An input centric paradigm for program dynamic optimizations and lifetime evolvement

Kai Tian
College of William & Mary - Arts & Sciences

AN INPUT CENTRIC PARADIGM FOR PROGRAM DYNAMIC OPTIMIZATIONS AND LIFETIME EVOLVEMENT

Kai Tian

Feicheng, Shandong, China

Bachelor of Engineering, Shanghai Jiao Tong University, China

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
May 2012

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

[illegible]

Kai Tian

Approved by the Committee, April, 2012

[illegible]

Committee Chair
Professor, Xipeng Shen
Computer Science Department, The College of William and Mary

Weizhen Mao

Professor, Weizhen Mao
Computer Science Department, The College of William and Mary

[illegible]

Professor, Qun Li
Computer Science Department, The College of William and Mary

[illegible]

Professor, Denys Poshyvanyk
Computer Science Department, The College of William and Mary

[illegible]

Professor, John Cavazos
Department of Computer & Information Sciences, University of Delaware

ABSTRACT PAGE

Accurately predicting program behaviors (e.g., memory locality, method calling frequency) is fundamental for program optimizations and runtime adaptations. Despite decades of remarkable progress, prior studies have not systematically exploited the use of program inputs, a deciding factor of program behaviors, to help in program dynamic optimizations. Triggered by the strong and predictive correlations between program inputs and program behaviors that recent studies have uncovered, the dissertation work aims to bring program inputs into the focus of program behavior analysis and program dynamic optimization, cultivating a new paradigm named input-centric program behavior analysis and dynamic optimization.

The new optimization paradigm 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 extracting 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 is input-centric adaptation, which capitalizes on the novel opportunities created by the first two components to facilitate proactive adaptation for program optimizations.

This dissertation aims to develop this paradigm in two stages. In the first stage, we concentrate on exploring the implications of program inputs for program behaviors and dynamic optimization. We construct the basic input-centric optimization framework based on offline training to realize the basic functionalities of the three major components of the paradigm. For the second stage, we focus on making the paradigm practical by addressing multi-facet issues in handling input complexities, transparent training data collection, predictive model evolvement across production runs. The techniques proposed in this stage together cultivate a lifelong continuous optimization scheme with cross-input adaptivity.

Fundamentally the new optimization paradigm provides a brand new solution for program dynamic optimization. The techniques proposed in the dissertation together resolve the adaptivity-proactivity dilemma that has been limiting the effectiveness of existing optimization techniques. Its benefits are demonstrated through proactive dynamic optimizations in Jikes RVM and version selection using IBM XL C Compiler, yielding significant performance improvement on a set of Java and C/C++ programs. It may open new opportunities for a broad range of runtime optimizations and adaptations. The evaluation results on both Java and C/C++ applications demonstrate the new paradigm is promising in advancing the current state of program optimizations.

Table of Contents

To my parents and my wife, for their dedicated love and continued support . . .

ACKNOWLEDGMENTS

The dissertation would never be possible without the help and support from many people. I would like to take this opportunity to thank everyone who has helped me in big and small.

Particularly, I would like to express my deepest gratitude to my advisor, Dr. Xipeng Shen, for his excellent guidance, dedicated help, great edification, especially for his trust, encouragement, patience and understanding. He brought me into an exciting research area and taught me how to do research from scratch. He guided me through the entire process of doing exciting research, from exploring new research ideas, investigating different approaches, circumventing difficulties and challenges, to presenting research results and giving great talks.

I would also like to thank my parents for their continued support. They have always been supporting me and encouraging me along the way. Their love makes me feel encouraged whenever I encounter difficulties. My success in graduate study would never happen without their tons of help and support. I would like to give them my best wishes with my best achievements.

My dear wife Jia is a consistent source of love, care, support and understanding. She is always there helping me and accompanying me through all good and bad times. Her dedicated love and care always cheered me up when I was discouraged. The best moment in my life was when she married me. I could not tell how lucky I am to have her in my life.

Finally, I would like to thank all committee members and my friends for their help over the years. Thanks to all of them for making my graduate study at William and Mary a wonderful, exciting and rewarding experience.

List of Tables

List of Figures

AN INPUT CENTRIC PARADIGM FOR PROGRAM DYNAMIC OPTIMIZATIONS AND LIFETIME EVOLVEMENT

Chapter 1

Introduction

Program behavior analysis is essential for the maximization of computing efficiency, as program optimizations rely on predictions of program behaviors. Program behaviors in the dissertation 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 locality, data values, dependency, function calling frequencies, and so on.

Accurately predicting program behaviors is fundamental for various program optimizations and runtime adaptations. 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.

Despite decades of remarkable progress, prior studies have not systematically exploited program inputs, a deciding factor for program behaviors. Program inputs refer to all the data that are accessed but not generated by the program, such as command line arguments, interactively input data, files to read, and so on.

