An Input-Centric Paradigm for Program Dynamic Optimizations

Kai Tian Yunlian Jiang Eddy Z. Zhang Xipeng Shen

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

Abstract

Accurately predicting program behaviors (e.g., locality, dependency, method calling frequency) is fundamental for program optimizations and runtime adaptations. Despite decades of remarkable progress, prior studies have not systematically exploited program inputs, a deciding factor for program behaviors.

Triggered by the *strong* and *predictive* correlations between program inputs and behaviors that recent studies have uncovered, this work proposes to include program inputs into the focus of program behavior analysis, cultivating a new paradigm named input-centric program behavior analysis. This new approach consists of three components, forming a three-layer pyramid. At the base is *program input characterization*, a component for resolving the complexity in program raw inputs and the extraction of important features. In the middle is *input-behavior modeling*, a component for recognizing and modeling the correlations between characterized input features and program behaviors. These two components constitute input-centric program behavior analysis, which (ideally) is able to predict the large-scope behaviors of a program's execution as soon as the execution starts. The top layer of the pyramid is *input-centric adaptation*, which capitalizes on the novel opportunities that the first two components create to facilitate proactive adaptation for program optimizations.

By centering on program inputs, the new approach resolves a proactivity-adaptivity dilemma inherent in previous techniques. Its benefits are demonstrated through proactive dynamic optimizations and version selection, yielding significant performance improvement on a set of Java and C programs.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

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

General Terms Languages, Performance

Keywords Program inputs, Dynamic optimizations, Java Virtual Machine, Proactivity, Seminal behaviors, Dynamic version selection, Just-In-Time compilation

1. Introduction

The goal of program behavior analysis is to uncover the patterns in a program's dynamic behaviors (e.g., cache requirement, function calling frequency) so that the future behaviors of the program can be accurately predicted. As program optimizations rely on accurate predictions of program behaviors, program behavior analysis is essential for the maximization of computing efficiency.

The *inputs to a program* refer to all the data that are not generated but accessed by the program, including command line arguments, interactively input data, files to read, and so on. Many studies have reported strong influence program inputs impose on the program's behaviors [6, 21, 25, 27, 28, 40, 42]. Such influence has been commonly regarded a hurdle for program optimizations: Static compilers have to optimize conservatively through transformations that fit all possible inputs [1, 2]; profiling-based optimizers often encounter cases that an optimization they apply based on some training runs work inferiorly on an execution of the program on a new input [5, 6, 27, 28].

The work described in this paper comes from a different perspective: The strong influence from program inputs, although causing challenges, may meanwhile provide *valuable hints and opportunities* for program behavior prediction and program optimizations. The rationale is that because of the decisive role of program inputs, the knowledge about them may offer important clues on how the program would behave, and because in many cases (although not always) program inputs become known when an execution starts, the clues they offer may help produce a large-scope prediction of the execution at its early stage, offering important guiding information for dynamic optimizers. In the experiment to be reported in Section 4.1, for example, the clues indirectly derived from program inputs lead to accurate prediction of

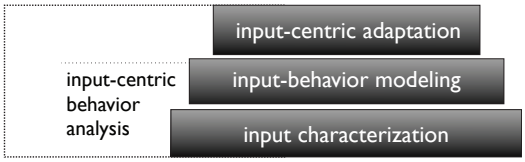

Figure 1. The components of input-centric program behavior analysis and optimizations. They form a pyramid, with the top level exerting the power of the analysis conducted by the lower levels.

the appropriate optimization level for each Java method at the early stage of an execution. The prediction helps the JIT (Just-In-Time) compiler appropriately optimize the Java method earlier than it does by default, yielding 10–29% performance improvement.

A few previous studies [25, 40, 43] have used certain features of program inputs for optimizations. But they have concentrated on several scientific kernels (e.g., sorting, FFT), and the exploitation of program inputs mainly relies on domain-specific knowledge (e.g., the data distribution is a feature important for sorting). A recent work [27] attempts to generalize the idea to a broader class of applications, but still with manual characterization of inputs required. It remains an open question how to exploit program inputs for optimizing general applications in a systematic and automatic manner. As a result, program input, a deciding factor for program behaviors, remains outside the focuses of most optimizers.

In this work, we aim to bring program inputs into the center of program optimizations by answering the following questions.

- First, is it worthwhile? In another word, are there any distinctive opportunities that an exploitation of program inputs can bring to program optimizations?
- If so, how to expose such opportunities, especially given the remarkable complexities of program inputs and program behaviors?
- Finally, if those opportunities can be exposed, how to capitalize on them for performance improvement? What changes need to be done to optimizers for the capitalization? How much performance improvement can be produced?

Our efforts for answering these questions yield a set of techniques forming a three-layer pyramid as shown in Figure 1. The bottom layer, *program input characterization* (Section 3.1), is fundamental. It extracts important features from raw, often complex, program inputs by taking advantage of statistical correlations among program behaviors. The second layer, *input-behavior modeling* (Section 3.2), recognizes and models the statistical relations between the features produced by the first layer and various program behaviors. The process is based on machine learning theory and techniques, with a systematic treatment to some special features of program behavior analysis (e.g., identification of categorical features, the tension between many input features and limited training runs). These two layers constitute *input-centric program behavior analysis*, through which, the runtime system is able to predict from the program inputs the behaviors of a large scope of the current execution, (ideally) as soon as the execution starts. The prediction opens many novel opportunities for the enhancement of dynamic optimizations. The third layer, *input-centric adaptation* (Section 3.3), helps overcome some inherent limitations—specifically, a proactivity-adaptivity dilemma—in current dynamic optimizers and converts the opportunities created by the first two components into performance improvement.

These three layers together form a new paradigm, namely the input-centric program behavior analysis and optimizations. Its central theme is the exploration and exploitation of program inputs. The techniques developed in this work address the key difficulties in each of its components, eliminate the needs for manual efforts, and for the first time, make automatic input-centric program optimizations feasible and profitable.

To examine the potential of the new paradigm, we apply it to two optimizers (Section 4). One is the JIT optimizer in Jikes RVM for Java programs, the other is the optimizer in a product compiler (IBM XL compiler) for C programs. Both experiments show that input-centric optimizations consistently outperform existing techniques. In the Jikes RVM, the speedups are up to 81% with average ranging from 10% to 29%; in the IBM XL compiler, the speedups are up to 58% with averages between 5% and 13%.

In summary, this work makes the following major contributions.

- It develops the first input-centric paradigm for program behavior analysis and optimizations. Some recent studies [21,27] have tackled certain challenges in some layers of the input-centric paradigm, offering the basis for this study. But none of the previous studies has proposed such a paradigm, or offered a completely automatic solution to realize the paradigm.
- This work systematically explores the special challenges existing in the construction of the statistical models between input features and program behaviors. Some related studies have employed machine learning tools for program optimizations but without considering the special properties of program behavior analysis. This work demonstrates that a systematic treatment to these properties may significantly improve the learning results and yield substantial enhancement to the optimization results.

- This work demonstrates input-centric adaptation by developing two example schemes. Meanwhile, it assesses the potential of input-centric optimizations by comparing them with both manual endeavors and the state-of-art optimization techniques.

