Modeling Relations Between Inputs and Dynamic Behavior for General Programs

Xipeng Shen Feng Mao

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{xshen,fmao}@cs.wm.edu

Abstract. Program dynamic optimization, for being adaptive to runtime behavior changes, has become increasingly important for both performance and energy savings. However, most runtime optimizations often suffer from the lack of a global picture of a program's execution, and cannot afford sophisticated program analysis. On the other hand, offline profiling techniques overcome both obstacles but are oblivious to the effects of program inputs.
An approach in the between is to offline find the connections between program inputs and runtime behavior, and then apply the knowledge to runtime optimizations. Although it potentially gets the best of both worlds, it faces a fundamental challenge: How to discover and model the relations between inputs and runtime behavior for general programs.
This work tackles the problem from three aspects. It proposes an eXtensible Input Characterization Language (XICL) to resolve the complexity of program inputs. With XICL translator, a raw input can be automatically converted to an attribute vector, which is then refined by a feature selector to remove redundancies and noises. Finally, statistical learning builds input-behavior models. Experiments on IBM XL compilers show accurate prediction of detailed execution profiles, helping profile-directed compilation outperform both static and offline profiling-based compilations, demonstrating the potential of the technique for continuous program optimizations.

1 Introduction

Program optimizations have evolved from static compilation to dynamic transformation, reflected by the increasingly growing interest in Just-In-Time compilers, continuous optimization techniques, and runtime compiling systems [3, 4, 7, 13, 14, 21]. During a program's execution, dynamic systems transform its code and data to better match the runtime behavior.

Most dynamic systems make optimization decisions upon the information collected through runtime sampling. The low tolerance of overhead makes it difficult if not impossible to capitalize on sophisticated program analysis or to discover important but costly behaviors (e.g. data access patterns). Moreover, most dynamic systems are reactive: they make decisions upon the execution just observed. An underlying assumption is that the program will behave the same as how it behaved. This assumption is fragile for program phases and runtime environment changes (e.g. garbage collections and interferences from other programs), and may mislead the optimizations.

On the other hand, profiling-directed compilation optimizes a program based on some profiling runs. Being an offline technique, the compilation can afford more analysis overhead. However, it is oblivious to the changes in program inputs, which often resulting in inferior optimization decisions since the real runs of a program may differ substantially from the profiling runs due to the difference of their inputs [1].

A tradeoff in the between is to offline build an input-behavior model connecting program inputs with its behavior. At the beginning of a real run, it predicts the runtime behavior by plugging the program input into the input-behavior model, and then optimizes the program accordingly. The technique combines the strengths of the both worlds: By moving behavior data collection and analysis offline, it is more affordable to sophisticated behavior analysis than pure runtime techniques; by modeling input-behavior relations, it handles input-sensitive behavior better than profiling-directed compilations.

In order to apply input-behavior models to general programs, we have to address two issues: How to deal with the complexity of program inputs, and how to build the connections between inputs and runtime behaviors. Although there has been some work in utilizing program inputs for specific optimization decisions, e.g. the selection of algorithms for sequential sorting by Li et al. [16] and for parallel sorting and matrix multiplication algorithms by Thomas et al. [20], we are not aware of any systematic explorations in solving these two issues for *general* programs.

In this work, we develop three techniques to build input-behavior models. The first is an eXtensible Input Characterization Language (XICL) for the formal expression of program inputs. The inputs to general programs can be as simple as an integer or as complex as an entire program with many options (e.g. inputs to a compiler). Furthermore, sometimes what matters to runtime behavior are the hidden attributes instead of literal value of an input. For instance, it is the size or content rather than the name of a file that determines the behavior of a file-compression. XICL is a mini-language for programmers to formally describe the format of program inputs and express a superset of the critical input attributes. After a programmer builds an input specification for a program using XICL, an interpretor can automatically translate any input of the program into an input vector with necessary attribute values. XICL makes input characterization tractable for general programs.