Recent studies [41] have reported strong influence program inputs impose on program runtime behaviors. They find 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). Formally, they 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. I'll use an example in Figure 1.1 to illustrate the intuition behind in the behavior correlations. Figure 1.1 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

```
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 1.1: A simplified mesh generation program.

correlate, if two behaviors stem from the different parts or features of an input) .

Triggered by the *strong* and *predictive* correlations between program inputs and behaviors, the dissertation aims to bring program inputs into the focus of program behavior analysis, cultivating a new paradigm named input-centric program behavior analysis and optimization. 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.

The dissertation aims to develop input centric paradigm in two stages. The first stage, Chapter 2, concentrates on exploring the implications of program inputs for program behavior analysis and prediction and the opportunities provided for program optimizations. We construct an offline training based framework to realize the basic functionalities of the three major components of the paradigm, including program input characterization, input-behavior modeling and input-centric adaptation. We identify seminal behaviors through offline training and build the models between seminal behaviors and the program behaviors of interest. Then during runtime, the program will be automatically optimized in a proactive, dynamic fashion based on the predictive models.

The second stage, Chapter 3, focuses on making the paradigm practical by addressing multifacet issues in handling input complexities, transparent training data collection, and so on. We propose a set of techniques that make it possible to automatically recognize the relations between program inputs and the appropriate ways to optimize it. The whole process happens transparently across production runs; no need for offline profiling or programmer intervention. We implement these techniques through modifying a Java Virtual Machine (JVM). On the modied JVM, a set of programs exhibit continuous efciency enhancement across their executions, with most runs significantly outperforming their performance on the default JVM.

Overall, the dissertation makes the following major contributions.

- The dissertation proposes and develops the first input-centric paradigm for program behavior analysis and optimizations. The dissertation systematically explores the special challenges existing in the characterization program input features, construction of the predictive models between input features and program behaviors.

- The dissertation demonstrates the use of input-centric optimization by integrating it into runtime systems for program optimizations. Meanwhile, it assesses the potential of input-centric optimizations by comparing them with both manual endeavors and the state-of-art optimization techniques.

- The dissertation is also the first exploration towards transparent input-centric dynamic optimizations, it reveals the main challenges and demonstrates the feasibility of the paradigm in practical running environments. The whole optimization paradigm happens transparently across production runs, hence cultivating a lifelong continuous optimization paradigm with cross-input adaptivity.

- It proposes several techniques to address the special challenges in transparent input-centric optimizations. Specifically, it develops randomized inspection-instrumentation to overcome difficulties for data collection for automatic input characterization over production runs. It develops a sparsity-tolerant algorithm to enable input characterization over data collected across production runs. It proposes a continuous learning framework that enables incremental evolvement of transparent input-centric dynamic optimizations with risks tightly controlled.

1.1 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents the basic input-centric optimization paradigm which is based on offline training. Chapter 3 addresses some limitations in Chapter 2 and focuses on making input-centric optimization paradigm happen transparently across production runs. Chapter 2 and Chapter 3 focus on program optimizations and performance improvement at program level. Chapter 4 wants to address the new challenges in performance improvement on job scheduling in modern Chip Multiprocessor (CMP) architectures. It gives a study on optimally co-scheduling jobs of different lengths on Chip Multiprocessors. Chapter 5 reviews some previous work related to the dissertation work and discusses the distinction between them. Chapter 6 concludes the dissertation with a short summary.

Chapter 2

The Basic Input-Centric Optimization Framework

2.1 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 [11,41,49,52, 53,77,82]. Such influence has been commonly regarded a hurdle for program optimizations: Static compilers have to optimize conservatively through transformations that fit all possible inputs [5,6]; 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 [9,11,52,53].

The work described in this chapter 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 2.4.1, for example, the clues indirectly derived from program inputs lead to accurate prediction of 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 12–29.6% performance improvement.

A few previous studies [49, 77, 83] 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 [52] 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 chapter, 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?

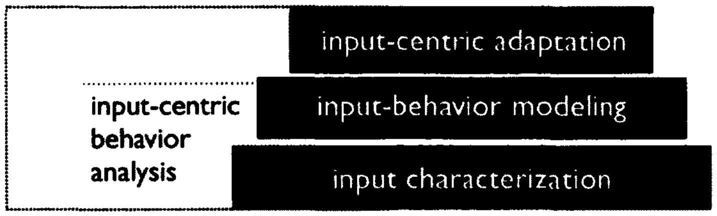


Figure 2.1: Proposed research components. They form a pyramid, with the top level exerting the power of the analysis conducted by the lower levels.