2. Qualitative View on the Importance of Program Inputs for Program Optimizers

Before describing the techniques in detail and reporting their quantitative results, we first give a qualitative discussion to convey some intuition for the importance of program inputs and the distinctive opportunities they may bring to program optimizations.

2.1 A Deciding Factor for Program Behaviors

The importance of program inputs for program optimizations stems from their important role in determining program behaviors. Formally, *program behaviors* in this paper refer to the operations of a program and the ensuing activities of the computing system in relation to the program input and running environment. Examples include dynamic call graphs, data access patterns, memory requirement, cache usage, and so forth. The various factors deciding the behaviors of a program may be qualitatively expressed by the following program behavior equation:

$$Prog.\ Behaviors\ =\ Inputs + Code + Environments. \quad (1)$$

The program code determines the set of instructions that may be executed in a run; the running environments consist of all the elements in the execution platform, including the OS, virtual machine, architecture, system workload, and so on; program inputs determine the exact set of instructions to be executed, their execution order and frequencies, as well as the data to be accessed.

As shown by the behavior equation, for a given program in a given environment, program inputs are the single important factor that decides the behaviors of the program in the execution. Many quantitative measurements have confirmed this strong connection on various kinds of program behaviors, including data locality [47], sorting algorithm selection [25,40], computation offloading [42], and memory management [28]. This connection is the intuition for program inputs to serve as clues for program behaviors prediction—the essence of input-centric program behavior analysis.

2.2 Implications to Program Optimizations

Conceptually, the benefits of exploiting program inputs for optimizations may be summarized as its potential to address a proactivity-adaptivity dilemma that limits existing program optimizers.

Existing approaches to program behavior analysis fall into three categories as illustrated in Figure 2. Static compilation [1, 2] focuses on code analysis, considers certain runtime environments (e.g., the number of registers), but mostly ignores inputs. They conservatively limit themselves to the properties holding for all inputs.

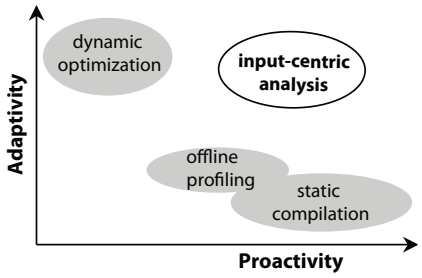

Figure 2. Insufficient treatments to program inputs causes a dilemma between the proactivity and adaptivity of program optimizations.

Offline profiling-based techniques typically choose several inputs as representatives for profiling and optimize the program accordingly. Their optimizations are limited to the behaviors exposed in the profiling runs, hence impairing their adaptivity to new inputs.

Finally, runtime behavior analysis [4, 8, 24, 31, 46] in dynamic optimizers (e.g., the runtime systems of Java and C#), overcomes the limitations of static and profiling techniques by sampling and analyzing program executions on the fly. It has good *adaptivity*—being able to adapt to the changes in running environments and program inputs. But it does not model or exploit program inputs: It simply uses the observed behaviors in a recent interval as the prediction for the future. As a result, runtime behavior analysis lacks the *proactivity* that the static and offline profiling techniques have—referring to that they analyze and predict the behaviors of the entire program before the start of any production run of the program.

The importance of proactivity may be less straightforward than that of adaptivity, but no less important. Its absence in existing dynamic optimization systems has resulted in three limitations. *First*, without a large-scope prediction of the behaviors of the current execution, an optimizer has to go through a behavior-monitoring phase periodically to learn about the execution before optimizing it. The delay impairs the benefits the optimizations may bring. *Second*, what the reactive way to learn about program behaviors regards is the execution in only some recent intervals. Consequently, the corresponding optimizations may be suitable to those intervals but inferior to the entire execution. For example, some Java methods that are heavily used in initialization stages may be rarely invoked in the main execution. Without a large-scope view, the JIT compiler in existing Java Virtual Machines may be misled to optimize those methods sophisticatedly, bringing virtually no benefits to the main execution, but considerable slowdown to the start-up [17]. *Finally*, the lack of proactivity limits the applicability of dynamic optimizations. For instance, a reactive way to dynamically

overcome the difficulty in making decisions for optimizations (e.g. unrolling levels for a loop, registers to spill) is to generate a version for each possible option and then try them one by one to select the best. The approach is hard to apply when the option space is large as the whole process happens during the current run. Even for small option space, the approach may have limited effectiveness especially when the segment of code (e.g. a subroutine) to be optimized has few invocations (as confirmed in Section 4.2).

Prior studies have revealed some evidences to those drawbacks of reactivity. In a study by Arnold and others [5], on a commercial Java Virtual Machine (IBM J9 [16]), programs may run 17-49% faster if the delay in optimizations is just partially removed. A later study [27] shows that enhanced proactivity increases performance even further. Other studies have seen similar benefits in memory management [28], locality phase prediction [35] and library development [25,40].

As a side note, the limitations of reactive approaches are not only for dynamic program optimizations, but also for dynamic adaptations in other levels, including operating systems and architectures. For instance, reactive co-scheduling [38] on chip multiprocessors requires the trials of possible co-runs (multiple jobs running on a single chip) to find the schedule that minimizes the effects of shared-cache contention. It is hard to scale as the number of possible co-runs is exponential in the numbers of jobs and computing units [20]. By providing a proactive way for large-scope program behavior prediction, the techniques presented next are potentially beneficial to those levels as well. Detailed discussions are out of the scope of this paper.

3. Input-Centric Behavior Analysis and Optimizations

Driven by the potential benefits of exploiting program inputs for optimizations, we have developed a set of techniques encapsulated in the pyramid in Figure 1. They are designed to tackle some critical obstacles to input-centric optimizations, including the characterization of complex program inputs, the construction of predictive models mapping from input features to program behaviors, and the capitalization of these models for program optimizations. This section presents the three layers of the pyramid in a bottom-up order.

3.1 Input Characterization

One of the major hurdles to exploiting program inputs is their complexity. An application may allow hundreds of options; those options may overshadow each other; input files may contain millions of data elements, organized in complex structures and representing various semantics (e.g., trees, graphs, videos).

The goal of input characterization is to address these complexities in a systematic and fully automatic manner. Specifically, it tries to reduce the raw, complex program inputs to a set of features. These features critically determine the behaviors of the program that are essential to its performance. Consider the GNU compression tool, *Gzip*. Its core includes a loop that applies Lempel-Ziv coding to a 32 KB segment of the input file in each iteration. Although the coding results and some fine-grained behaviors may differ on different input files, the major behaviors (loop trip-counts, function calling frequencies, etc.) of the program do not as long as the input files are of a similar size and some critical compression options remain the same [34]. Similarly, for a quick-sort program, the distribution rather than the values of input data determines the sorting process [25, 40]. As to be presented in Section 4, our examination of 10 Java and 14 C applications shows that for most of them, it is enough to predict their main behaviors from a small number of input features.

However, automatically extracting the critical features directly from program inputs is a remarkably challenging task. An input data file may have arbitrary structures and semantics, ranging from a graph to an audio to a database or even a program. It may have a large number of attributes, from as simple as the values of some special numbers in a file to as concealing as the density of a graph, the frequency range of an audio signal, the distribution of a bag of data, and the numbers of various constructs in a program.