The second technique is input feature selection. From the input vector generated by XICL translator, this step finds the elements in the vector that are critical to the runtime behavior of interest. One input argument can have many attributes; some may be remotely relevant to the behavior, and some may strongly correlate with others. Feature selection can save the collection of redundant information, improving the generality of the constructed input-behavior model.

The third step is the construction of input-behavior models through statistical learning techniques. We explore various regression techniques and employ cross-validation to alleviate the overfitting problem. This step also marks unpredictable behaviors so that the runtime system can avoid poor predictions to those behaviors.

We apply the technique to 3 SPEC CINT2000 programs and a data mining program. The input-behavior models predict their detailed profiles with 90-99% accuracy. On a IBM Power4 machine, the predicted profiles help IBM XLC compiler outperform both

static and pure offline profiling-based compilations. It demonstrates the potential of the technique for continuous program optimizations.

In the rest of this paper, we present XICL in Section 3, feature selection in Section 4, model construction in Section 5, and evaluation in Section 6. In Section 7, we discuss the issues on training inputs and the ways to hide training process. Section 8 describes related work, followed by a short summary.

2 Program Behavior Model

Program dynamic behaviors fall into two categories depending on whether they are hardware-related or not. The goal of this work is to develop a technique for systematic detection and prediction of the effects of *inputs* on program behavior. Therefore, we concentrate on hardware-unrelated behaviors to avoid the distractions from other factors. But the techniques can be extended to hardware-related behaviors as well.

Program code and inputs are the only factors determining hardware-unrelated behaviors. A general formula is $B = g(P, I)$, where B is a hardware-unrelated behavior, P is a program, I is the program's input, and g represents the mapping function. In the study of input effects for a given program, P is constant and I is the only changing factor; thus we fold P into the mapping function and rewrite the formula as $B = f(I)$, where f is the mapping function from input I to behavior B.

Constructing an input-behavior model is to determine function f. With such a model, plugging any input into f will generate the predicted behavior of the program's execution on that input even without starting the real execution. One option to build the model is through symbolic analysis on program code, but it faces the difficulty of pointer analysis, alias analysis, and uncertainty of runtime behavior.

Our approach is to formalize the task as a statistical learning problem, and use regression analysis to solve it. By feeding a program with different inputs, $I_1, I_2, \ldots, I_N$, we observe the corresponding behavior of the program's executions, represented by $B_1, B_2, \ldots, B_N$. The input-behavior pairs, $< I_i, B_i > (i = 1, 2, \ldots, N)$, compose a training set. We use linear regression and regression trees to derive the approximation of f from the training set (Section 5.)

The complexity of program inputs is a major obstacle to the regression analysis. Developing a scalable technique to convert raw inputs to a clean form with critical input attributes contained is vital for input-behavior modeling.

3 Input Formal Expression

This section explains how XICL resolves input complexity. As illustrated by Figure 1, a typical program command-line in Unix/Linux consists of four components: command, options, option arguments, and operands. The last three form the content of a raw program input. In this work, we assume program inputs are given at the starting of the program's execution. Interactive programs can be addressed by incremental model building, a topic left for our future study.

Raw program inputs are not suitable for input-behavior analysis for four reasons. First, in many cases it is the attributes (e.g. the size of an input file) instead of the literal

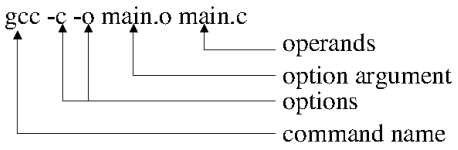

Fig. 1. Command-line Components

values (e.g. the name of an input file) of a command-line component that determine a program's behavior. But it is not always possible to automatically determine what a raw input represents (e.g. a text file, a socket, or a graph) and what attributes should be included. Furthermore, some attributes are domain-specific and require programmers' knowledge. A typical example is the initial ordering of the input data for sorting [16,20].

Second, input-behavior analysis requires the separation of qualitative attributes from quantitative ones. Unlike quantitative attributes (e.g. file size), qualitative attributes such as file types are categorical. Regression analysis treats them differently. Directly from raw inputs, it sometimes is difficult to separate them apart. For instance, the optimization level of a compiler is a qualitative attribute, even though it is an integer number.