- 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 2.1. The bottom layer, *program input characterization* (Section 2.3.1), 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 2.3.2), 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 2.3.3), 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 the dissertation 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 2.4). 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 12% to 29.6%; in the IBM XL compiler, the speedups are up to 58% with averages between 5% and 13%.

The rest of this section proceeds as follows. Section 2.2 first gives a qualitative view on the importance of program inputs for program optimizations. Section 2.3 presents the input-centric pyramid with all of its three layers. Section 2.4 reports the evaluation results in Jikes RVM and IBM XL C. Section 2.5 concludes this section with a short summary.

2.2 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.

2.2.1 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 the dissertation 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:

$$Prog.\ Behaviors = Inputs + Code + Environments. \tag{2.1}$$

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 [88], sorting algorithm selection [49, 77], computation offloading [82], and memory management [53]. This connection is the intuition for program inputs to serve as clues for program behaviors prediction—the essence of input-centric program behavior analysis.

2.2.2 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.

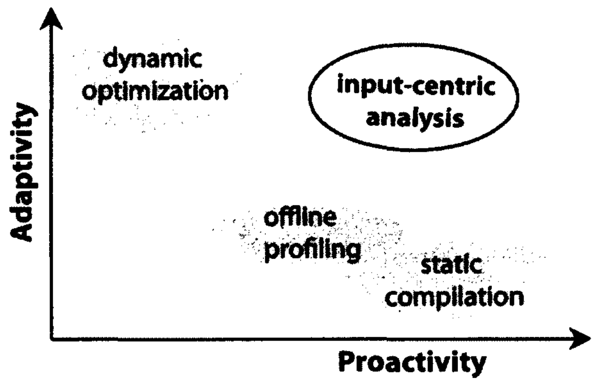


Figure 2.2: Insufficient treatments to program inputs causes a dilemma between the proactivity and adaptivity of program optimizations.

Existing approaches to program behavior analysis fall into three categories as illustrated in Figure 2.2. Static compilation [5, 6] 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.

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 [8, 18, 47, 57, 86] 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 [34]. *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 2.4.2).

Prior studies have revealed some evidences to those drawbacks of reactivity. In a study by Arnold and others [9], on a commercial Java Virtual Machine (IBM J9 [33]), programs may run 17-49% faster if the delay in optimizations is just partially removed. A later study [52] shows that enhanced proactivity increases performance even further. Other studies have seen similar benefits in memory management [53], locality phase prediction [67] and library development [49, 77].

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 [70] 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 [40]. 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 dissertation.

2.3 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 2.1. 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.

2.3.1 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, etc).

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 32KB 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 (e.g., loop trip-counts, function calling frequencies) of the program do not as long as the input files are of a similar size and some critical compression options remain the same [65]. Similarly, for a quick-sort program, the distribution rather than the values of input data determines the sorting process [49, 77]. As to be presented in Section 2.4, our examination of 10 Java and 14 C/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 number of various constructs in a program.

In the dissertation, we employ a technique, named seminal-behavior analysis, to circumvent the difficulties. 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. number 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 [41] 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, we 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. At the center of the framework is the concept of seminal behaviors.

2.3.1.1 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 earliness 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, a seminal behavior set, 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 a seminal behavior.*

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.

2.3.1.2 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 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.

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.

[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.

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 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 3.5 confirm this intuition.

The computation of predictive capabilities in our experiments is based on the standard 10-fold cross-validation [35]. 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 2.3 illustrates the result of this step on a 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 2.3, 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.

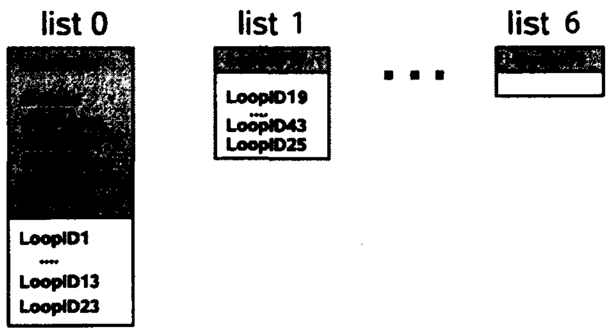


Figure 2.3: The affinity lists of program *mcf*.

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 3.5 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.

In the dissertation, 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) get exposed, we may immediately predict the trip-counts of loop-2 by putting C_1 into the linear equation $C_2 = 4 * C_1 + 9$.

The entire process for finding seminal behaviors is through a fully automatic tool; no manual efforts are necessary. 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.

2.3.2 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 execution, 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 treatment. Section 2.4 will show that treating these special challenges is critical for the quality of the produced input-behavior models.

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 [35]. 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, and so on.

To make the explanation concrete, we use the compilation of Java methods in Jikes RVM [7] 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 a set of target behaviors, each member of which corresponds to 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.

