Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors

Yunlian Jiang[†], Eddy Z Zhang[†], Kai Tian[†], Feng Mao[†], Malcom Gethers[†], Xipeng Shen[†],
Yaoqing Gao[⋆]
[†]Computer Science Dept., The College of William and Mary, Williamsburg
[⋆] IBM Toronto Lab

Abstract

This paper presents a finding and a technique on program behavior prediction. The finding is that surprisingly strong statistical correlations exist among the behaviors of different program components (e.g., loops) and among different types of program-level behaviors (e.g., loop trip-counts versus data values). Furthermore, the correlations can be beneficially exploited: They help resolve the proactivity-adaptivity dilemma faced by existing program behavior predictions, making it possible to gain the strengths of both approaches—the large scope and earliness of offline-profiling–based predictions, and the cross-input adaptivity of runtime sampling-based predictions.

The main technique contributed by this paper centers on a new concept, seminal behaviors. Enlightened by the existence of strong correlations among program behaviors, we propose a regression-based framework to automatically identify a small set of behaviors that can lead to accurate prediction of other behaviors in a program. We call these seminal behaviors. By applying statistical learning techniques, the framework constructs predictive models that map from seminal behaviors to other behaviors, enabling proactive and cross-input adaptive prediction of program behaviors. The prediction helps a commercial compiler, the IBM XL C compiler, generate code that runs up to 45% faster (5%–13% on average), demonstrating the large potential of correlation-based techniques for program optimizations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization, compilers

General Terms Performance, Measurement

Keywords Program behavior prediction, Dynamic optimizations, Seminal behaviors, Feedback directed optimizations, Program behavior correlations

1. Introduction

Accurate prediction of program behaviors is the basis of various program optimizations. Program behaviors in this paper refer to the operations of a program and the ensuing activities of the computing system, in relation to the input and running environment. Examples include memory references, data values, function calling frequencies, and so on. The prediction of program behaviors critically determines how optimizers transform a program and the resulting performance. As the complexity in modern hardware and software continuously grows, accurate behavior prediction becomes both more important and more challenging than before.

Besides accuracy, two other properties of behavior prediction are essential for optimizations: scope and timing. The scope of a prediction may be a small execution interval, a loop, a procedure, or the entire program. The larger the scope is, the more likely the optimizer is able to avoid local-optimum traps when making optimization decisions. The third property, the timing of prediction, refers to when a prediction can occur. The earlier the prediction occurs, the earlier an optimization can happen, and the larger the portion of the execution that may benefit from the resulting code. We also call the earliness the *proactivity* of a prediction.

In existing program optimizers, behavior predictions are based on either training runs (in profiling-based optimizers) or runtime sampling (in runtime optimizers). Their strategies are essentially the same: using the behaviors of a program component (e.g., a procedure or loop) observed previously (in either a training run or the earlier part of the current execution) to predict the future behaviors of the *same* component. This strategy, although effective for many programs, can lead to a *proactivity-adaptivity dilemma*: Predictions based on training runs have a large scope and good proactivity, but cannot adapt to input changes, whereas, predictions based on runtime sampling have good adaptivity but limited scope and proactivity.

Recent studies show that prediction based on program inputs may gain the strengths of both approaches, improving optimizations significantly. For instance, improvements of 7%–21% have been observed on a variety of Java programs [14]. However, that approach relies on programmers' manual specifications on program inputs. An automatic solution to the proactivity-adaptivity dilemma remains an open question.

In this paper, we attack the problem by exploiting the correlations among the behaviors of program components. The intuition is simple. Consider the trip-counts (number of iterations) of two loops, L1 and L2. Suppose that they strongly correlate with each other (e.g., the trip-counts of L1 are always about double those of L2). Then, as soon as the trip-counts of one of them become known in an execution, the trip-count of the other will be easily predicted.

A set of questions must be answered for using those correlations for behavior prediction. How common are such correlations in

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-60558-635-9/10/04. . . $5.00.

programs? How can they be identified? And how can they be exploited?

This paper presents our explorations in answering those questions. It first reports a systematic measurement (Section 2), showing that strong statistical correlations exist not only among the behaviors of different program components commonly, but also among different types of program-level behaviors (e.g., loop trip-counts versus data values).

It then introduces a technique to exploit the correlations for program behavior prediction and optimizations. The technique centers on a new concept (Section 3.1), *seminal behaviors*, which refers to a small set of behaviors that strongly correlate with most other behaviors in the program, and meanwhile, expose their values early in typical executions. Section 3.2 presents a framework for identifying seminal behaviors and building predictive models that map from seminal behaviors to other behaviors. Section 3.3 and Section 3.4 discuss how to use seminal behaviors for behavior prediction and program optimizations, respectively. Experimental results in Section 4 show that seminal behaviors can lead to accurate prediction of several types of program behaviors. The prediction is distinctive in being both cross-input adaptive and proactive—the whole-program behaviors can be predicted as soon as the values of the seminal behaviors get exposed (no later than 10% of the execution). The prediction helps a commercial compiler, IBM XL C/C++ Enterprise Edition 10.1, generate code that runs up to 45% faster (5%–13% on average).

In summary, this work makes three main contributions.

- To the best of our knowledge, this work is the first study that systematically explores and exploits the statistical correlations among different types of program-level behaviors for dynamic program optimizations.
- This work introduces the concept of seminal behaviors and their identification, laying the foundation for correlation-based behavior prediction.
- The proposed seminal-behavior-based prediction offers an automatic way to predict program behaviors proactively and cross-input adaptively, resolving the proactivity-adaptivity dilemma in existing techniques.

