this question we created a theoretical model. In the model, we consider that in the input search space of a program, some number of optimal program configurations dominate different regions of the input space. Assume inputs uniformly distributed in the space. Let p_i be the size of the region i, and s_i be the speedup the program gets if the dominate configuration in region i is used for an input in that region. Assume that no speedups are obtained if that configuration is not used for that input.

If we assume the k landmark configurations are sampled uniform randomly, the chance for sampling to miss region i is $(1-p_i)^k$. So, the total expected loss in speedup, L, compared to a perfect method that has taken all dominate configurations would be:

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

Where $p_i s_i$ represents the cost of missing that region of the search space in terms of speedup.

Figure 7a shows the value of this function as p_i changes. It is assumed that all s_i is the same. 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 7b 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



(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 increased. Y-axis units are omitted because the problem-specific scaling term in the model is unknown.

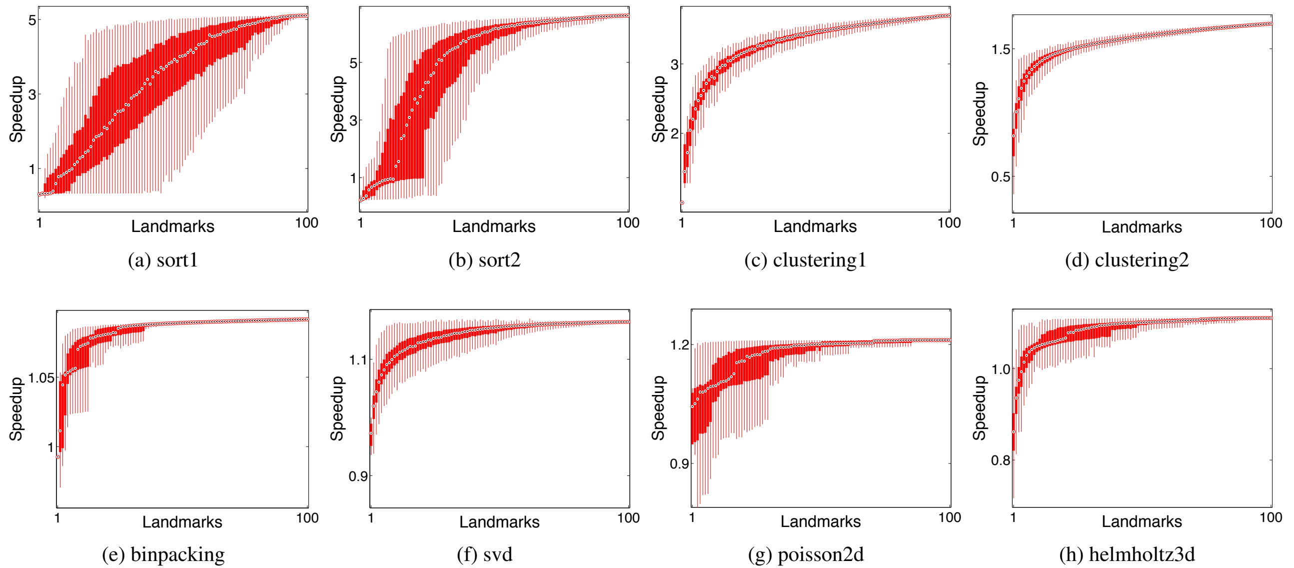


Figure 8: Measured speedup over static oracle as the number of landmark configurations changes, using 1000 random subsets of the 100 landmarks used in other results. Error bars show median, first quartiles, third quartiles, min, and max.

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 [13, 19, 29, 32, 36, 41, 47]. They concentrate on some specific library functions (e.g., FFT, sorting) while the algorithmic choices in these studies are limited. Input-centric program optimization [42] showed the benefits in enhancing Just-In-Time compilation. Jung and others have considered inputs when selecting the appropriate data structures to use [30]. Several recent studies have explored the influence of program inputs on GPU program optimizations [33, 39]. Chen and others have evaluated input sensitivity of iterative optimization [16]. Muralidharan and others have recently built a programmer-directed autotuning framework named Nitro for selecting the appropriate code variants at runtime [34].

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 [46] utilizes empirical autotuning to produce a cache-contained matrix multiply, which is then used in larger matrix computations in BLAS and LAPACK. FFTW [20] uses empirical autotuning to combine solvers for FFTs. Other autotuning systems include SPARSITY [28] for sparse matrix computations, SPIRAL [37] for digital signal processing, and OSKI [45] for sparse matrix kernels.

The area of iterative compilation contains many projects that use different machine learning techniques to optimize lower level compiler optimizations [3, 4, 22, 35]. These projects change both the order that compiler passes are applied and the types of passes that are applied. PetaBricks [6] 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 [10, 12, 14, 15, 24, 26, 27, 31, 40] 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 [27] 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 [25] to provide feedback to the control system. A similar technique is employed in [26], where a simpler heuristic-based controller dynamically adjusts the degree of loop perforation performed on a target application to trade QoS for performance. Fursin and others have explored run-time evaluation of program optimizations with phase-sensitivity [21]. 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 [9, 17, 43, 44] in the dynamic optimization space, where information available at runtime is used combined with static compilation techniques to generate higher performing code. Some work [18] has studied a library-based approach to gain performance portability across heterogeneous systems on chips while guaranteeing particular accuracy levels.

6. Conclusions

We have shown a two level solution to the problem of input sensitivity in algorithmic autotuning. It provides a general means of automatically determining what algorithmic optimization to use when different optimization strategies suit different inputs. Through 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 over using a single configuration for all inputs, and a 34x speedup over a traditional one-level method. We also showed that there are fundamental diminishing returns as more input adaptation is added to a system, and that a little bit of input adaption goes a long way.

Acknowledgement This material is based upon work supported by DoE awards DE-SC0005288, DE-SC0008923 and Early Career Award and the National Science Foundation (NSF) under Grant No. 1464216, No. 1320796 and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DoE or NSF.

References

[1] Government's open data. `http://www.data.org/`.

[2] UCI data sets. `http://archive.ics.uci.edu/ml/datasets`.

[3] 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.

[4] 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.

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

[6] 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.

[7] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of The 23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[8] 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.

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

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

[11] P. Berube, J. Amaral, R. Ho, and R. Silvera. Workload reduction for multi-input profile-directed optimization. In *Proceedings of the IEEE / ACM International Symposium on Code Generation and Optimization*, 2009.

[12] 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.

[13] 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.

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

[15] Y. Chen, S. Fang, L. Eeckhout, O. Temam, and C. Wu. Iterative optimization for the data center. In *ASPLOS*, 2012.

[16] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI'10, pages 448–459, 2010.

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

[18] S. Fang, Z. Du, Y. Fang, Y. Huang, Y. Chen, L. Eeckhout, O. Temam, H. Li, Y. Chen, and C. Wu. Performance portability across heterogeneous socs using a generalized library-based approach. *ACM Transactions on Architecture and Code Optimization*, 11, 2014.

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

[20] 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".

[21] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Transactions on High-Performance Embedded Architectures and Compilers*, 4050:34–53, 2007.

[22] 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.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[24] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2012.

[25] 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.

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] X. Li, M. J. Garzarán, and D. Padua. Optimizing sorting with genetic algorithms. In *CGO*, 2005.

[33] 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.

[34] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A framework for adaptive code variant tuning. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 501–512. IEEE, 2014.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

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

[47] 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.