Classification and Regression Many classification and regression methods are applicable to input-behavior modeling. In the dissertation, we select Decision Trees as the primary approach for both classification and regression, because of its simplicity and other appealing 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 [35]. It divides the input space into local regions, and assigns a class label to each of those regions. Figure 2.4 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 [35].

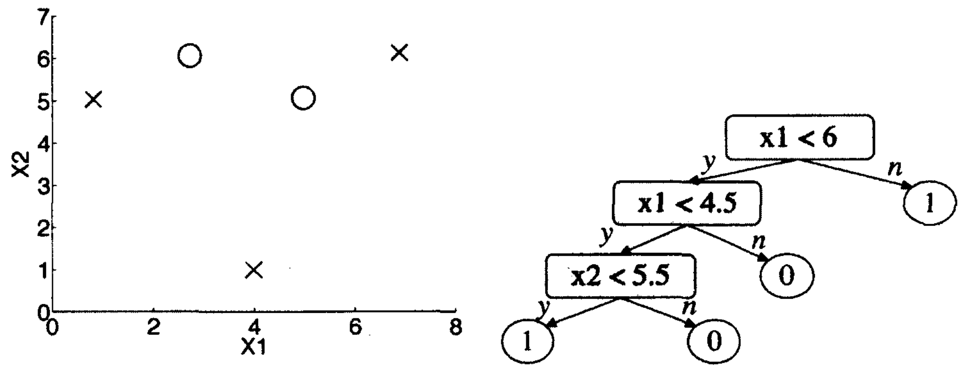


Figure 2.4: A training dataset (o: class 0; x: class 1) and the classification tree. Each leaf node is labeled with a class number.

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 five of its appealing properties. *First*, the construction and use of Decision Trees are simple, efficient, and fully automatic. *Second*, Decision Trees has very good interpretability. Unlike Neural Network 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 handles 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 the important 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. This feature reduction 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 (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.

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 [35]. 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 this problem through feature selection techniques. As mentioned earlier in this section, Decision Trees can automatically filter out non-important features. Moreover, during the LMS regression, we apply a standard stepwise method to recognize the 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. If adding the feature into the model improves the predictive capability of the model substantially (evaluated through F-statistic analysis [35]), the feature is added into the model. This process continues until no more features can be added. By reducing the number of features, the required number of training runs becomes smaller. Meanwhile, the risk control (presented next) distinguishes the subspaces in the input space that are predictable from those that are not. This discrimination further reduces the tension between the number of training runs and the number of features as the learning space is pruned.

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 [52] 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 the dissertation, 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 uses multiple confidence values to handle such differences. Moreover, it maintains the range of the input sub-spaces (more precisely, seminal behavior spaces) which have been covered by the training runs. If the new input falls outside of the range, 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 across production runs of the program. The newly observed behaviors and input features 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 periodically (when the machine is idle) using the expanded data set, with the confidence levels updated accordingly.

2.3.3 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.

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 2.2.2.

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 2.2.2, 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.

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 [7]. 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 incur 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, we modify the Jikes RVM so that as soon as the values of seminal behaviors become explicit in an execution, it plugs 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 [9, 52].)

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 2.4.1 reports the resulting performance improvement.

Use 2: Proactive Dynamic Version Selection. The second use is on C/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 [20]. 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 get exposed in that run. In this way, we do not need the trials of the different versions during runtime, hence circumventing both limitations the previous work faces. Details of the implementation will be presented in Section 2.4.2 along with the performance results.

Discussions. Proactive optimizations do not conflict with the presence of program phase shifts. Many studies [67, 68] 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, while 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.

2.4 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 the default dynamic optimizers in an open-source Java Virtual Machine (Jikes RVM) and a product compiler (IBM XL compiler), along with some previous results from manual endeavors, demonstrate the potential of the input-centric paradigm for program optimizations.

2.4.1 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 2.1 reports the used benchmarks, which come from three benchmark suites. They have been used in a previous study [52], in which, manually characterized input features are employed for proactive optimizations[2]. So using

[2]We did not include the program *fop* because the current implementation of the seminal behavior analysis encounters some unknown problems when running on that program.

this set of benchmarks makes it convenient to compare with the previous results. In the previous work [52], the authors collected a number of extra inputs for each of the programs for their experiments. Those inputs are used in this dissertation as well.

Behavior Prediction Accuracy. The fourth to sixth columns in Table 2.1 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 [53, 69]. 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 2.3.3.

Table 2.1: The Number of Seminal Behaviors and Prediction Accuracies

Program	#	Through seminal behaviors				Manual approach [52]	
	of	# of	Pred accuracy			# of	Pred accuracy
	inputs	sem.	call	min	opt.	features	opt.
		beh.	freq	heap	level		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 [3]; d: dacapo [13]; g: grande [2]