2. Correlations Among Program-Level Behaviors

In this section, we first present a qualitative view of the behavior correlations, using an example to illustrate the intuition behind in. We then report the measurement of the correlations in 14 programs, quantitatively examining the properties and strength of the correlations.

2.1 A Qualitative View

Formally, we define *behavior correlations* as follows. The behaviors of two program components are correlated if, when the inputs to the program change, their values vary together in a way not expected on the basis of chance alone.

The existence of behavior correlations is due to the connections inherent in program code. Take Figure 1 as an example. It outlines a simplified code for mesh generation. The "main" function invokes a recursive function "genMesh" to create a mesh for the vertices listed in an input file. Before the creation, it reads vertices and a reference mesh in the function "mesh_init" in preparation; after the creation, it verifies the generated mesh in the function "verify".

The example illustrates both deterministic and non-deterministic connections among program behaviors. A deterministic example is the relation between the value of "vN" and the trip-counts of the two "for" loops in "mesh_init". Once the value of "vN" is known, the numbers of the iterations are easily determined. Similar relations exist between "vN" and the number of times the recursive function "genMesh" is invoked, and the size of the vertex array "v". A non-deterministic connection exists between the "while" loop in the function "mesh_init"and the "for" loop in the function "verify". Although these two loops tend to have the same trip-counts, they may differ with each other when the generated mesh is wrong. Some of these relations may be detectable by compilers, but many of them are undetectable because of the complexities in pointer analysis, alias analysis, and interprocedure analysis.

The existence of the correlations can be explained from another perspective. For a given program in a given environment, all the behaviors essentially stem from the same entity–the inputs to the program. They are hence likely to correlate with one another (although do not necessarily correlate, if two behaviors stem from the different parts or features of an input) .

2.2 Quantitative Measurements

2.2.1 Methodology

Behaviors under Study In this experiment, we concentrate on the following program-level dynamic behaviors: loop trip-counts (numbers of iterations), procedure calling frequencies, the number of times a basic block is accessed (*data profiles*), the counts of certain values from some special expressions, referenced by the nodes and edges in the control flow graph, (*edge profiles* and *node profiles*, respectively), which are important for program optimizations (judged by the IBM XL C/C++ compiler v10.1). We choose these types of behaviors because of their importance for program optimizations. In fact, the final three kinds of profiles compose the entire feedback the IBM XL compiler uses for its profiling-directed compilation. The other two kinds of behaviors are important for loop optimizations and function inlining.

We collect the loop trip-counts through a modified GCC (v4.3.1), and obtain the calling frequencies through GNU gprof (v2.19). The machine is equipped with Intel Xeon 5310 quad-core processors running the Linux 2.6.22 operating system. We get the other three kinds of profiles using IBM XL C/C++ Enterprise Edition 10.1 on IBM Power5 processors (with the IBM AIX 5.3.8 operating system installed). The compiler is the primary commodity compiler on AIX platforms. The use of two different platforms offers the opportunity for studying the correlations between the behaviors of a program on different platforms.

Programs Table 1 lists the programs used in our experiments. They include 14 C programs in SPEC CPU2000 and SPEC CPU2006. We include no C++ or Fortran programs because the instrumentor we implement (the modified GCC) currently works only for C programs. We exclude those programs that are either similar to the ones included (e.g., bzip2 versus gzip) or have special requirements on their inputs and make the creation of extra inputs (which are essential for this study) very difficult.

Although each benchmark comes with several sets of inputs by default, more inputs are necessary for a systematic study of statistical correlations. We collect more inputs as shown in the fourth column of Table 1. During the collection, we try to ensure that the inputs are typical in the normal executions of the benchmarks. Specifically, we collect those inputs by either searching the real uses of the corresponding applications or deriving the inputs after gaining enough understanding of the benchmark through reading its source code and example inputs. Some of those inputs come from Amaral's research group [6]. The sixth column of Table 1 shows the changes that different inputs brought to the loop trip-counts, reflecting the large differences among those inputs.

```
main(int argc, char * argv){
  ...
  mesh_init (dataFile,mesh,refMesh);
  genMesh (mesh,0,mesh->vN);
  verify (mesh, refMesh);
}
```

```
// recursive mesh generation
void genMesh (Mesh *m, int left, int right){
  if (right>3+left){
    genMesh (m, left, (left+right)/2);
    genMesh (m, (left+right)/2+1, right);
    ...}
  ...
}
```

```
void verify (Mesh *m, Mesh *mRef){
    ...
  for (i=0; i< m->edgesN; i++){
    ...
  }
}
```

```
Mesh * mesh_init
(char * initInfoF, Mesh* mesh, Mesh* refMesh)
{
   FILE * fdata = fopen (initInfoF, "r");
   fscanf(fdata, "%d\n", &vN);
   mesh->vN = vN;
   v = (vertex*) malloc (vN*sizeof(vertex));
   // read positions of vertices
   for (i=0; i<vN; i++) {
      fscanf(fdata, "%f %f\n", &v[i].x, &v[i].y);
     ...}
   // sort vertices by x and y values
   for (i=1; i< vN; i++){
       for (j=vN-1; j>=i; j--){
          ...}
    }
   while (!feof(fd)){
      ...
     /* read edges into refMesh for
        later verification */
   }
}
```

Figure 1. A simplified mesh generation program.