In this work, we employ a technique, named seminal-behavior analysis, to circumvent the difficulties. Seminal-behavior analysis is a technique proposed recently by Jiang and others [21]. It is enlightened by the strong correlations among program behaviors. Such correlations are statistical properties. For instance, in five runs of the program *mcf* on five different inputs, one of its loops (denoted as loop-1) has iterations (15, 41, 52, 89, 101), and another (denoted as loop-2) has iterations (69, 173, 217, 365, 413). Statistical analysis can easily determine that the trip-counts (i.e., the numbers of iterations) of these two loops have a linear relation as $C_2 = 4 * C_1 + 9$ (C_1, C_2 for the trip-counts of the two loops). Jiang and others show that such statistical correlations widely exist both among loop trip-counts and from loop trip-counts to other types of behaviors, including function invocations, data values, and so on.

Based on those observations, they developed a three-step automatic approach to recognizing a small set of behaviors in a program, named a seminal behavior set. These behaviors satisfy two properties. First, they have strong correlations with many other behaviors in the program so that knowing their values would lead to accurate prediction of the values of other behaviors. Second, the values of those behaviors become known in an early stage in a typical execution of the program. The seminal behavior set of the program *mcf*, for example, is composed of 10 behaviors: the trip-counts of five of its loops, the values of four of its variables whose values come directly from command line arguments or input files, and its input file size. In all measured runs, the values of most seminal behaviors become explicit during the first 10%

portion of an execution. Their values show strong statistical correlations with the trip-counts of most loops and the calling frequencies of most functions in the programs.

The entire process for finding seminal behaviors is through a fully automatic tool; no manual efforts are necessary. The tool is composed of a compiler (specifically, a modified GCC [21]), a profiling component, and a data analysis component. Given a program and a set of inputs, during the compilation stage, the compiler inserts some instructions into the program to prepare for program behavior collection. Then the profiling component runs the program on each of the inputs; the behaviors reported by the inserted instructions form a database. After the profiling step finishes, the data analysis component analyzes the content of the database (using regression techniques) to examine the statistical relations among observed behaviors, and then determines which of those behaviors may serve as seminal behaviors based on their earliness (i.e., how early their values become known in an execution) and the strength of their correlations with other behaviors. The previous paper [21] contains the formal definition of seminal behaviors and the detailed design and implementation of the tool.

In this work, we adopt seminal behaviors for program input characterization. The basic rationale is that because seminal behaviors have strong correlations with many other behaviors, their values essentially capture the important features of the current program inputs and offers clues for program behavior prediction. For instance, in the 2-loop example mentioned in the earlier paragraph, suppose loop-1 is identified as a seminal behavior. In a new execution, as soon as its loop trip-counts (C_1) become known, we may immediately predict the trip-counts of loop-2 by plugging C_1 into the linear equation $C_2 = 4 * C_1 + 9$.

By using seminal behaviors, we avoid the needs for direct attacks to the complexities in program raw inputs. It offers an indirect way to characterize program inputs in a fully automatic manner.

3.2 Input-Behavior Modeling

Input-behavior modeling is the second component of input-centric behavior analysis. Its goal is to construct models, namely *input-behavior models*, that capture the connections between input features—represented by seminal behaviors—and program behaviors. With such models, the prediction of program behaviors from an arbitrary input becomes possible.

The modeling is through cross-run incremental learning. For a given application, during each of its executions, the runtime system records the values of seminal behaviors and (sampled) program behaviors in a database. After a certain number of runs, a learning agent applies statistical learning to the database to construct input-behavior models. For a run on a new input, the runtime system uses the constructed models to predict how the program will behave, preparing for proactive dynamic optimizations. The learning occurs repeatedly for continuous enhancement of the models.

In this section, we first describe the formulation of the problem of input-behavior modeling as a statistical learning problem and outline the basic solutions. We then concentrate on several challenges in the learning process that are special to input-behavior modeling and describe our answers. Section 4 will show that treating these special challenges is critical for the quality of the produced input-behavior models.

3.2.1 Problem Formulation

The input-behavior modeling is a statistical learning process. Its objective is to determine a function that maps inputs, characterized by seminal behaviors (denoted by V), to target behaviors (denoted by B). The mapping function is represented as $B = f(V)$, where V is a vector with each element corresponding to one seminal behavior. The learning target, B, can be one behavior or a vector of multiple target behaviors. In the latter case, the learning process builds a mapping function between V and each of the target behaviors.

During the learning process, on every run of the program on an input data set, we obtain the values of both V and B. After a number of runs, we accumulate a database $\{< V_i,\ B_i >\}$ $(i = 1, 2, \cdots, N)$. Determining the function f from such a database is a typical statistical learning problem. Specifically, when the target behavior has categorical values (i.e., its value set has a limited number of members), the problem is a classification problem; when the target behavior has continuous values, it is a regression problem [18]. The function f can be in a form of mathematical formulas or in a less structured form, such as Decision Trees, Support Vector Machines, Neural Networks.

To give a concrete explanation, we take the compilation of Java methods in Jikes RVM [3] as an example. In Jikes RVM, the JIT compiler may optimize a Java method at 4 levels (-1, 0, 1, 2). Due to the tradeoff between compilation time and execution time, the appropriate optimization level differs for different methods. Moreover, for a specific method, the best level may vary across program inputs. To apply input-centric analysis and optimizations to this example, we may build a model between program input features and the appropriate optimization level for each Java method. So, in this example, B is not a single behavior, but a set of target behaviors, each member of which is the appropriate optimization level of a Java method. Correspondingly, f is a set of functions, with each member mapping from input features to one member of B. Because optimization levels are categorical, all functions in f are classifiers, the determinations of which may proceed independently from one another.

3.2.2 Classification and Regression

Many classification and regression methods are applicable to input-behavior modeling. In this work, we select Decision Trees as the primary approach for both classification and regression, because of its simplicity and other appealing

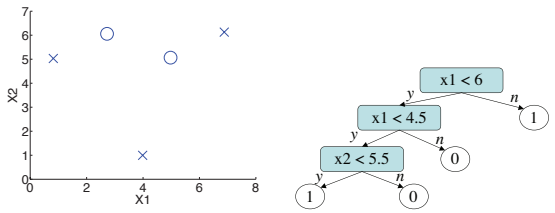

Figure 3. A training dataset (o: class 0; x: class 1) and the classification tree. Each leaf node is labeled with a class number.

properties to be listed at the end of this section. The specific forms of Decision Trees for classification and regression are named Classification Trees and Regression Trees respectively.

A classification tree is a hierarchical data structure implementing the divide-and-conquer strategy [18]. It divides the input space into local regions, and assigns a class label to each of those regions. Figure 3 shows such an example. Each non-leaf node asks a question on the input features. Each leaf node has a class label. For a new input, going through the tree with its features would lead us to a leaf node. The label of that leaf node is the prediction on the new input. The key in constructing a classification tree is in selecting the appropriate questions to ask in each non-leaf node. The goodness of a question is quantified by purity: A question is desirable if after the split based on the question, the data in each subspace has the same class label. Many techniques have been developed to automatically select the appropriate questions based on the entropy theory [18].