The results show that the seminal behaviors can predict most of those behaviors with over 94% accuracy. The rightmost two columns in Table 2.1 list the results from a previous work [52]. 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 important features (e.g., for most of the Grande benchmarks), the enhanced statistical learning algorithms, as described in Section 2.3.2, improves the prediction accuracy substantially by systematically handling categorical features and reducing the dimensionality of the input space through PCA and step-wise 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 input-centric optimization guided by the behavior model based prediction, as described in Section 2.3.3. 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 [52] using the previously proposed proactive optimizations based on manually characterized input features. To further examine the effectiveness of the proposed techniques, we also compare the performance of input-centric optimization with the performance (denoted as "Replay") of replay compile mode in Jikes RVM [10]. In replay compile mode, a Java program is executed two times for each input. The first run is to collect advice files based on profiling information. In the second run, the replay compile mode directly applies optimization to the Java program based on the advice files with the same input. Ideally replay compile mode can acquire a complete knowledge of the current program execution and utilize it to optimize program at the beginning of each run. A comparison with the replay compile mode can show the efficiency of the input-centric optimization which is based on program behavior prediction rather than an accurate knowledge.

Figure 2.5 shows the results. The baseline is the performance of the default executions in Jikes RVM. As the speedup differs on different inputs, each bar shows the minimum, average and maximum speedup of all the runs of a program. On average, input-centric optimization yields 12–29.6% 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 over manual endeavors. The small distance from the ideal replay compile mode demonstrates the effectiveness of the input-centric optimization framework in program behavior prediction and proactive optimization.

2.4.2 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

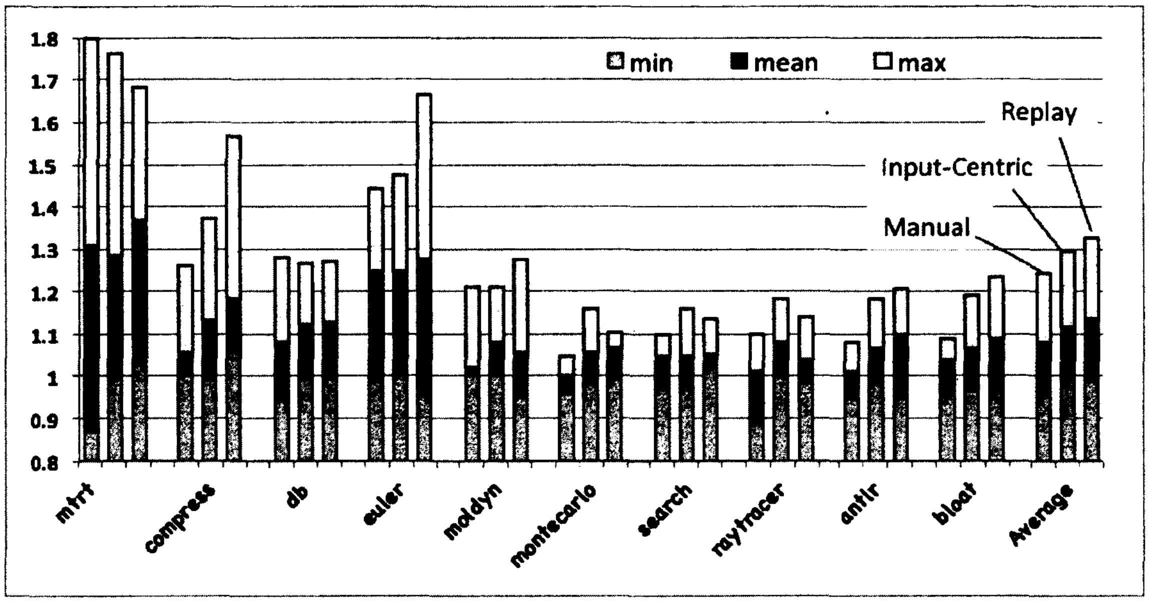


Figure 2.5: The speedup ranges produced by input-centric proactive optimization, compared to the default optimization scheme in Jikes RVM (the baseline), the proactive optimizations based on manually characterized input features (the "Manual" bars), and the ideal replay compile mode (the "Replay" bars) in Jikes RVM.

correspond to the node, edge, and data profiles, reflecting the dynamic behaviors of the program on basic block frequency, function call edges, and data 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 [20]. 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, as soon as its seminal behavior values become explicit, they 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 2.2 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 only works 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 critical 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 the predictability of seminal behaviors. We collect more inputs as shown in the third column of Table 2.2. 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. The fourth column of Table 2.2 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 2.6 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.

Because we have a number of inputs for a program and the speedups on them are different, each bar in Figure 2.6 shows the speedup range. The results reveal the following phenomena.

Table 2.2: 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	6.0×10^6	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

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 the 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 a considerable distance between 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 usefulness of seminal behaviors and the promise of proactive 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.

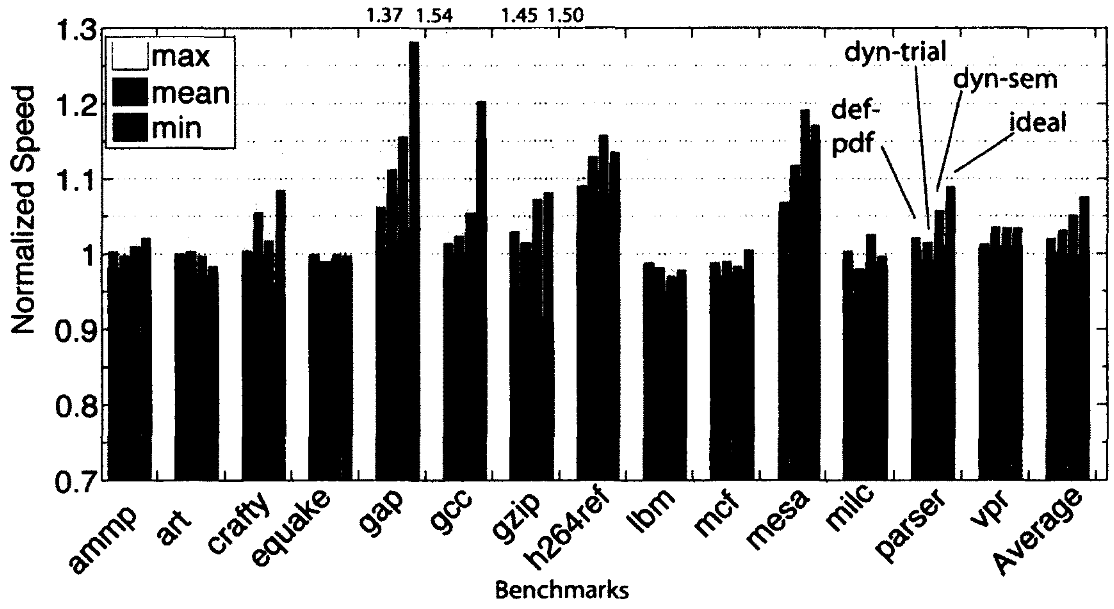


Figure 2.6: Speedup ranges from offline profile-feedback-driven compilation (*def-pdf*), dynamic version selection based on runtime trials (*dyn-trial*), seminal-behavior-based version selection (*dyn-sem*), and the compilation driven by ideal profiles (*ideal*). The baseline is the performance by IBM XL compiler at the highest optimization level.

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.

2.5 Summary

Many studies have shown the important influence of program inputs on program behaviors, but this chapter, for the first time, offers a systematic exploration on how to automatically exploit inputs for program 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/C++ applications demonstrate the effectiveness of the new paradigm in advancing the current state of program optimizations.

Chapter 3

Transparent Input-Centric Optimization Paradigm

The previous section demonstrates the importance of program inputs for program dynamic optimization. The developed framework, although sufficing for preliminary explorations, relies on offline training heavily. That reliance significantly limits the applicability of the technique as in many cases, the offline profiling is either not feasible (e.g. under a time pressure for software release) or not preferable for extra efforts required.