Table 1. Benchmarks

Program					Factor of changes	Mean corr coef from loop to				
name	description	lines	inputs	loops	caused by inputs	loop	call	edge	node	data
ammp	Computational Chemistry	13263	20	425	9.9×10^1	1.00	0.97	0.97	0.91	1.00
art	Image Recognition / Neural Networks	1270	108	101	4.0×10^4	1.00	1.00	0.99	0.88	0.70
crafty	Game Playing: Chess	19478	14	425	4.6×10^8	0.99	0.97	1.00	0.99	0.98
equake	Seismic Wave Propagation Simulation	1513	100	106	1.0×10^2	1.00	1.00	0.99	0.15	1.00
gap	Group Theory, Interpreter	59482	12	1887	1.1×10^8	0.99	0.98	0.93	0.77	0.91
gcc	C Compiler	484930	72	7615	1.1×10^6	0.98	0.98	0.97	0.94	0.92
gzip	Compression	7760	100	223	4.3×10^7	0.98	0.94	0.95	0.86	0.98
h264ref	Video Compression	46152	20	2074	2.1×10^9	1.00	1.00	1.00	0.98	1.00
lbm	Fluid Dynamics	875	120	27	6.0×10^6	1.00	0.93	1.00	1.00	1.00
mcf	Combinatorial Optimization	1909	64	76	1.4×10^5	0.94	0.99	0.99	0.30	1.00
mesa	3D Graphics Library	50230	20	995	2.0×10^1	1.00	1.00	1.00	0.28	1.00
milc	Physics / Quantum Chromodynamics	12837	10	473	2.1×10^9	0.98	0.98	1.00	0.70	1.00
parser	Word Processing	10924	20	1350	2.1×10^6	0.99	1.00	0.99	0.97	0.98
vpr	FPGA Circuit Placement and Routing	16976	20	435	3.9×10^6	0.99	0.98	0.99	0.84	0.64

2.2.2 Calculation of Correlations

We use the standard way in statistics, the Pearson product-moment correlation coefficient [10], to quantify the correlations. Let X and Y represent two behaviors of a program, such as the trip-counts of two different loops. Suppose we run the program for n times, each time on a different input data set. We get n measurements of both X and Y, written as x_i, y_i where $i = 1, 2, \ldots, n$. The correlation coefficient of X and Y is calculated as follows:

$$r_{XY} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_X s_Y},$$

where, $\bar{x}$ and $\bar{y}$ are, respectively, the mean values of X and Y, and s_X and s_Y are the sample standard deviation of X and Y.

The value range of correlation coefficients is [-1,1]. The absolute value of a correlation coefficient indicates the strength of a *linear* relationship between two random variables. The higher the absolute value is, the stronger the relationship is.

Correlation coefficients cannot directly reflect the non-linear relationship between behaviors. (The regression to be presented in Section 3 captures some of those relationships.) But that limitation does not affect the conclusion of the experiment: The observed correlations already confirm the common existence of strong correlations among program behaviors, as shown next.

2.2.3 Measurement Results

In this section, we first report the correlation coefficients among the trip-counts of different loops and then describe the coefficients from loops to other types of behaviors. The concentration on loops is because of their importance in programs and their critical roles in the exploitation of correlations, as Section 3 will describe. In all measurements, we ignore the behaviors that have never occurred in any of the executions (correlations among them are trivially 1).

Inter-Loop Coefficients In the study of correlations among loops, we compute the correlation coefficients between the trip-counts of every pair of loops. For each loop, we find the highest correlation coefficient between this loop's trip-counts and all the other loops'. We refer to such coefficient as the *inter-loop coefficient* of this loop. If its inter-loop coefficient is high, this loop's trip-count is likely to be able to be predicted accurately as soon as the trip-count of the other loop is known. The seventh column in Table 1 shows the average of the absolute values of such coefficients across all loops in each program. All numbers are greater than 0.98 except 0.94 for *mcf*. Since the maximal correlation coefficient is 1, the inter-loop correlations are remarkably strong.

More detailed information on the correlations appears in the first column of Figure 2. Each pie shows the distribution of the values of the correlation coefficients. The dominance of the high coefficients further confirms the common existence of strong correlations between loop trip-counts.

Loops and Others We measure the correlation coefficients between loop trip-counts and each of the four other types of behaviors: procedure calling frequencies, edge, node, and data profiles. The corresponding correlations are respectively denoted as loop-call, loop-edge, loop-node, and loop-data.

For each target behavior (e.g., the calling frequency of a procedure), we compute the correlation coefficients between the trip-counts of every loop and this target behavior. The highest value is taken as the coefficient from loops to this target behavior. It (partially) reflects the possibility to predict the value of the target behavior from a loop.

The rightmost four columns in Table 1 report the average of such coefficients for each type of behaviors. The loop-call and loop-edge coefficients are all greater than 0.93. The loop-node coefficients are the lowest among the four, smaller than 0.84 for five programs. The main reason is the many conditional statements in their source code. The loop-data coefficients are higher than 0.92 for most of the programs. The two exceptions are programs, *art* and *vpr*, 0.7 and 0.64 respectively. The rightmost four columns of Figure 2 reveal the distribution of those coefficients.

Summary and Implications This section shows that even though conditional branches in a program sometimes weaken the correlations between loops and basic block execution frequencies, overall, strong statistical correlations exist between loop trip-counts, and from loop trip-counts to other types of behaviors. It suggests the possibility of using the correlations for runtime behavior prediction. When the values of certain types of behaviors of some program components (e.g., a set of loop trip-counts) are exposed in an execution, we may use them as the predictors of the behaviors of other (to-be-executed) components in the program. This kind of prediction is both proactive, occurring before the execution of the other program components, and adaptive, being specific to the current input data set.