A regression tree shares with a classification tree in the form of the data structure and the construction process. The primary differences exist in the calculation of purity and the final label of a leaf node. Both differences are due to the numerical rather than categorical values of the learning targets. By default, a leaf node in a regression tree is labeled as the average value of all the instances contained in the node. In our implementation, we add an extension to enhance the predictive capability by applying Least Means Square (LMS) to the instances in the leaf node to produce a linear function of the input features.

We adopt Decision Trees as the main learning technique for its five-fold appealing properties. *First*, the construction and use of Decision Trees are simple, efficient, and fully automatic. *Second*, Decision Trees has excellent interpretability. Unlike Neural Networks or other learning models working as a black box to users, a decision tree is essentially a set of "if...else..." rules. This interpretability can not only help users analyze and verify the results easily, but also increase the understanding to the problem. *Third*, Decision Trees handle both categorical and numerical input features smoothly. Both kinds of features are common in program behavior analysis. *Fourth*, its tree structure is a natural match with non-linear relations. The combination with linear regression techniques (e.g., LMS) makes it capable to handle various relations between input features and the learning targets. *Finally*, it automatically selects important input features. It is possible that some input features are either constant across all inputs or irrelevant to the learning targets. During the construction of decision trees, because questions on those features cause no impurity reduction, they will not appear in the trees. The reduction of features helps reduce the dimensionality of the problem, preventing certain noises from hurting the quality of the constructed decision trees.

Our strategy for the statistical leaning is a two-level strategy. For a given target behavior, before applying Decision Trees, we first try LMS to fit the training data set with polynomial (linear or quadratic) functions. Using cross validation [18] (i.e., part of data for training and others for testing), we estimate the quality of the fitting. We resort to Decision Trees only when the linear models cannot fit the data well; such cases typically suggest the existence of non-linear relations. The entire learning process involves no manual intervention.

3.2.3 Challenges Special to Input-Behavior Modeling

Input-behavior modeling imposes several special challenges to the classification and regression techniques. A systematic treatment to these challenges in the modeling process turns out to be critical for the quality of the modeling results.

Categorical Features. The first challenge is on the presence of categorical features, referring to the features whose values can only be one of a limited number of values. Examples include the optimization levels in Jikes RVM, the options that control which compression algorithm to select in a compression tool, the type of an input file, and so forth. In many statistical learning problems, categorical features are marked beforehand along with their value ranges. But in input-behavior modeling, such knowledge typically does not exist. Consequently, special operations are necessary for identifying and utilizing such features during the statistical learning process.

For identification of categorical features, we exploit two heuristics. First, if the data type of a feature is not a number (integer, float, etc.), the feature is considered as categorical. Otherwise, if the number of unique values of the feature contained in the training data set is smaller than a threshold, the feature is considered as categorical as well. The threshold is defined as 10% of the total number of the occurrences of the feature in the training data set. The intuition is that the large number of repetitive values suggest that the value of the feature in a new run is likely to be one of its values that have appeared in the training runs. So even if it is not categorical, treating it as a categorical feature would typically work well in terms of the predictive capability of the produced model. Note, if the new value happens to be something not covered by the training set, the risk control (to

be presented) would prevent the mistaken predictions from causing inferior consequences.

During the model construction process, we deal with categorical features via the use of indicator matrices [18]. A feature with k categories, is converted to a vector of $k-1$ binary features. If an observed value of this feature is the ith ($i = 1, 2, \cdots, k-1$) category, the ith binary feature is set to 1, and all the others are set to 0. When the feature value equals the kth category, all $k-1$ features are set to 0.

Feature Selection. The second special challenge resides in the tension between the many seminal behaviors and the limited number of training runs. A large number of seminal behaviors form a high-dimensional input space with each seminal behavior as one dimension. Learning in such a space demands a large number of training data. However, collecting many representative inputs and then conducting profiling runs are time consuming and not always feasible.

We alleviate that tension through feature selection techniques. As mentioned earlier in this section, Decision Trees can automatically filter out unimportant features. Moreover, during the LMS regression, we apply a standard stepwise method to further filter out important features. The method works as follows. It first builds an initial model based on the observations in the training runs with a minimum number of features included in the model. It then examines each feature that does not show up in the current model. It adds a feature into the model only if it finds out that the addition improves the predictive capability of the model substantially (evaluated through F-statistic analysis [18]). This process continues until no more features can be added. In addition to the feature reduction, a risk control scheme (presented next) helps reduce the tension between the number of features and the number of training runs as well. The scheme distinguishes the subspaces in the input space that are predictable from those that are not, hence pruning the learning complexity.

An alternative to the stepwise method for feature selection is the Principle Component Analysis (PCA). It casts features to the orthogonal axes of a principle component space and selects only those directions along which the values of the data show large variations. The use of PCA allows no presence of categorical features. We apply PCA only when all features are numerical, and use the stepwise method otherwise.

Risk Control and Model Evolvement. It is important to prevent wrong predictions from hurting program optimizations. Discriminative prediction is an approach proposed in a recent work [27] for risk control. The learner keeps assessing the confidence level of the input-behavior models and predicts only if the confidence level is higher than a preset threshold. (The optimizer falls back to the default reactive strategy when the confident is low.) The confidence is measured through cross-validation on the collected behaviors of history runs stored in the database. This risk control is coarse-grained in that the whole program has only one confidence value, regardless of the variations in the input feature space and the behaviors to predict.

In this work, we extend the approach to allow fine-grained control. The motivation is that the predictive capability of a model often varies in different regions in the input feature space. It also depends on the behaviors to predict. The fine-grained risk control maintains a confidence value for each input sub-space to capture such differences. When Decision Trees are constructed in the model training process (Section 3.2), the tree leaf nodes, along with the value ranges of training seminal behaviors, form the input sub-spaces. Otherwise, there is only one subspace, outlined by the value ranges of all seminal behaviors in the training data set. All confidence values are compared against a single predefined confidence threshold (0.7 in our experiments, the same as previous work uses [27]). If a new input falls outside of the confident sub-spaces, the prediction is shut down automatically and the system falls back to the default optimization scheme.

The behavior models and the confidence values may evolve on newly conducted training runs. The observed seminal behaviors and target behaviors in a new run may be added into the training data set so that the safe region can be continuously expanded. The models may be retrained using the expanded data set, with the confidence levels updated accordingly.

3.3 Input-Centric Adaptation

The large-scope prediction of program behaviors, enabled by input-centric behavior analysis, opens many new opportunities for program optimizations. We name the new way of optimization *input-centric adaptation*. Two features, proactivity and holism, distinguish input-centric adaptation from existing dynamic optimizations.

3.3.1 Proactivity and Holism

Being proactive means that the optimizations happen at the early stage of a program's execution, no need to wait for the finish of periodical monitoring phases. The direct advantage is that it can determine and apply suitable optimization decisions early, avoiding the optimization delays in reactive schemes. In addition, it expands the applicability of dynamic optimizations to the scenarios where reactive schemes cannot work well, such as the job co-scheduling problem mentioned in Section 2.2.