The goal of the second stage of this dissertation is to make input-centric analysis and optimizations happen transparently across production runs, hence developing a lifelong continuous optimization paradigm with cross-input adaptivity. Figure 3.1 is a conceptual graph for outlining the optimization paradigm that works over a runtime program optimizer (e.g., JIT in a JVM). In this paradigm, a production run of a program belongs to one of three stages, corresponding to the three squares. The first stage is input characterization, which produces a vector that captures the key features of the program's inputs. The second stage constructs a predictive model $f()$, which maps the values of the features of an input to the appropriate optimization decisions. We name the model *input-opt model.* All future runs will be optimized with the assistance of the input-opt model: As soon as the values of the features of an input become known, the runtime optimizer may optimize the program by resorting to the prediction of the input-opt model. The optimization exploits the knowledge extracted from history runs, and meanwhile, is customized to the current input. After each optimized run, some feedback information (collected with negligible overhead) is sent back to an idle-time analyzer (running at machine idle time or on a separate idle machine) to refine the input characterization and model construction to further enhance the optimizations to future executions.

3.1 Introduction

Dynamic program optimizations play a central role for enhancing the performance of applications in managed programming languages (e.g., Java and C#) [8], as well as scripting languages (e.g., Javascript) [30]. Even though remarkable progresses have been achieved, most existing dynamic optimizations have not systematically exploited an important factor: program inputs.

Program inputs refer to all the data that are accessed but not generated by the program, including command-line options, content of input files, and so on. Many studies have reported the importance of program inputs in determining program behaviors and hence appropriate optimization decisions [41, 49]. The strong correlation suggests the potential of using program inputs as hints to predict large-scoped behaviors of a program and hence assist dynamic optimizations. Chapter 2 show that the hints from program inputs may help make dynamic optimizations more proactive (e.g., optimizing a method appropriately before any of its invocations) and long-sighted (i.e., optimizing for the efficiency of the entire execution rather than a small interval), leading to significant speedups. For instance, Chapter 2 have shown 10–29% speedup for a set of Java programs and 5–13% for a number of C programs when inputs are considered in program dynamic optimizations, Li and others have observed 44% performance potential for sorting library construction when certain attributes of input data sets are used [49].

However, program inputs are often complex—a plausible reason for the lack of exploitation of inputs in existing dynamic optimizations. An input file, for instance, may have various syntactic structures and semantics (e.g., a tree, a graph, a video, a document, or a program). To exploit inputs for optimizations, it is necessary to obtain a clean structure that captures the important input features and is amenable to automatic processing. But the complexity of inputs makes this task extremely challenging. Chapter 2 and some prior studies have proposed the use of annotations [52] or offline profiling-based solutions [41]. But both put extra burdens on programmers. In addition, the annotation approach requires extensive knowledge of the programmers on both the application and its interactions with the underlying execution stack, while the offline solutions demand a large number of representative inputs and many offline profiling runs with detailed code instrumentation. These limitations impair the adoption of these solutions in practice.

This chapter is one step towards addressing this open question. It examines the basic feasibility of transparent integration of input-consciousness into dynamic program optimizations, particularly in managed execution environments (e.g., Java Virtual Machines) equipped with Just-In-Time compilers (JIT). It uses transparent, continuous learning across production runs as the basic vehicle, proposes several techniques to address some key challenges for the transparent integration, and investigates the implications of cross-run learning on each main component of transparent input-centric dynamic optimizations.

3.1.1 Overview of Transparent Input-Centric Optimization

As Chapter 2 describes, transparent input-centric dynamic optimization mainly consists of three components: input characterization, input-behavior modeling, and input-based adaptive optimizations. *Input characterization* identifies features of program inputs that are important for optimizations. *Input-opt modeling* constructs predictive models, which maps the values of the features of an input to the appropriate optimization decisions for the corresponding execution. We name the models *input-opt models*. These two components provide the foundation for the final component, *input-based adaptive optimizations*, which feeds the values of input features of the current run into the constructed input-opt models to guide the dynamic optimizations for the present run.

In Chapter 2, most parts of transparent input centric optimizations, except the third component, happen in an offline training process, requiring the application developers to conduct a large number of offline-profiling runs with detailed instrumentation and data collection.

The solution investigated in this chapter tries to integrate all components into a continuous learning process that happens across *production runs*. It is fully automatic, imposing no special requirement on either application developers or users.

As Figure 3.1 illustrates, in this paradigm, all three components of transparent input-centric dynamic optimizations mingle into a continuously evolving process. A production run may benefit from the current results of input characterization and input-opt modeling when dynamic optimizers use them to help optimize the current execution. On the other hand, after each production run, new observations are added into the database, which is used periodically (during the idle time of a machine) to refine the input characterization and input-opt models to better serve future runs.

This cross-run learning scheme circumvents the needs for offline profiling, and hence overcomes

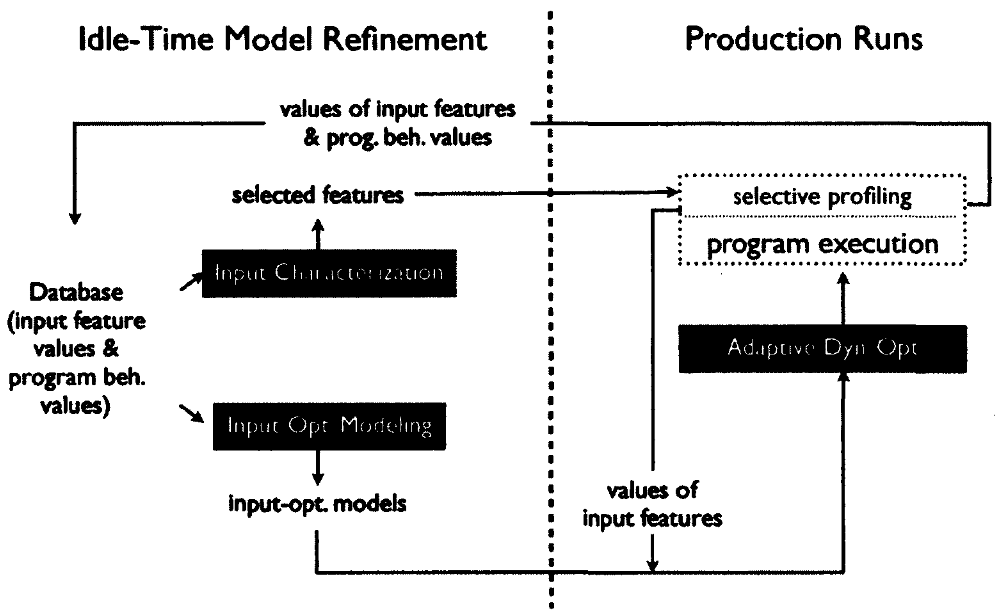


Figure 3.1: A high-level view of transparent input-centric optimization paradigm.