The usefulness of the prediction is determined by its accuracy and how early the predictors expose their values. The next section describes a framework for identifying the appropriate set of predictors and the use of them for program optimizations.

3. Exploitation of Behavior Correlations

In this section, we describe a framework and a set of techniques for exploiting the correlations revealed in the previous section. The framework has two functions: 1) to find the small set of behaviors that expose their values early in an execution and strongly correlate with other behaviors in the execution; 2) to build predictive models that capture the correlations among program behaviors. At the center of the framework is the concept of seminal behaviors.

3.1 Concept of Seminal Behaviors

Roughly speaking, seminal behaviors are those behaviors that are suitable to be used for predicting other behaviors. Before presenting the formal definition, we need to introduce two concepts.

DEFINITION 1. *For a given set of behaviors B and a threshold r, a set of behaviors S is a* <u>*predictor set*</u> *of B if there is a mapping function f from S to B such that the average Euclidean distance between $f(V_S)$ and V_B is less than r, where V_S and V_B are the values of S and B.*

DEFINITION 2. *Let S be a predictor set of a given set of behaviors B of a program G. In an execution of G, let B' represent the subset of B whose values are exposed after the exposure of the values of S. The* <u>*earliness*</u> *of S in that execution is defined as $|B'|/|B|$.*

In these two terms, we express the definition of seminal behaviors as follows.

DEFINITION 3. *For a given set of behaviors B of a program G,* <u>*a seminal behavior set,*</u> *S, is a predictor set of B whose earliness, averaged across all executions of G, is the highest among all the predictor sets of B. Each member of S is called* <u>*a seminal behavior*</u>.

The definition suggests two properties of a seminal behavior set. First, it leads to accurate prediction of other behaviors. Second, it enables the earliest (on average) prediction among all B's behavior-predictor sets. These two properties make a seminal behavior set desirable for the uses in proactive and cross-input adaptive prediction of program dynamic behaviors.

3.2 Identification of Seminal Behaviors

The concept of seminal behaviors suggests that whether a behavior is a seminal behavior depends on B, the behaviors to be predicted. In this work, we concentrate on those five types of behaviors (loop trip-counts, calling frequencies, edge, node, data profiles) listed in Section 2.2.1.

A brute-force way to identify seminal behavior sets is to enumerate every possible subset of the program's behaviors, try all kinds of mapping functions, and consider all executions of the program. The high complexity suggests the need for heuristics and approximations. We employ a heuristics-based framework as described next.

Candidate Behaviors Rather than consider all kinds of behaviors, we select two types of behaviors as the candidates for seminal behaviors. The first is program *interface behaviors*, which mainly include the values directly obtained from program inputs. Specifically, this type of behaviors include the values obtained directly from command lines[1] and file operations. We ignore the content of a file if the corresponding file operations are within a loop whose trip-count is either large (greater than 10 in our experiments) or unknown during compile time. Those data are likely to be a massive data set for processing; their values may not influence the coarse-grained behaviors much, but including them may significantly inflate the candidate behavior set and complicate the recognition of seminal behaviors. Instead, we include the trip-counts of those surrounding loops as they often reflect the size of the data set. We also record the size of input files, obtained through file descriptors, as another clue of the size of data. All these behaviors together form a set called the *interface behavior set*.

The second type of behaviors we include are the trip-counts of all the loops (beside those that are already counted as interface behaviors) in the program. This inclusion is due to the importance of loops and the correlations the previous section shows.

Computation of Predictive Capability From the definition of seminal behaviors, we know that they must be able to lead to accurate prediction of other behaviors. For a given set of behaviors B, we define *predictive capability* of a set S as the number of behaviors in the set $B - S$ that can be predicted from S with an accuracy above a predefined threshold (80% in this study).

For the reduction of complexity, we take a simplification as follows. We limit B to loop trip-counts during the examination of

[1] In this work, we assume that the applications are C programs with inputs coming from command lines. The analysis can be applicable to other programs with interactive features; details are out of the scope.

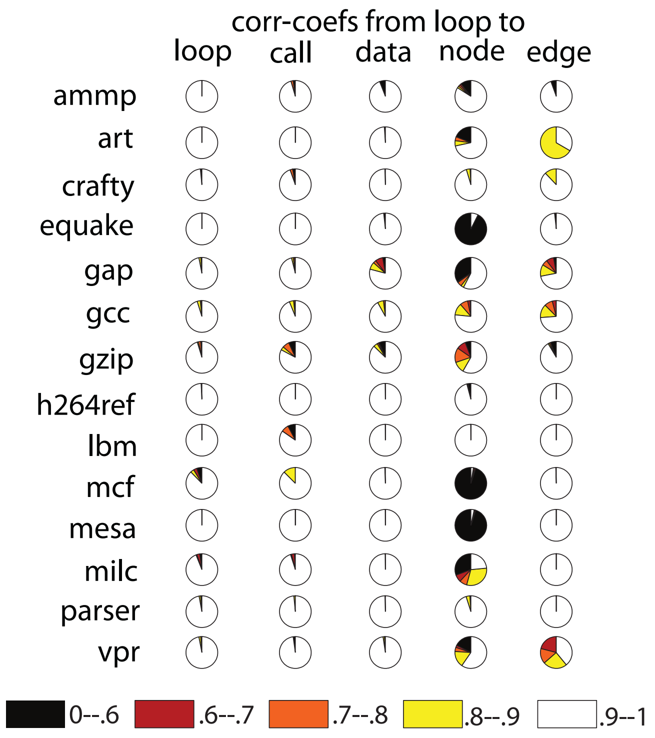

Figure 2. Distribution of values of correlation coefficients.

the predictive capability of different candidate behavior sets. The intuition is that because there are strong correlations between loops and other types of behaviors, the sets selected in this way are likely to show good predictive capability on other types of behaviors as well. The results in Section 4 confirm this intuition.