The third reason is the relations among input components. Two arguments in an input can refer to the same option and one may overshadow the other. Raw inputs do not contain such information. Furthermore, finding the corresponding components between two raw inputs can be difficult if not knowing the format of the command line.

The last reason is that raw inputs don't contain the default values of input options. Although many programs allow a number of options, a command-line often has only several of them explicitly indicated, leaving other options carrying their default values. Without uncovering those values, most of the training data for the regression analysis would be incomplete, resulting in poor accuracy.

All of these suggest the necessity for a scheme which can deal with the complexity and convert raw inputs to a cleaner form with important input attributes contained; XICL is our solution.

3.1 Extensible Input Characterization Language (XICL)

XICL is a mini-language for programmers to formally describe the format and the potentially important attributes of a program's inputs. In order to enable the automatic translation of raw inputs to a well-structured format, the programmer of an application need write an input specification using XICL. The specification describes all the options and operands accepted by the program, in a format that a XICL translator can use to determine the role of each component in an arbitrary legal command-line and consequently convert the command-line to an input vector containing necessary attributes.

XICL Constructs For the purpose of clarity, we will use the example contained in Figure 2 to show the use of XICL. The program is to find the shortest routes in a given graph. It allows three options: "-e", "–echo", and "-n". The first two options are equivalent to each other, determining whether intermediate results will be printed or not. The

last option determines the number of shortest paths to find. Figure 2 (b) and (c) contain the input specification and an attribute-deriving procedure written by a programmer.

```
SYNOPSIS: route [options] FILE
DESCRIPTION: A program to find the shortest path in a graph. FILE has node number at the
             beginning followed by the graph structure.
OPTIONS:
   -e, --echo: print intermediate results. It is off by default.
   -n NUM:    find NUM shortest paths.  Default is 1.
```

(a) Usage of program *route*

```
option{ name = -e:--echo; has_arg = N; type = BIN; attr = VAL; default = 0; }
option{ name = -n; has_arg = Y; type = NUM;  attr = VAL; default = 1; }
operand{ position = 1:1; type = FILE; attr = FSTN1:mEDGS; }
```

(b) Input specification in XICL. VAL and FSTN1 are predefined attributes, representing the value of an option and the first number in a file, while mEDGS is defined by a programmer as shown in (c).

```
ATTS * mEDGS (char * f) {
   n = readNumOfEdges(f);
   storeTo(n, patts->pAtt[0].value);
    patts->pAtt[0].isQuan = true;
    return patts;
}
```

(c) An attribute-deriving procedure

Example command-line:
route -n 3 graph1
where, graph1 contains 100 nodes
and 1000 edges.

Input vector produced by XICL translator:
(0, 3, 100, 1000)

(d) An example input and the produced vector

Fig. 2. An example illustrating the use of XICL

The primary constructs of XICL include two structures, respectively for the description of options and operands in a command-line, as shown in Figure 3.

The first element of the option construct contains the possible names for the option. An option can have multiple names equivalent to each other, which is common in the Linux programs that conform POSIX conventions and have GNU long options. For example, the first entry in Figure 2 (b) shows that "-e" and "–echo" are equivalent option names. Options with equivalent names correspond to the same set of elements in the input vector to be produced. The other elements of an option construct indicate whether the option allows arguments, the predefined type of the option, the potentially important attributes of the option, and the default value of the option.

The "position" element in an operand structure indicates that the specification of the operand can be applied to all the operands in range [START, END] of the operand list in the command-line. For instance, the third entry in Figure 2 (b) has "position" of "1:1", showing that the entry is applicable to the first operand, 'FILE', in an command line. If the "position" value is "1:3", the first three operands in a command-line will use that entry's specification. Note, unlike equivalent option names, different operands, even if having the same operand construct, correspond to different sets of elements in the input vector to be produced.

```
option{
  name = NAME1:NAME2:...;  // names of the option
  has_arg = N/Y;       // has an argument or not
  type = TYPE;         // type of the option
  attr = ATT1:ATT2:...; // potentially important attributes
  default = DEFAULT; // default value
}

operand{
  position = START:END; // legal positions of the operand
  type = TYPE;         // type of the operand
  attr = ATT1:ATT2:...; // potentially important attributes
}
```

Fig. 3. Primary constructs in XICL

Two of the structure elements, "type" and "attr", deserve more explanations. For easier use, we predefine five types for commonly used options and operands. Each contains a group of predefined attributes that can be used for the value of "attr". The five types are BIN for binary values, NUM for numerical values, STR for strings, FILE for files, and OTH for others. Totally they have 25 predefined attributes [17].