The holism has two-fold meanings. On the program level, it means that input-behavior models predict the behaviors of the entire program (or a large portion of it) rather than a small window of the execution. With that view, optimizers may make more accurate decisions so that many issues, such as the JIT compilation problem mentioned in Section 2.2, can be resolved. The second meaning of the holism is that the large-scope proactive behavior prediction makes cross-layer optimization more feasible than before. The predicted

behaviors may be passed across software execution layers, facilitating the coordination among compilers, OS, and virtual machines. Detail in this cross-layer aspect is yet to be explored in our future work.

3.3.2 Example Uses

In this section, we describe two uses of input-centric adaptation to explain how it may be integrated into existing dynamic optimizers. The next section reports the resulting performance improvement, showing the quantitative benefits brought by the proactivity and holism of the input-centric adaptation.

Use 1: JIT Optimizations in Jikes RVM. The first use is on the enhancement of Java program performance through JIT. We implement a prototype of seminal-behavior-based proactive dynamic optimizations in Jikes RVM [3]. Like most Java Virtual Machines, Jikes RVM is reactive: During an execution, the RVM observes the behaviors of the application through sampling, whereby, it determines the importance of each Java method in the application, and invokes the JIT compiler to (re)optimize the method accordingly. As compilation incurs runtime overhead, the JIT in RVM offers four compilation levels. The high-level optimizations (more sophisticated and hence taking more time) are supposed to be used only for important Java methods, and low-level optimizations for others.

Complexities reside in the determination of the importance of a method. At each time point, Jikes RVM assumes that the time a method will take in the rest of the execution is the same as the time it has already taken; the longer that time is, the more important the method is. In reality, this assumption often does not hold, causing inaccurate prediction of the importance. The JIT compiler may be hence misled to optimize an unimportant method sophisticatedly. On the other hand, RVM may not recognize the importance of a really critical method until the late stage of the execution due to the reactivity of the scheme. As a result, the compiler may compile the method multiple times in increasing levels gradually, rather than in the highest level at the early encounter of the method. These issues cause extra compilation overhead and impair the effectiveness of the dynamic optimizations.

We integrate input-centric behavior analysis into Jikes RVM 2.9.1 to enable proactive dynamic optimizations. First, we identify seminal behaviors through offline training and build the models between seminal behaviors and the appropriate optimization levels for every Java method in a program. The appropriate optimization levels used in the training process come from the default cost-benefit model in the Jikes RVM. The cost-benefit model determines the appropriate optimization level from the hotness (i.e., invocation times and length) of a Java method. By feeding the cost-benefit model with the hotness of all Java methods obtained at the end of a training run, we get the appropriate optimization levels to be used in the training process.

After that, as soon as the values of seminal behaviors become explicit in an execution, Jikes RVM can plug those values into the constructed predictive models to predict the appropriate optimization levels of the Java methods (if the input falls into the safe regions and the confidence level is high enough). Specifically, when a Java method is encountered for the first time, it is compiled using the basic optimizer (at the lowest optimization level), but meanwhile, a recompilation event is pushed into the Jikes RVM event queue so that the method will quickly be recompiled at the level predicted from the seminal behaviors. (Not using the predicted level for the first-time compilation is to avoid immature optimizations because many references are possibly not resolved yet. This scheme is consistent with those used in previous studies [5, 27].)

Compared to the default dynamic optimization schemes in Jikes RVM, the new approach avoids unnecessary recompilations and tends to generate high-quality code at the early stage of the program execution. Section 4.1 reports the resulting performance improvement.

Use 2: Proactive Dynamic Version Selection. The second use is on C program optimizations through dynamic version selection. Dynamic code version selection is a technique for enabling the adaptation of program optimizations on input data sets. Our study is particularly based on the work by Chuang and others [10]. In that work, for each function, the compiler generates several versions using different optimization parameters. During runtime, 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 technique shows promising results. But as the authors point out, the way a version is selected is subject to some limitations. First, the timed execution of the different versions may operate on different parts of a data set, causing unfairness in the comparison, and leading to an inferior selection result. Second, the technique is unlikely to benefit the functions that have only a few invocations, because of the requirement of runtime trials. This limitation is especially serious when such functions contain large loops and dominate the entire execution time.

Seminal behaviors offer a solution to these issues. The key insight is that because seminal behaviors capture the dominant influence of program inputs on 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 new run as soon as the values of seminal behaviors become explicit in that run. In this way, we do not need the trials of the different versions during runtime, hence circumventing both limitations of the previous work. Details of the implementation will be presented in Section 4.2 along with the performance results.

Discussions. As a short summary, to apply input-centric analysis and optimizations to a program, the user needs to do the following two-step operations: prepare a set of program inputs and conduct a profiling run on each of the input, and then apply the input-centric analysis tool to the data collected during the profiling runs. During the second step, the tool automatically recognizes seminal behaviors and builds up predictive models that map from seminal behaviors to some behaviors of interest (e.g., calling frequencies). After that, when the program runs with a modified optimizer (e.g., the modified Jikes RVM in our implementation), the program will be automatically optimized in a proactive, dynamic fashion based on the predictive models.

Proactive optimizations do not conflict with the presence of program phase shifts. Pior studies [35,36] have shown the predictability of phase shifts. The proactive optimizations in input-centric adaptation can thus be applied before each phase.

Despite that proactive dynamic optimizations overcome some limitations of existing (reactive) optimization schemes, we view the proactive scheme as a complement rather than a replacement to existing dynamic optimizations. The runtime sampling in existing reactive schemes is important for the assessment of the quality of the behavior prediction, and the reactive optimizations serve as a fall-back option when the proactive schemes cannot work well. Moreover, in the scenarios where cross-run learning is not desirable for overhead or other reasons, reactive schemes are especially valuable.

4. Evaluations

In this section, we report the results of the two input-centric adaptation techniques described in the previous section. The comparisons of these results with those from the default dynamic optimizers in an open-source Java Virtual Machine (Jikes RVM), a product compiler (IBM XL compiler), and previous manual endeavors, demonstrate the potential of the input-centric paradigm for program optimizations.

In all the experimental results, the input sets for training and testing have no overlaps. Specifically, we employ the standard cross-validation scheme [18] in all experiments. The scheme evaluates a predictive model iteratively. In each iteration, it takes out some inputs (e.g., 90% in our experiment) from the input sets for the training of the model and then uses the rest inputs for testing. The overall average of the prediction accuracies of all the iterations are taken as the final result.

The seminal behaviors used in all experiments are obtained in the way described in a previous study [21]. The earliness threshold is 0.9, meaning that the trip-count of a loop cannot serve as a seminal behavior if its value cannot be determined (directly or indirectly) in the first 10% of an execution. It is worth noting that because the trip-counts of many loops can be determined from their iteration upper-bounds and lower-bounds, they often become known much earlier than the executions of the loops finish.

4.1 Optimizations by JIT

In this section, we first report the effectiveness of seminal behaviors in predicting the behaviors of 10 Java benchmarks, and then present the performance improvement coming from the JIT compilations guided by the predicted behaviors.