The computation of predictive capabilities in our experiments is based on the standard 10-fold cross-validation [10]. It works iteratively. Suppose we did N profiling runs of a program, and obtained N instances of S and B. In each iteration, 9/10 of the N instances are used to construct predictive models from S to B, and the other 1/10 are used to test the model for prediction accuracy.

Next, we describe the framework for seminal behavior identification first, and then explain the construction of predictive models.

Identification Framework A brute-force way to identify seminal behaviors is to compute the predictive capability and earliness of every subset of candidate behaviors and choose the best one. To circumvent the exponential complexity, we take an incremental approach, which gradually builds a number of affinity lists. *An affinity list* is a list consisting of two sets of behaviors, a header set and a body set, such that the values of the behaviors in the header can lead to accurate prediction of the values of those behaviors in the body.

The construction of affinity lists proceeds as follows. It starts with the set of interface behaviors, because of their earliness and direct connections with program inputs. It ignores those interface behaviors that have constant values across all training runs as they are irrelevant to behavior variances among different runs. It then uses the remaining interface behaviors as predictors, builds predictive models from them to each loop trip-count. The loop trip-counts that can be predicted accurately are put into the body of the first affinity list. All the interface behaviors that appear in the predictive models are put into the header of that affinity list. The first column in Figure 3 illustrates the result of this step on a program *mcf*.

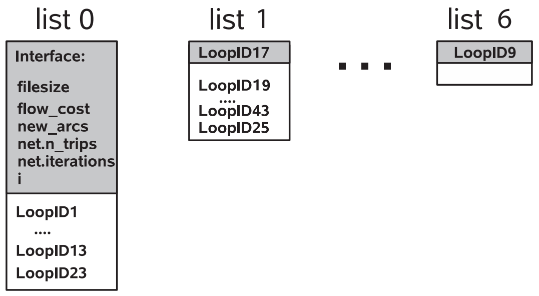

Figure 3. The affinity lists of program *mcf*.

The construction process then selects, in an order shown next, one of the remaining candidate behaviors as the header of the second affinity list, computes the predictive capability of this header on the remaining behaviors, and adds the predictable ones into the body of the second affinity list. This process continues until no candidate behaviors are left. For the program *mcf* shown in Figure 3, the process constructs 6 affinity lists; the last one has an empty body. An affinity list with an empty body means no behaviors are predictable from its header.

In our experiment, the order, in which loop behaviors are selected, is the order of the time when the trip-counts of the loops get exposed. (The average order is used when there are two or more training runs.) This is to maximize the earliness of the resulting seminal behavior set.

The union of the headers of the affinity lists forms a possible seminal behavior set as all other candidate behaviors are predictable from it. These header sets may be ranked in a descending order of the sizes of their bodies. The exclusion of the low-rank header sets may have little influence on the prediction of most behaviors. Section 4 examines the trade-off of the size and the predictive capability of those sets.

We note that the headers of the affinity lists essentially embody a kind of characterization of program inputs: Each of the headers reflects some aspects or attributes of the inputs; together they determine most of the program's behaviors.

Predictive Models In this part, we describe some details of the construction of the predictive models used during the identification process. We employ two standard regression techniques, namely LMS linear regression and Regression Trees [10]. The former handles linear relations among behaviors, the latter for non-linear relations. The construction process applies Regression Trees only if the linear regression results are not good enough (automatically assessed through cross-validation). During the construction of the first affinity list, the standard forward stepwise feature selection [10] is used so that only important interface behaviors are stored in the header.

Both LMS and Regression Trees models are efficient to build and use. The resulting models are represented by only a small number of coefficients (for linear models) and questions (for Regression Trees). (We limit the tree size to be no greater than 10.) A specially appealing feature of Regression Trees is that it handles both numerical and categorical values smoothly.

Discussions There are several points worth mentioning. First, as we mentioned earlier, static analysis cannot capture many relations because of the complexities in the program, and difficulties in pointers and aliases analysis. But its integration into our framework may help reduce certain overhead by revealing some definite connections.

Second, the accuracy threshold used in the construction of affinity list determines the number of resulting affinity lists. The appropriate value depends on the ultimate use of the prediction. We take 80% as the threshold for our experimental exploration.

Finally, using multiple behaviors rather than a single behavior as the header in each affinity list (besides the first list) may improve the prediction accuracy. However, the large number of possible combinations of behaviors would significantly increase the training cost. Detailed explorations are out of the scope of this paper.

3.3 Uses for Behavior Prediction

Using seminal behaviors for behavior prediction is straightforward. It just needs to build a predictive model mapping from the values of the seminal behaviors to those of the target behaviors. We employ the same regression techniques as described in the previous paragraph for the model construction. In a new execution, as soon as the values of the seminal behaviors are exposed, the models will be able to immediately predict the values of the target behaviors, hence enabling proactive, cross-input adaptive prediction, opening new opportunities for dynamic optimizations.

3.4 Uses in Program Optimizations

As shown in the previous sections, a small set of seminal behaviors are enough to produce reasonably accurate prediction for various types of behaviors, suggesting the potential of seminal behaviors for input-specific program optimizations. This section discusses the uses.