The value of "attr" includes a group of attributes that the programmer regards as important ones to an option. Besides the predefined attributes, XICL allows programmers to write their own procedures to produce attributes for an option, providing the flexibility for addressing special and difficult attributes. The name of the procedure can be used in "attr" as part of the attributes of an option or operand. For example, the attribute "mEDGS" used in the third entry in Figure 2 (b) is such an attribute, supported by a function written by the programmer, "ATTS * mEDGS(char *f)" in Figure 2 (c) in order to determine the number of edges in the input graph file. During the translation of raw inputs, the translator will invoke the procedures to compute those attributes. In order to do that, the returned value of the procedures should be in structure ATTS defined as shown in Figure 4.

The ATTS structure allows a procedure to return a group of attributes, the value of each stored in a buffer. A boolean flag, "isQuan", indicates whether the value is quantitative or qualitative. Quantitative values are stored as floating-point type in the buffer.

XICL Translator and Input Vector The XICL translator converts an arbitrary command-line into a vector containing the attributes of input components. Its input includes a command-line, the input specification of a program, the XICL library, and the library supporting programmer-defined attributes. It parses the command-line in light of the input specification and generates an input vector, whose format is illustrated in Figure 2 (d). Each element in the input vector is an attribute value. The vector length equals the total number of all options plus the attributes of the operands that appear in the command-line. Operands' attribute values always reside at the end of the input vector.

```
// structure of a single attribute
typedef struct ATT_{
    bool isQuan; // a quantitative or qualitative value
    char * value; // the attribute value
  }ATT;

// structure of a set of attributes
typedef struct ATTS_{
    ATT* pAtt; // pointer of an ATT array
    int  n;         // the number of attributes
  }ATTS;
```

Fig. 4. Structures for programmer-defined attributes, used in attribute-deriving procedures.

In the realm of software testing, there are some explorations on extracting program interfaces [8,9,12]. The key difference from XICL is that their main goal is to maximize the coverage of the program and they don't incorporate input attributes that determine program performance.

4 Feature Selection

The input vectors generated from the XICL translator tend to contain some redundancies. Since programmers are often not exactly sure what attributes are important, they tend to include anything potentially relevant. Moreover, relevant attributes may have strong correlation with each other (e.g. the number of words and the number of bytes in a file). These redundancies not only cause the overhead of collecting useless attributes, but more importantly, result in large regression coefficients in input-behavior models and hurt prediction accuracy.

Feature selection is to select the important attributes from preliminary input vectors. From the perspective of statistical learning, feature selection increases the relative coverage of the training data.

Suppose N is the dimensionality of input vectors, l_i and u_i are the lower and upper bounds of the possible input space in ith dimension, and along that dimension $\overrightarrow{l_i}$ and $\overrightarrow{u_i}$ are the lower and upper bound of the area that is covered by the training data. (For simplify, here we assume a continuous range in each dimension.) The upper bound of the covered area is thus a hypercubic with each side spanning from $\overrightarrow{l_i}$ to $\overrightarrow{u_i}$ ($i = 1, 2, \ldots, N$). It is embedded in the N-dimensional possible space defined by l_i and u_i. For an input vector whose corresponding point in the space falls out of the hypercubic, it is often difficult to predict the program's behavior on that input.

As the dimensionality N increases, the required number of training data increases exponentially in order to make the hypercubic cover a certain portion of the possible space. That is well known as the curse of dimensionality in statistical learning [10]. Feature selection reduces the dimensionality of input vectors, and hence increases the relative portion of the covered area.