Methodology. The machine we use is equipped with Intel Xeon E5310 processors, running Linux 2.6.22. All experiments use Jikes RVM as the virtual machine. Table 1 reports the used benchmarks, which come from three benchmark suites. They have been used in a previous study [27], in which, manually characterized input features are employed for proactive optimizations[1]. So using this set of benchmarks makes it convenient to compare with the previous results. In the previous work [27], the authors collected a number of extra inputs for each of the programs for their experiments. Those inputs are used in this current work as well.

Behavior Prediction Accuracy. The fourth to sixth columns in Table 1 report the accuracies of three types of behaviors predicted from seminal behaviors. We select these three types of behaviors for prediction because of their relevance to program optimizations and memory management. The first type is method calling frequency, a type of behaviors critical for inter-procedure optimizations (e.g., function inlining). The second type is minimum heap size, referring to the minimum size of the heap on which a Java program can execute successfully. This property is important for determining the heap pressure and has been used for the selection of garbage collectors [28,37]. The third type is the appropriate optimization level of each Java method, a key decision affecting the optimization results by the JIT compiler in Jikes RVM as mentioned in Section 3.3.

The results show that the seminal behaviors can predict most of those behaviors with over 94% accuracy. The rightmost two columns in Table 1 list the results from a previous work [27]. It uses manually characterized input features to help proactive dynamic optimizations. The numbers of features are not as large as the numbers of recognized seminal behaviors for half of the programs. The average accuracy of the predictions from seminal behaviors is 9% higher than the previous results. The higher accuracy comes from two sources. The first is that the recognized seminal behaviors characterize the input features better than those features manually produced. This is not surprising: Given the high complexity in some of those programs (especially those from JVM 98 and Dacapo suites) and their inputs, it is hard for manual characterization to determine all the input features that are critical to the programs behaviors. On the other hand, even when the manual characterization finds all im-

[1] We exclude one program named *fop* because of some implementation limitation of the current seminal behavior analysis tool.

Table 1. The Numbers of Seminal Behaviors and Prediction Accuracies

Program	# of inputs	Through seminal behaviors				Manual approach [27]	
		# of sem. beh.	Pred accuracy			# of features	Pred accuracy
			call freq	min heap	opt. level		opt. level
Compress[j]	20	2	0.93	0.99	0.99	1	0.94
Db[j]	54	4	0.98	0.96	0.84	2	0.86
Mtrt[j]	100	2	0.89	0.84	0.97	2	0.82
Antlr[d]	175	39	0.95	0.96	0.95	3	0.83
Bloat[d]	100	7	0.76	0.99	0.96	2	0.85
Euler[g]	14	1	0.99	0.98	0.99	1	0.91
MolDyn[g]	15	2	0.83	0.98	0.98	1	0.81
MonteCarlo[g]	14	1	0.98	0.99	0.99	1	0.83
Search[g]	9	2	0.97	0.99	0.99	2	0.96
RayTracer[g]	12	1	0.9	0.98	0.98	1	0.85
Average	51.3	6.1	0.92	0.97	0.96	2	0.87

j: jvm98; d: dacapo; g: grande

portant features (e.g., for most of the Grande benchmarks), the enhanced statistical learning algorithms, as described in Section 3.2, improves the prediction accuracy substantially by systematically handling categorical features and reducing the dimensionality of the input space through PCA and stepwise selection. These results demonstrate the importance of a systematic treatment to the special properties of the input-behavior modeling during the learning process.

Performance Comparison. We modify the Jikes RVM to enable proactive optimizations guided by the seminal-behaviors–based prediction, as described in Section 3.3. Compared to the default dynamic optimization scheme in Jikes RVM, the new optimization strategy saves compilation time by avoiding unnecessary recompilations of a method, and saves execution time by generating efficient code early.

We compare the resulting performance with the performance of the default executions and with the performance (denoted as "Manual") of the programs optimized [27] using the previously proposed proactive optimizations based on manually characterized input features.

Figure 4 shows the results. The baseline is the performance of the default executions. As the speedup differs on different inputs, each bar shows the average and maximum speedup of all the runs of a program. On average, the seminal-behavior–based proactive optimizations yield 10.2–29.4% speedup over the default executions. Because of the improved prediction accuracies, they outperform the *Manual* results by 1.9–5.3%. These results, for the first time, demonstrate that proactive optimizations based on automatic input characterization may produce even higher speedup than manual endeavors do.

4.2 Proactive Dynamic Version Selection

Our experiments of proactive dynamic version selection are based on the PDF (profile-driven-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, reflecting the dynamic behaviors of the program on basic block frequencies and function return values. The compiler then recompiles the application using the profiling results as feedback (through the option "-qpdf2") and generates a specialized executable for the input. We conduct this experiment on an IBM server equipped with Power5 processors and AIX v5.3.

In this experiment, as a preparation, for each program, we first select five representative inputs to do five independent PDF compilations, generating five versions of the program. During that process, we also record the values of the seminal behaviors in each of the five runs.

Next, we run the programs with different inputs. We collect following running times of the programs to compare the performance improvements of different version selection techniques.

- *default*: the default static compilation at the highest optimization level.
- *def-pdf*: the default PDF compilation. As default PDF compilation does not adapt to inputs, we obtain the performance of a program on an input by running each of the five versions on this input and getting the average running time of the five runs.
- *dyn-trial*: corresponding to the previous version selection technique [10]. The five versions of each function are tried in its first five invocations and the one with the shortest running time is used for the rest of the run. This scheme is a typical reactive scheme for runtime version selection.
- *dyn-sem*: seminal-behavior-based version selection. In each testing run, its seminal behavior values are compared against the seminal behaviors of the five training runs. The highest similarity determines which of the five versions will be used for the rest of the execution. The similarity comparison is in terms of Euclidean distance (normalized to remove the differences of value ranges among dimensions).
- *ideal*: the ideal case, in which, the real profile of a run is used for the PDF optimization of that run.

The performance in the first case, *default*, is taken as the baseline.

Methodology. Table 2 lists the programs we've used in this experiment. They include 14 C programs in SPEC CPU2000 and SPEC CPU2006. We include no C++ or Fortran programs because the instrumenter we implement currently works for C programs only. 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 critical for this study) difficult.