For programs running in a managed environment, such as Java Virtual Machines (JVMs), the seminal behaviors-based prediction may help the Just-In-Time (JIT) compilers make better decisions on the timing and parameters in method optimizations. During an offline profiling process, the seminal behaviors of an application and the predictive models for certain kinds of program behaviors can be determined using the approach described in Section 3.2. After that, when a new run of the application launches on a new input, the runtime system can use the predictive models to predict how the application will behave in the rest of this run as soon as the values of the seminal behaviors get exposed in the current execution. The JIT compiler can then optimize the application in a way that best suits the predicted behaviors. This process is similar to many dynamic optimization systems (such as JVMs), except that the knowledge of seminal behaviors makes the optimization *proactive* to the major part of the execution. As showed by previous studies [5, 14], proactive dynamic optimizations may outperform traditional reactive schemes significantly.

For programs written in imperative languages, such as C, seminal behaviors may boost the effectiveness of dynamic code version selection for performance improvement. Dynamic code version selection is a technique for enabling the adaptation of program optimizations on input data sets [9]. The default scheme works in this way. For each function, the compiler generates several versions using different optimization parameters. At run time, those versions are used and timed in the first certain number of invocations of a function; the version taking the shortest time to run is selected for the rest of the execution. The reliance on runtime trials of different versions makes the technique hard to apply to the functions that have very few invocations in a run. (Such functions could contain major loops and be important for the program execution.) If we can build a mapping from the values of seminal behaviors to the suitable versions during training time, we can immediately predict the best version to use for a real run as soon as the values of seminal behaviors get exposed in that run. In this way, we do not need the trials of the different versions in real runs, hence circumventing the limitations the default scheme has.

4. Evaluation

This section first reports the identified seminal behaviors and their effectiveness in predicting other behaviors, and then examines the potential of the prediction for performance improvement through profile-directed-feedback (PDF) compilation by the IBM XL compiler. (Section 2.2.1 has described the platforms and programs we use.)

Seminal Behaviors and Prediction Accuracy Table 2 reports the accuracy of seminal-behavior–based behavior prediction. The data in the table are organized in four sections. The first (leftmost) section corresponds to the case when only the interface behaviors are taken as seminal behaviors, the second and third sections correspond to the cases when the seminal behavior sets also include the other affinity headers (all are loop trip-counts) whose earlinesses are over 90% and 80%, respectively. The rightmost section corresponds to the case when the headers of all affinity lists are included in seminal behavior sets.

In each section, the column "num" lists the sizes of the seminal behavior sets, and the other columns show the average accuracy when we use the seminal behaviors to predict each of the five types of behaviors. The standard 10-fold cross-validation [10] is used so that each time the testing and training data have no overlap.

We first discuss the overall average. With just interface values, the loop and data behaviors can be predicted with an over 92% accuracy, while the accuracies on the other three types of behaviors are from 69% to 82%. Because the values of all interface behaviors get exposed in the first 1%–3% portion of every execution, this case has very high earliness. When we sacrifice some earliness to allow certain loop behaviors get exposed and used along with the interface values for prediction (the 90% case), the accuracies improve by about 3% for loop trip-counts and data profiles, 7% for function invocations and node profiles, and 10% for edge profiles. However, when more loops are used for prediction, the prediction accuracies increase only a little. For some programs, the accuracies even become worse. The diminishing benefits are due to the well-known *curse of dimensionality* in statistical learning—too many predictors cause overfitting to the models.

We now concentrate on the "earliness $\geq$ 90%" section in Table 2. All programs show high prediction accuracies on loop trip-counts and data profiles. Several programs show modest accuracies on the other three types of behaviors, mainly because of the large numbers of conditional statements in those programs. An extreme example is *mesa*. Its two largest files are *get.c* and *eval.c*. There are 5 switch-case statements with 1008 cases in *get.c*, and 231 cases and 108 "if" statements in *eval.c*. So many branches make it hard to predict the execution paths to a given node in control flow, which explains the low prediction accuracy of its node profiles. It is consistent with the low correlations between loops and node profiles shown in Table 1.

Not all prediction accuracies are consistent with the correlations in Table 1. For example, *mcf* has very low correlations between loops and node profiles, but shows 90% prediction accuracy for node profiles. This inconsistency is because as mentioned earlier, correlation coefficients cannot capture non-linear relations, whereas the prediction models can. On the other hand, even though *crafty* shows high correlation coefficients between loops and node profiles, the prediction accuracy of node profiles is only 45%. One reason for the inconsistency is that some important loops are missing in the seminal behavior set because their trip-counts get exposed late. When all headers of the affinity lists are used, the accuracy boosts to 61%. Another reason for the modest accuracy is the small number of inputs *crafty* has. More inputs and improved seminal behavior identification can possibly improve the prediction accuracy.

As one of the most complex benchmarks, the program *gcc* is worth a detailed examination. This GNU general-purpose compiler includes hundreds of command-line options, 130 files, 484,930 lines of code. Besides the complexity of its code, its inputs—C programs—are also very difficult to characterize, as shown in previous work [12].

In our work, the seminal behavior identification process marks 48 interface behaviors, from which, four are identified as seminal behaviors because of their strong correlations with loop trip-counts. Among the 7,615 loops of *gcc*, the process constructs 129 affinity lists with one loop in each list header. The values of 50 out of the 129 headers get exposed in the first 10% portion of each *gcc* execution. After they are added into the seminal behavior set, the prediction accuracy for almost all types of behaviors jumps to over 93% (except 86% for function invocations). Given the complexity of *gcc*, these results offer an especially strong evidence of the feasibility of seminal behaviors for proactive, cross-input adaptive prediction of program behaviors.