We explore two methods for feature selection: principle component analysis (PCA), and selection upon T-test. PCA finds the directions in which the input data have the largest variances [10]. It converts the original input space to an orthogonal space (PCA space) and ranks the axes of the new space in terms of the data variances in their directions. The directions are called *principle components*. As PCA space is an orthogonal space, principle components have no correlations with each other. One can discard less-important components to reduce the effect of noises. Using data that are projected to PCA space has been shown beneficial for various regression and classification tasks. On the other hand, because every principle component is the combination of all input attributes, PCA does not eliminate the need for collecting any input attribute.

The second technique used in our exploration is based on T-test [10]. The T-statistic tests the hypothesis that the regression coefficient of an attribute is zero when the other attributes are in the model. T-test produces a P-value, *observed significance level*, for each attribute. Non-significant P-value of an attribute indicates that it does not have predictive capability in the presence of other attributes. We iteratively remove the non-significant attributes. In every iteration, no more than one attribute can be removed because an attribute that does not have predictive capability in the presence of the other attributes may have predictive capability when some of those are removed from the model.

The T-test method works in the original input space. Therefore, unlike the PCA method, it can *remove* non-important attributes and thus save the collections of redundant features. However, since the original input space is usually not orthogonal, it suffers from the correlations between input attributes. We explored both methods and selected T-test for its comparable effectiveness and the advantage in feature reduction.

Through feature selection, the input vectors are converted to *refined vectors* containing only the input information critical to the behavior of interest. Along with the behavior data collected during offline profiling, they compose the training data to the input-behavior model builder.

5 Model Building

Model building is the final step to determine the mapping function from program inputs to runtime behavior. We use linear regression and regression trees to uncover the linear and nonlinear relations between them.

Suppose f is the function mapping input $\overrightarrow{I}$ to a runtime behavior B for a given program. Given training data set $< \overrightarrow{I_i}, B_i >$ ($i = 1, 2, \ldots, N$), the goal of typical regression analysis is to find the approximation of function f, represented by $\hat{f}$, such that the sum of error square, $\sum_{i=1}^{N}(B_i - \hat{f}(\overrightarrow{I_i}))^2$, is minimized.

Linear regression assumes that function $f(\overrightarrow{X})$ is linear to the inputs. The problem is to determine the order and the coefficients of the function so that they minimize the sum of error square.

Nonlinear relations cannot be approximated well by linear regression. A branch in the code of *gzip*, for example, is not taken at all if the input size is larger than 90M; for smaller inputs, the number of times it is taken increases linearly to the input size. The cycles in Figure 5 show the observed behavior, that is, the number of times that branch

is taken for different size of inputs. We use regression tree to deal with the nonlinear relations.

Regression tree splits input space by applying information theory on the training data. By default, a regression tree uses the mean value in a leaf node as the prediction for any input that falls into that node. We build a linear model in each leaf node using linear regression to improve the accuracy. Figure 5 shows the fitting result from both linear regression and regression tree methods for the *gzip* example.

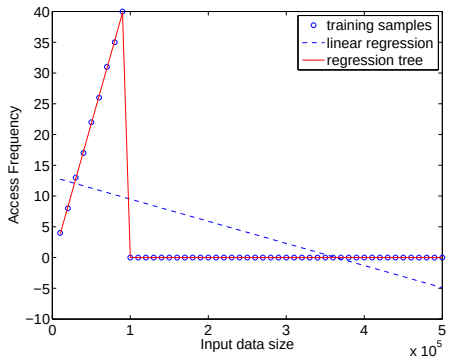

Fig. 5. Regress tree for uncovering nonlinear relationship between inputs and behavior

6 Evaluation

On an IBM Power4 machine, we collect detailed execution profiles through the profile-directed feedback (PDF) functionality in IBM XL compilers. The profile contains the number of times the basic blocks in a program are accessed during an execution, which are the behaviors to be predicted in the experiments. The reason for choosing block frequency is that its effect on program optimization (e.g. function inlining, block layout, loop transformation) is so important that IBM XL compilers mainly rely on it for profile-directed optimization.