the limitations of prior solutions. But for the scheme to work effectively, several new challenges must be addressed for each of the three components.

The challenges to input characterization are the most difficult of all. Due to the complexity of inputs, the only existing solution for automatic input characterization is the *seminal-behavior identification* approach, proposed by Jiang and others [41]. It however requires the collection of many runtime behaviors of a large number of runs through detailed instrumentation, which is unaffordable for production runs. Runtime sampling is a natural direction. However, how to achieve large coverage quickly without disturbing performance of production runs is a challenge, especially for seminal-behavior identification because *1)* it is based on statistical correlation analysis, the data required by which is usually tremendous; *2)* dynamic instrumentation is necessary, which complicates overhead control as both the inserted instructions and the instrumentation process itself cause overhead.

In this chapter, I propose randomized inspection-instrumentation to solve the problem. The solution is based on a cross-user sampling scheme [50]—that is, accumulating data sampled from many users' executions—with two extensions. Its inspection-instrumentation mechanism helps control instrumentation-incurred overhead. Its randomization feature accelerates the coverage of the data collection by randomizing the coverage of the samples among users. Section 3.2.2 describes these two techniques.

The data collected through the lightweight profiling tend to be sparse, causing difficulty for the prior approach, seminal-behavior identification, to effectively characterize program inputs. We solve the problem by developing a sparsity-tolerant algorithm, which capitalizes on the partial overlaps of the data from different runs to incrementally propagate the extracted knowledge on program inputs. Section 3.2.3 presents the algorithm.

The challenges to the other components of transparent input-centric dynamic optimizations mainly come from the continuous evolvement of input features and input-opt models, a phenomenon that exists in no prior offline-based scheme. The issue is how to exert the full power of the incrementally enhanced predictive models without risking much the negative effects of prediction errors.

Sections 3.3 and 3.3.3 describe our solution to this issue. It uses self-assessment and selective prediction to control the risks of wrong predictions. By maintaining a confidence value, it prevents immature uses of input-opt models without being too conservative, coordinates the different components of transparent input-centric optimizations, and ensembles them into a concerted continuous optimization system to proceed transparently and profitably.

Experiments based on a JVM, namely Jikes RVM [8], demonstrate that the proposed techniques are effective in addressing some major obstacles for transparent input-centric dynamic optimizations. On the Java benchmarks evaluated, the proposed techniques outperform their executions on the default Jikes RVM and the previous input-oblivious approach substantially (Section 3.5).

We stress that creating a complete transparent input-centric dynamic optimizer that is ready to deploy in practice is not the goal of this current work. To reach that goal, there are many other obstacles (e.g., differences in platforms and software versions as detailed in Section 3.4) to conquer, which require many efforts from the community that are probably far beyond what can fit into a single paper. The contributions of this dissertation are at the proposal of solutions to some of the key obstacles, and the demonstration of the feasibility of the desired scheme in some basic setting.

In summary, this chapter makes four main contributions.

- As the first exploration towards transparent input-centric dynamic optimizations, it reveals the main challenges and demonstrates the feasibility of the paradigm in some basic settings.
- It proposes randomized inspection-instrumentation to overcome difficulties for data collection for automatic input characterization over production runs.
- It develops a sparsity-tolerant algorithm to enable input characterization over data collected across production runs.
- It proposes a continuous learning framework that enables incremental evolvement of input-centric dynamic optimizations with risks tightly controlled.

The rest of this chapter is organized around the challenges each of the three key components of transparent input-centric dynamic optimizations has to meet so that the entire integration of input-consciousness can occur transparently over production runs. But first, we give a brief description of the underlying platform we use as it is closely relevant to the remaining discussions.

3.1.2 Platform: Jikes RVM

We use Jikes RVM [8], an open-source JVM originally from IBM, as our main platform for its representativeness as a dynamic optimization system. We briefly describe some of its features that are closely relevant to the following sections.

Jikes RVM uses method-level JIT compilation. Like most existing dynamic optimization systems, the optimizer in Jikes RVM is reactive: During an execution, it 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 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 the others.

$$\boxed{\begin{array}{c} \text{opt-level} = \underset{i \in \{0,1,2\}}{\text{argmax}} \, (\text{Benefit(i)} - \text{Cost(i)}). \\ \text{Benefit(i)} = T_{\text{future}} * (1 - 1/\text{speedup(i)}). \\ \text{Cost(i)} = \text{compileSpeed(i)} * \text{method\_size}. \end{array}}$$

Figure 3.2: The default cost-benefit model (with simplification for illustration purpose) in Jikes RVM for determining the optimization level for a Java method. (T_{future}: the estimated time the method is expected to take (if not optimized) in the rest of the current execution. *speedup*(i): the expected speedup of the method after being optimized at level i. *compileSpeed*(i): the compilation speed at level i.)