As to the overhead, the main time is on the training process, including the identification of seminal behaviors and the construction of predictive models. But since the training happens during offline, the overhead is not critical. The prediction of a behavior using the predictive models takes little time, as it only requires the computation of a linear expression and possibly several conditional checks (for Regression Trees). In our experiments, the prediction of the 7,615 loops of *gcc* takes the longest time, but still finishes within 11 milliseconds. The next section demonstrates the practical usefulness of seminal behaviors in program optimizations.

Potential for Performance Improvement. We examine the potential of seminal behaviors for performance improvement through the PDF (profile-directed-feedback) compilation offered by the IBM XL C compiler.

The default PDF compilation works in two steps. For a given application, the compiler first instruments it (through the option "-qpdf1") and lets users run it on a training input. That run generates a file, containing three sections that correspond to the node, edge, and data profiles mentioned in Section 2.2.1. The compiler then recompiles the application using the profiling results as feedback (through the option "-qpdf2").

To examine the usefulness of seminal behaviors, we let the XL compiler do PDF compilation using those predicted profiles. We compare the performance of the resulting code with that of the static compilation (on the highest optimization level) and the PDF compilation on real profiles. Figure 4 shows the results (average of 5 repetitions; negligible variances observed). Because the speedup of a program varies across inputs, we show the range of speedup, with the performance of static compilation as the baseline, on all inputs in each bar.

It is worth mentioning that unlike the predicted profiles, the real profiles are typically not usable by the compiler in production runs as they are not available until the finish of the execution. So the right bars in Figure 4 essentially show the speedup in the ideal case. The occasional better performance from predicted profiles than that from real profiles is due to the imperfect design of the compiler. It is no surprise given the extreme complexity in compiler construction.

On average, the programs produced by predicted profiles achieve 6%–15% speedup, within 2% distance from the speedup real profiles can bring (the right bars). The up to 14% speedup of GCC specially demonstrates the potential of seminal behaviors for input characterization.

5. Discussions

Seminal-behavior-based prediction may occasionally exhibit low accuracies. It would be desirable to prevent such predictions from negative affecting program optimizations. One option is to check the prediction accuracy by observing the behaviors exposed during an execution and either correct or disable the predictions for other behaviors accordingly. The other option is to adopt the discriminative scheme [14], which assesses the predictive capability of a seminal behavior set through history runs and use the set for prediction only if its assessment result passes some confidence threshold.

We note that proactive behavior prediction complements rather than conflicts with reactive prediction and program phase detections. Program phases can be treated as a special kind of behaviors to be modeled and predicted by the framework. A system with both proactive and reactive schemes may predict large-scope behaviors early, and meanwhile, adapt to program or environment changes quickly.

6. Related Work

This work makes the following main contributions: the uncovering of the strong correlations among the behaviors of different program components and among different types of program behaviors, and the introduction of seminal behaviors and the uses of them for cross-input proactive prediction of large-scope program behaviors. This section summarizes the previous studies related to each of them.

Correlations between Program Components We are not aware of any prior studies on systematically exploring and exploiting statistical correlations between the behaviors of different program components. Previous explorations on the connections between program components mainly concentrate on static analysis [1, 2], including data-flow analysis, symbolic analysis, and so on. The connections revealed in those analyses typically have a limited scope, because of the difficulties in pointer analysis and alias analysis.

Correlations between Different Types of Behaviors Previous studies (e.g., [3, 15]) have observed strong correlations between

Table 2. Numbers of Seminal Behaviors and Prediction Accuracies

Prog	interface values						earliness $\geq 90\%$						earliness $\geq 80\%$						all headers					
	num	accuracy					num	accuracy					num	accuracy					num	accuracy				
		loop	call	edge	node	data		loop	call	edge	node	data		loop	call	edge	node	data		loop	call	edge	node	data
ammp	1	99.5	96.7	100	91.1	99.7	1	99.5	96.7	100	91.1	99.7	1	99.5	96.7	100	91.1	99.7	1	99.5	96.7	100	91.1	99.7
art	4	91.0	96.8	100	82.0	96.8	4	91.1	96.8	100	80.0	96.1	4	89.7	93.4	100	80.1	96.2	5	90.4	88.1	100	85.2	96.3
crafty	1	89.9	58.9	88.2	35.5	76.0	2	91.1	63.0	90.8	44.5	79.3	4	93.0	70.8	90.5	58.2	84.7	6	93.6	72.6	89.7	61.3	85.6
equake	1	98.0	100	100	96.3	99.3	1	98.0	100	100	96.3	99.3	1	98.0	100	100	96.3	99.3	1	98.0	100	100	96.3	99.3
gap	2	97.5	44.9	11.9	44.2	76.6	7	99.5	78.7	56.3	69.7	88.5	8	99.6	79.7	59.0	71.6	89.2	8	99.6	79.7	59.0	71.6	89.2
gcc	4	82.9	38.9	56.2	61.0	78.5	54	97.0	86.1	93.6	95.4	95.6	92	99.2	94.9	98.8	100	99.1	133	99.3	94.9	97.8	100	99.4
gzip	3	92.2	87.0	84.1	67.5	94.5	6	91.6	87.6	83.5	69.0	94.5	6	91.6	87.6	83.5	69.0	94.5	6	91.6	87.6	83.5	69.0	94.5
h264ref	3	99.8	99.8	98.7	98.8	99.8	4	99.8	99.7	97.0	97.8	99.7	4	99.8	99.7	97.0	97.8	99.7	5	99.8	99.5	98.0	97.9	99.7
lbm	3	99.8	90.1	100	100	100	3	99.8	90.1	100	100	100	3	99.8	90.1	100	100	100	3	99.8	90.1	100	100	100
mcf	5	87.3	87.7	100	92.2	97.8	10	92.2	91.0	100	89.5	97.5	10	92.2	91.0	100	89.5	97.5	10	92.2	91.0	100	89.5	97.5
mesa	1	100	100	99.5	12.2	100	1	100	100	99.5	12.2	100	1	100	100	99.5	12.2	100	1	100	100	99.5	12.2	100
milc	2	79.2	72.1	37.1	27.4	93.9	18	83.0	72.8	100	52.0	99.7	21	78.1	66.8	100	52.1	99.7	21	78.1	66.8	100	52.1	99.7
parser	1	90.2	85.4	73.8	75.9	87.6	2	91.8	88.0	79.2	78.0	90.8	18	94.0	87.8	79.5	81.9	91.0	23	93.4	86.1	79.3	79.5	88.9
vpr	3	93.3	95.1	60.4	81.9	94.6	9	95.2	95.5	64.0	82.2	95.8	9	95.2	95.5	64.0	82.2	95.8	10	95.1	95.1	65.4	82.2	95.7
Average	2.4	92.9	82.4	79.3	69.0	92.5	8.7	95.0	89.0	90.3	75.5	95.5	13.0	95.0	89.6	90.8	77.3	96.2	16.6	95.0	89.2	90.9	77.7	96.1