6.1 Methodology

Our experiments use an IBM eServer pSeries 690 Turbo machine equipped with 1.7GHz Power4 processors. It runs AIX 5.1 with XL compiler version 6. Table 1 shows the benchmarks we used for our experiments. All programs are from SPEC CPU integer benchmarks except *kmlocal*, which is a K-means clustering program from University of Maryland [11]. Including *kmlocal* is for performance comparison with commercial compilers since those compilations often have already been tuned toward SPEC CPU benchmarks.

The third column of Table 1 shows the numbers of lines of source code in each benchmark. The next three columns show the number of inputs we used, the number of behaviors (block frequencies), and the difference of behaviors induced by those inputs. The 23 to 114 times difference indicates the high sensitivity of those programs on their inputs. The last two columns contain the number of raw features provided by a programmer and the features selected by our technique. The selected features include the size and the type of the input file for *gzip*, the first two numbers in the input file for *mcf* (which correspond to the numbers of timetabled and dead-head trips), the number of lines in the input file for *parser*, and the number of points for *kmlocal*.

Table 1. Benchmark Characteristics

benchmarks	description	lines	#inputs	#behaviors	behavior changes	#raw features	#selected features
gzip	spec2000	8614	50	1775	27X	18	2
mcf	spec2000	2412	250	289	114X	4	2
parser	spec2000	11391	40	4416	39X	7	1
kmlocal	data clustering	6617	50	3333	23X	12	1

6.2 Accuracy of Behavior Prediction

IBM XL compilers have the functionality for profiling-directed feedback (PDF) compilation. The compilation includes three steps. It first instruments the program and let users run the executable on one or more training inputs to generate some profiles. A profile contains the number of times each basic block has been accessed in an execution. Multiple blocks can point to a single number in the profile if the compiler determines that they always have the same number of accesses. The compiler then recompiles the program using the profile(s). This is a typical offline profiling optimization technique.

In our experiments, we treat each frequency number in a profile as a program behavior. The goal is to predict the profile of an execution on a new input without even starting the execution.

Figure 6 shows the prediction accuracy. We use leave-one-out method to evaluate the prediction accuracy. Every time the method takes one data item out of the data set and uses the other data for model training. Then it measures the prediction error for the picked data. The process operates on every data and the average of the errors is taken as the estimation of the prediction error of the constructed model [10]. The formula for accuracy calculation is shown below, with f and $\hat{f}$ for the real and predicted behavior values. Using *max* in the divider is to normalize the accuracy to 0 to 1.

$$accuracy = 1 - \frac{|\hat{f} - f|}{max(f, \hat{f})}$$

Benchmarks *gzip* and *kmlocal* have less than 1% error for all behaviors, *parser* has 4% median error, and *mcf* has about 10% median error. The relatively larger error of mcf is due to its two quantitative features, requiring more training data than others.

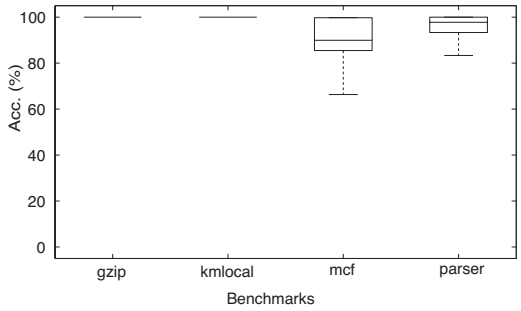

Fig. 6. Boxplot of the prediction accuracy for XL profiles. A box contains the median 50% results and the inside horizontal line shows the median value, with two outside horizontal lines for the range of values.

We obtain the results using a combination of linear regression and regression trees. The model builder automatically chooses the better model for each behavior using 8-fold cross validation [10]. The idea is to take one eighth training data as validation data to measure the models trained by the remaining data.