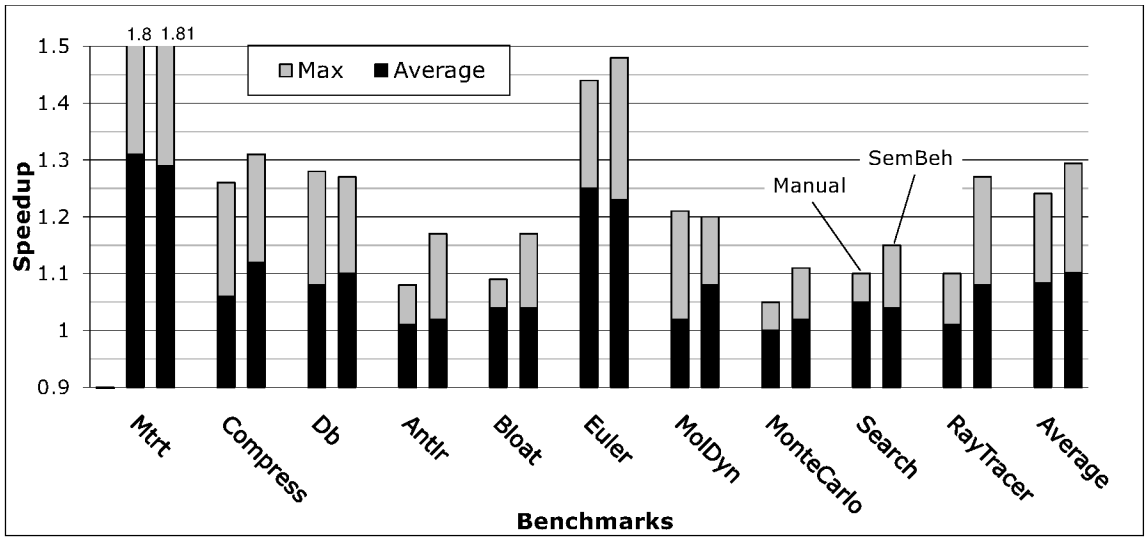

Figure 4. The speedup ranges produced by seminal-behavior–based proactive optimization (the "SemBeh" bars), compared to the default optimization scheme in Jikes RVM (the baseline) and the proactive optimizations based on manually characterized input features (the "Manual" bars).

Although each benchmark comes with several sets of inputs by default, more inputs are necessary for a systematic study of the predictability of seminal behaviors. We add extra inputs as shown in the third column of Table 2. During the collection, we try to ensure that the inputs are typical in the normal executions of the benchmarks. More 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 inputs come from the collection by Amaral's group [6]. The fourth column of Table 2 shows the largest changes of the loop trip-counts of those programs caused by the changes in inputs.

The numbers of seminal behaviors are shown in the fifth column. The rightmost three columns report the accuracies of the three types of profiles predicted from seminal behaviors. Most programs show reasonable prediction accuracy. Several programs show modest accuracies, mainly due to 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*. The branches result in complex non-linear relations between seminal behaviors and the basic block access frequencies, which explain the low prediction accuracy of its node profiles.

Performance Results. Figure 5 reports the speedups of the programs over the baseline performance. All runtime overhead are counted in. In each scenario, five repetitive measurements are conducted. Only negligible variances appear among the measurement results, and the average value is used.

Table 2. Benchmarks Used in Dynamic Version Selection and Prediction Accuracies

Name	# of lines of code	# of inputs	factors of changes caused by inputs	# of sem. beh.	accuracy edge	node	data
ammp	13263	20	9.9×10^1	1	100	91.1	99.7
art	1270	108	4.0×10^4	4	100	80.0	96.1
crafty	19478	14	4.6×10^8	2	90.8	44.5	79.3
equake	1513	100	1.0×10^2	1	100	96.3	99.3
gap	59482	12	1.1×10^8	7	56.3	69.7	88.5
gcc	484930	72	1.1×10^6	54	93.6	95.4	95.6
gzip	7760	100	4.3×10^7	6	83.5	69.0	94.5
h264ref	46152	20	2.1×10^9	4	97.0	97.8	99.7
lbm	875	120	6.0×10^5	3	100	100	100
mcf	1909	64	1.4×10^5	10	100	89.5	97.5
mesa	50230	20	2.0×10^1	1	99.5	12.2	100
milc	12837	10	2.1×10^9	18	100	52.0	99.7
parser	10924	20	2.1×10^6	2	79.2	78.0	90.8
vpr	16976	20	3.9×10^6	9	64.0	82.2	95.8

Because we have a number of inputs for a program and the speedups on them are different, each bar in Figure 5 shows the speedup range. The results reveal the following phenomena. *1)* The *ideal* results show that a large potential (5–15% average and up to 67% speedup over all programs) exists for using profiles as feedback to optimize those programs. *2)* However, due to the lack of adaptivity to different inputs, the *def-pdf* yields only 2–7% speedup averaged over all the programs. *3)* The *dyn-trial* improves the speedup to 3–8%, but still having a considerable distance from the optimal. As mentioned earlier in this section, only after five invocations of a function, *dyn-trial* can decide on a suitable version for that function. Measurement through *gprof* shows that the average portion of an execution that can benefit from *dyn-trial* is only 62%. For example, the top function in *gap*

is *__mcount*. Although it takes 41% time of an execution, it is invoked only once. This limitation explains the distance of the resulting performance from the full speed. *4)* By circumventing both issues facing *dyn-trial*, *dyn-sem* boosts the speedup to 5–13%, about 3% from the *ideal*. The essential reason for the promising results is that seminal behaviors make it possible to get both proactivity and cross-input adaptivity, gaining the strengths of both the offline profiling (*def-pdf*) and the runtime adaptation (*dyn-trial*).

Given that the baseline is a highly tuned commercial compiler and the speedups come from a fully automatic process, the results demonstrate the promise of input-centric dynamic optimizations.

As a side note, the results illustrate the imperfectness of the compiler implementation. On some programs (e.g., *h264ref*), sometimes the *ideal* case is worse than the *default* or other cases. This imperfectness is no surprise given the complexity and the use of heuristics in compiler implementation.

The majority of the overhead of our technique is on the training process, including the identification of seminal behaviors and the construction of predictive models. But since the training happens 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.

5. Related Work

Prior research in program optimizations falls into three categories in light of the treatment to program inputs. First, static compilation either limits itself to the properties holding for any input, or uses ad-hoc estimation for dynamic behaviors [1, 2]. For example, a loop is considered ten times more frequently accessed than others [11].

Second, offline profiling-based methods typically choose several inputs as the representatives to conduct profiling runs and optimize the program accordingly. Such empirical optimization methods have been adopted in the construction of some numerical libraries or kernels, such as ATLAS [43], PHiPAC [7], SPARSITY [19], SPIRAL [33], FFTW [14], STAPL [40]. However, the lack of cross-input adaptivity impairs the effectiveness of offline profiling-based techniques on optimizing programs with input-sensitive behaviors.

The third category includes run-time optimizations that transform a program during its execution. Some of them exploit runtime invariants through programmers' annotations or other efforts; examples include 'C from Kaashoek's group [32], Tempo from Consel's group [29], and DyC from Chambers and Eggers' group [15]. Others monitor execution through runtime profiling to optimize a program; examples include the dynamic feedback work by Diniz and Rinard [12], the continuous program optimizations by Kistler and Franz [22], the ADAPT project by Voss and Eigenmann [41], the CoCo project by Childers, Davidson and Soffa [9], the continuous program optimization (CPO) project by Wisniewski and his colleagues [44], the dynamic optimizations on LLVM [23], the runtime support for managed languages like Java and C# [4, 8, 24, 31, 39, 46]. These techniques observe runtime behaviors directly, and typically employ reactive optimization scheme. They do not treat inputs explicitly; their effectiveness is limited by the constraints discussed in Section 2.2. There has been some work that uses input features, such as the computation offloading by Wang and Li [42], the adaptive algorithm selection from Li and his colleagues [25] and from Rauchwerger and others [40]. Their explorations are mainly on a specific class of applications and use manually defined input features. Amaral and others have investigated the influence of inputs on benchmark design [6].