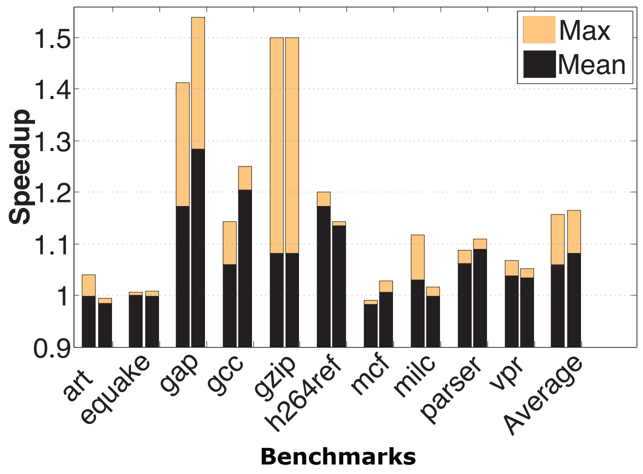

Figure 4. Speedups from profile-directed-feedback compilation using predicted profiles (left bars), and real profiles (right bars). The baseline is the performance by IBM XL compiler with the highest optimization level.

program control flow signatures and hardware-level performance (instructions per cycle, branch miss rates, cache misses, etc.). Our work shows that strong correlations also exist between different types of *program-level* behaviors. The correlations observed in previous work are mainly used for performance prediction and phase detection. The correlations revealed in this current work are important for guiding program optimizations.

Seminal Behaviors for Cross-Input Adaptation Seminal behavior is a new concept introduced in this work. It allows the proactive prediction of a large scope of behaviors, while also supporting cross-input adaptation.

Previous work in cross-input adaptation mainly falls into two categories. The first category treats program inputs explicitly; those studies manually specify a set of input features that are important for the execution of the application, and then use search or machine learning techniques to derive a model to help the execution of the application adapt to those features in an arbitrary input. Examples include the parametric analysis for computation offloading [17], machine learning-based compilation [12], adaptive sorting [13]. Because of the required manual efforts, those explorations have been focused on some particular applications. A recent work [14] proposes the use of specification languages for alleviating the efforts for general programs, but it still needs certain manual specifications. Seminal behaviors offer a way to automatic cross-input adaptation.

The second category of prior work includes the large body of runtime adaptive optimizations (e.g., [4, 7, 8, 11, 16, 18]). These techniques deal with input-sensitive behaviors implicitly through runtime sampling-based reactive behavior prediction. In contrast, seminal behaviors offer proactive predictions. The comparison has been discussed in Section 1.

7. Conclusion

In this paper, we have described a systematic exploration on the correlations among program-level behaviors, and introduced the concept of seminal behaviors for program behavior prediction. By employing a set of statistical learning techniques, we empirically demonstrate the existence of seminal behaviors for a variety of benchmarks. We evaluate the effectiveness of seminal behaviors in predicting a set of other kinds of program behaviors on both code and data levels. The results suggest that seminal behaviors are promising in automating input-based behavior prediction, offering new opportunities for the advancement of dynamic program optimizations.

Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful comments on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM Corporation.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[3] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2004.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, MN, October 2000.

[5] M. Arnold, A. Welc, and V.T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *the Conference on Object-Oriented Systems, Languages, and Applications*, 2005.

[6] P. Berube and J. N. Amaral. Additional inputs for SPEC CPU2000. http://www.cs.ualberta.ca/Ẽberube/compiler/fdo/inputs.shtml.

[7] W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of PLDI*, 2006.

[8] B. Childers, J. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proceedings of NSF Next Generation Software Workshop*, 2003.

[9] P. Chuang, H. Chen, G. Hoflehner, D. Lavery, and W. Hsu. Dynamic profile driven code version selection. In *Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, 2007.

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

[11] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of PLDI*, 2006.

[12] H. Leather, E. Bonilla, and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009.

[13] 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.

[14] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machine. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[16] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 93–102, Snowbird, Utah, June 2001.

[17] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 119–130, 2004.

[18] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In *PAC2 Conference on Power/Performance Interaction with Architecture, Circuits, and Compilers*, 2004.