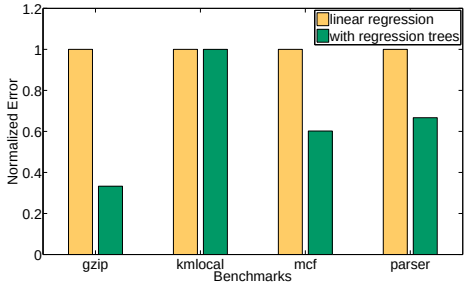

Fig. 7. Error reduction from regression trees

Regression trees improve prediction accuracy significantly. Figure 7 shows the prediction errors from pure linear regression methods and the above combined method. For legibility, we normalize the errors by those of pure linear regressions. Except *kmlocal*, all the other benchmarks show 33% to 68% error reduction. Program *kmlocal* does not need regression trees since it has only linear relations.

6.3 Effects on Optimizations

As a demonstration to the effects of the prediction errors on program optimization, we feed the predicted profiles to XL compiler and compare its PDF compilation results

with those from static compilation and offline profile-directed compilation, which uses the average profile of training runs. Figure 8 shows the comparison on *kmlocal*. We use the result from the default level-2 optimization as the baseline. The level-2 PDF compilation using the predicted profiles produces an executable that is 10% faster. It is remarkable that the compilation, although using just "-O2" optimization level, beats the highest level compilation "-O5" by 5% and the offline compilation by 3.8%. (We didn't use PDF on level 5 because on that level the XL compiler failed in producing profiles for some programs for unclear reasons.) For the three SPEC benchmarks, we didn't see significant benefits from both PDF compilations. A possible reason reason is that the commercial compiler has been highly tuned to those benchmarks.

It is worth to note a desirable feature of the prediction model. During offline training, the model builder can mark the behaviors that are not predictable and avoid predicting them during runtime, which enables the avoidance of negative effects of poor predictions on optimizations.

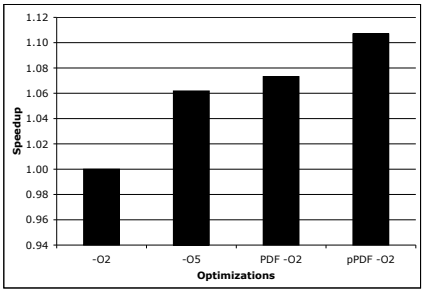

Fig. 8. Speedup of *kmlocal* with different optimizations: "-O2" for the level-2 optimization, "-O5" for the highest level optimization, "PDF-O2" for the "-O2" compilation directed by average profiles, "pPDF-O2" for the "-O2" compilation directed by predicted profiles.

We want to emphasize that the purpose of the experiment is to demonstrate the potential benefits input-behavior models can bring to program optimization. The experiment requires one binary version per input, which is obviously impractical. Although one may classify inputs and generate one version per class, a more proper use of the technique is in continuous (or runtime) program optimization, where, code optimizations occur during a program's execution and the input-behavior models can therefor provide useful guidance.

7 Training Inputs

This section discusses the effect of training inputs, the way to obtain training inputs, and how to hide the training process.

Training data determines the coverage of a model. The input space of a program can have many dimensions. However, typical uses of the program often form just a small subspace. For example, despite that *gzip 1.3.5* has 16 options, users often use few if any of them when compressing a file. This property along with feature selection techniques makes input-behavior modeling tractable. Focusing on typical uses of a program, the technique can avoid poor prediction by simply not predicting for inputs falling outside the covered area.

The distribution of training input in the covered area is important for model building. We conduct a comparison experiment on *gzip* by using two training sets with the size of 20 files distributed from 0 to 500MB respectively in a uniform and exponential distribution. All predictions using linear inputs have accuracy greater than 97.5%, while one thirds of the predictions using exponential inputs have errors larger than 10%. When collecting training data, it is therefore important to make training inputs well distributed in the covered area.

In our experiments, we use a semi-automatic way to generate training inputs. For each program, we create an input-generator based on the understanding of the program. For instance, for *gzip*, we pack a number of different types of files into a tar file and then use an input generator to randomly pick some parts of the file to compose training files of different sizes.

Collecting training inputs can be tedious. Much research is investigating automatic input generation in software engineering area [8, 9, 12]. Although most of the generators are designed for correctness testing, they are potentially usable for optimization purposes. Another possible solution is to implicitly collect inputs from users' real uses of an application. That is particularly useful for continuous optimizations where a program has the opportunity to be continuously optimized after its release. Training takes time. One way to hide the training process from users is to let computer systems start the training process when the machine is idle.