A recent work [27] also uses input features as the clue to predict the best optimization levels for a Java method to enable proactive optimizations. Our current study differs from the previous work substantially. To the best of our knowledge, this current work is the first that proposes and systematically develops the three-layer paradigm of input-centric program behavior analysis and adaptations. The previous work [27] is a case study showing the potential of cross-run information for Java method optimizations. It is preliminary in all the three aspects of the input-centric paradigm. First, for input characterization, it relies on programmers to provide a specification of input features, which adds extra burden to programmers and introduces possible errors. Second, it uses classification trees for input-behavior modeling but without systematically considering categorical features, feature selection (the stepwise method), and fine-grained risk control. Finally, for runtime adaptation, this current work explores optimizations for not only managed environments but also traditional imperative languages.

This current study draws on the observations on behavior correlations and the concept of seminal behaviors contributed by Jiang and others [21]. That previous work shows the existence of the correlations and seminal behaviors, but contains no use of them either for systematic input characterization, or for any kind of proactive dynamic optimizations. There are some other studies [30] that have exploited the connections between loops and method hotness. They mainly use loop upperbounds as heuristics, with no systematic exploration conducted into the statistical correlations among a broad range of program behaviors.

In the realm of software testing, there has been a body of work on input specification and generation. Those techniques focus on the interface to program modules such as procedures or classes. They do not characterize the hidden attributes for the prediction of runtime behaviors.

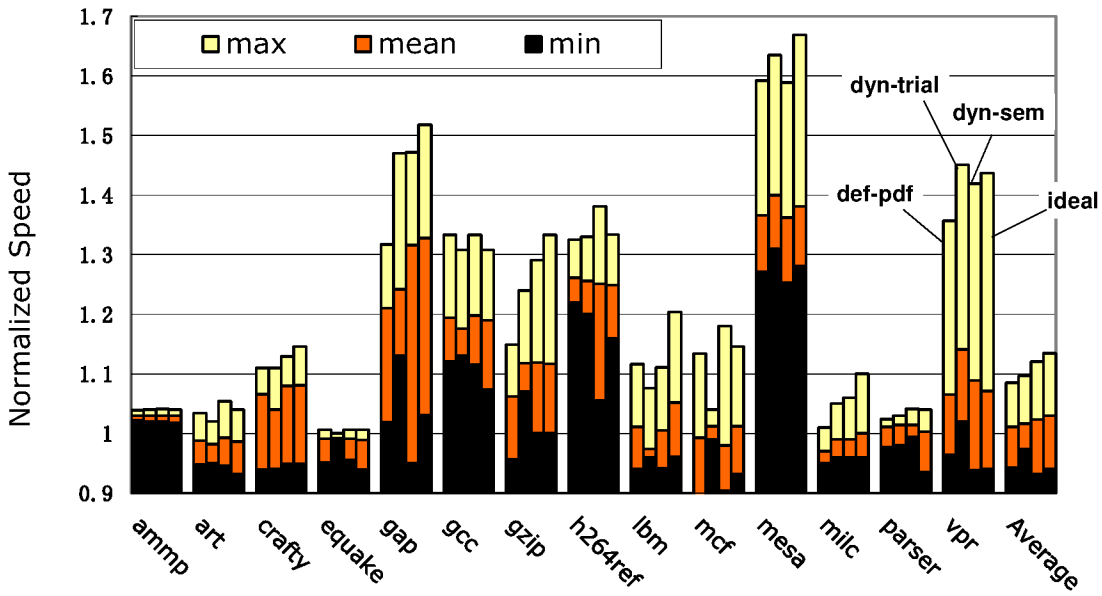

Figure 5. Speedup ranges from offline profile-feedback-driven compilation (*def-pdf*), dynamic version selection based on runtime trials (*dyn-trial*), seminal-behavior-based version selection (*dyn-sem*), and the compilation driven by ideal profiles (*ideal*). The baseline is the performance by IBM XL compiler at the highest optimization level.

Techniques in branch prediction (e.g., [45]) and value prediction (e.g., [26]) also exploit some correlations among program behaviors, but mainly on branches or data values and are typically implemented on hardware. The correlation-based input characterization adopted in this project requires more sophisticated larger-scope correlation analysis on a broader range of program behaviors than before. It also differs from dynamic invariant detectors, such as Daikon [13], whose goal is to find invariants rather than statistical relations.

6. Conclusion

Many studies have shown the important influence of program inputs on program behaviors, but this paper, for the first time, offers a systematic exploration on how to automatically exploit inputs for proactive dynamic optimizations. It describes a three-layer pyramid and presents a set of techniques for each layer to cultivate the paradigm of input-centric program behavior analysis and optimizations. The techniques together make dynamic optimizations proactive and holistic, eliminating some drawbacks of existing dynamic optimizations. The results on both Java and C applications demonstrate the effectiveness of the new paradigm in advancing the current state of program optimizations.

Acknowledgments

We are grateful to the Compiler Group at the IBM Toronto Software Lab, especially Yaoqing Gao and Shimin Cui, for their help on our questions about IBM XL compilers. We thank Jose Nelson Amaral's research group at the University of Alberta for sharing the extra inputs of SPEC benchmarks. We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499, 0811791, and 0954015, 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.

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. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 111–129, 2002.

[5] M. Arnold, A. Welc, and V. Rajan. Improving virtual machine performance using a cross-run profile repository. In *the Conference on Object-Oriented Systems, Languages, and Applications*, pages 297–311, 2005.

[6] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

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

[8] W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of PLDI*, pages 332–340, 2006.

[9] 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.

[10] 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.

[11] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 273–282, 1994.

[12] 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*, pages 71–84, Las Vegas, May 1997.

[13] M. Ernst, J. Perkins, P. Guo, S. McCamant, M. T. C. Pacheco, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

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

[15] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–304, Atlanta, Georgia, May 1999.

[16] N. Grcevski, A. Kilstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[17] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 24–34, 2008.

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

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

[20] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 220–229, October 2008.

[21] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Geathers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 248–256, 2010.

[22] T. P. Kistler and M. Franz. Continuous program optimization: a case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.

[23] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, May 2005.

[24] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of PLDI*, pages 239–251, 2006.

[25] X. Li, M. J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–124, 2004.

[26] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO-29)*, pages 226–237, 1996.

[27] 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)*, pages 92–101, 2009.

[28] F. Mao, E. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 91–100, 2009.

[29] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.

[30] M. A. Namjoshi and P. A. Kulkarni. Novel online profiling for virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 133–144, 2010.

[31] M. Paleczny, C. Vic, and C. Click. The Java Hotspot(TM) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[32] M. Polettto, W. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.

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

[34] X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2007.

[35] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.

[36] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of International Symposium on Computer Architecture*, pages 336–349, San Diego, CA, June 2003.

[37] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *Proceedings of the International Symposium on Memory Management*, pages 91–102, 2007.

[38] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 66–76, 2000.

[39] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the International Symposium on Memory Management*, pages 49–60, 2004.

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

[41] 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.

[42] 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.

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

[44] 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.

[45] T. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[46] C. Zhang and M. Hirzel. Online phase-adaptive data layout selection. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 309–334, 2008.

[47] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.