8 Related Work

Prior research in program optimizations falls into four categories in light of the treatment to program inputs. Static compilation either limits itself to the properties holding for any input, or uses ad-hoc estimations for dynamic behavior. For example, a loop is considered ten times more frequently accessed than others [5]. Offline profiling-based methods assume that the profiling runs are the representatives of the real runs and simply make optimization decisions upon those several runs [2]. Neither of them captures dynamic behavior for programs that are input-sensitive. Run-time methods make decisions by monitoring program behavior and trying different optimizations [3, 4, 7, 13, 14, 21]. Although they observe the actual behavior on an input directly, those methods suffer from run-time overhead and hence cannot afford sophisticated analyses. The last class of methods build models by applying machine learning techniques to offline training runs and conduct run-time prediction by plugging in the input characteristics of a real run [16, 20]. Compared to runtime techniques, they move most analysis to offline without sacrificing much confidence in the actual program behavior. However, the previous explorations are limited to several scientific computing kernels, including ma-

trix multiplication and sorting. This work is an exploration to general programs. Ding et al. [6, 23] studied locality prediction across inputs but without dealing with input complexity and feature selection. Shen et al. [19] studied across-input phase sequence prediction based on locality phase analysis [18], but didn't resolve the complexity of inputs either. Berube and Amaral explore the use of Machine Learning techniques in benchmark design for profile-directed optimization [1], but didn't explore the input-behavior models in program optimizations.

For input characterization, some relevant work exists in the realm of software testing, the explorations in test data generation [8,9,12,22]. Most of those techniques focus on the interface to program modules such as procedures or classes, rather than the input to the whole program. Behavior interface specification languages, like Java Modeling Language (JML), enable the specification of the constraints or contract between a class and its clients [15]. These constraints can be regarded as a kind of input attributes, but they are for correctness and don't capture the factors affecting program performance

To our knowledge, this work is the first systematic study in tackling the complexity of raw inputs and providing a framework to consider general input attributes for performance optimizations.

9 Conclusions

This work develops a set of technique to uncover the relations between inputs and dynamic behavior for general programs. It proposes XICL to handle the complexity of program inputs and incorporate the critical input attributes into a formal format. It uses PCA and T-test to remove the redundancies and correlations inside the original input attribute set. It builds the statistical model through linear regression and regression tree methods. The experiments demonstrate the high prediction accuracy of detailed execution profiles for programs with complex control flows. The technique bridges program inputs and dynamic behavior, opening the opportunities for using input-behavior models in continuous program optimizations.

References

1. P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.
2. P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. Hwu. Profile-guided automatic inline expansion for c programs. *Software Practice and Experience*, 22(5), 1992.
3. W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2006.
4. B. Childers, J. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proceedings of 2003 International Parallel and Distribute Processing Symposium (IPDPS)*, 2003.
5. J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1994.
6. C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

7. P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, May 1997.
8. J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, October 1999.
9. P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.
10. T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.
11. T. Kanungo, D. M. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. A local search approximation algorithm for k-means clustering. In *Proceedings of the 18th ACM Symposium on Computational Geometry*, 2002. The program is available at `http://www.cs.umd.edu/users/mount/Projects/KMeans/`.
12. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.
13. T. P. Kistler and M. Franz. Continuous program optimization: a case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.
14. J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2006.
15. G. Leavens, A. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
16. X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.
17. X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. Technical Report WM-CS-2007-07, Computer Science Dept., College of William and Mary, July 2007.
18. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.
19. X. Shen, Y. Zhong, and C. Ding. Phase-based miss rate prediction. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, IN, September 2004.
20. N. Thomas, G. Tanase, O. Tkachyshyn, and J. Perdue. A framework for adaptive algorithm selection in stapl. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.
21. M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah, June 2001.
22. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of International Symposium on Software Testing and Analysis*, 2002.
23. Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3), 2007.