During an execution, the default Jikes RVM uses a cost-benefit model to determine whether a method should be recompiled at a higher optimization level. As shown in Figure 3.2, in the cost-benefit model, the cost is the time needed to compile the Java method, estimated from the size of the method and some predetermined compilation speeds at various compilation levels. This cost calculation is directly used in the inspection-instrumentation scheme described in the next section. The benefit is estimated as the expected time savings in the rest of the execution because of this recompilation. In Jikes RVM, there are some predetermined constants that represent the average speedup each optimization level produces [8].

The parameter T_{future} in the benefit formula means the time that the method is expected to take in the rest of the current execution. Jikes RVM assumes that T_{future} equals the time this method has already taken. As a consequence, in one run, a method may be recompiled several times at increasing optimizing levels, as Jikes RVM realizes the importance of the method gradually.

3.2 Transparent Input Characterization

Among the many challenges for transparently integrating input-consciousness into dynamic optimizations, the most difficult ones reside in the first component, input characterization.

The main goal of input characterization is to reduce the raw program inputs to a set of features. These features critically determine the behaviors of the program that are essential to its performance.

The difficulty comes from the complexities in program inputs. 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).

Our solution is based on a recently proposed concept, *program seminal behaviors*. We first briefly review the concept and then describe our techniques for transparently characterizing program inputs over production runs.

3.2.1 Review on Program Seminal Behaviors

The concept of program seminal behaviors is proposed by Jiang and others [41]. It comes from the strong statistical correlations among program behaviors. For example, two loops in a program have iterations (15, 41, 52, 89, 101), and (69, 173, 217, 365, 413) in five runs respectively. Statistical analysis can easily determine that the trip-counts (i.e., the numbers of iterations) of these two loops have a linear relation as $C_2 = 4 * C_1 + 9$, where, C_1 and C_2 are the trip-counts of the two loops. Jiang and others have shown 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 an automatic approach to recognizing a small set of behaviors in a program, named a *seminal behavior set*. These behaviors have two properties. First, they have strong statistical 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 a SPEC CPU2006 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 of these seminal behaviors become known during the first 10% portion of an execution.

Recall that the essence of transparent input-centric dynamic optimizations is to use program inputs as the hints to predict the behaviors of the program and guide the optimizations. The properties of seminal behaviors suggest that they can play the same role that program inputs play in the optimization paradigm. *A seminal behavior set can be hence viewed as one form of characterization of* *program inputs.*

The previously proposed approach to recognizing seminal behaviors includes a profiling step and an analysis step. The profiling step collects the values of some candidate behaviors (e.g. all loop trip-counts) by running the program—which has been instrumented in detail—on many different inputs. The analysis step greedily classifies the behaviors into some affinity lists. Let S be the set of all candidate behaviors. In each iteration of the analysis step, one behavior b is taken out of S. Using the data collected in the profiling step, the algorithm finds all behaviors in S that strongly correlate with b, extracts them out of S, and puts them into a newly created list (called an affinity list); b is called the head of the affinity list. This process repeats until S is empty. The union of the heads of all affinity lists is taken as the seminal behavior set.

Although the approach has shown effective for program behavior prediction and optimizations [41], its design fails to meet the needs of transparent input characterization. There are two major hurdles. First, as a statistical approach, the method requires tremendous data collected through detailed instrumentation, which is apparently unaffordable for production runs. Second, the algorithm for seminal behavior identification works for large, dense profiling results. But data collected through production runs tend to be sparse. How to recognize seminal behaviors over sparse data remains unclear. The following two sub-sections present our solutions to each of them.

3.2.2 Randomized Inspection-Instrumentation for Data Collection

There have been much work on lightweight runtime profiling [14,18,37,42]. In this chapter, we use cross-user sampling [50] as the underlying vehicle for its strength in quickly accumulating a large set of samples. The basic idea of cross-user sampling is simple. In each run of a given program, the JIT instruments a small portion of the program and record the profiling results. The sampled information from different runs and by different users (e.g., all the customers of a software) of the program are accumulated together to form one data set.

Initially, applying cross-user sampling to input characterization appears to be a straightforward process. But when we take into consideration the distinctive properties of input characterization, it turned out to be a challenging task.

The special difficulties exist in two aspects. *First*, as the data to collect are for thorough statistical correlation analysis, the required instrumentations are intensive. For instance, one type of data needed is the trip-count of every loop, attaining which often requires the insertion of counter update instructions that needs to be executed in every iteration of the loop. *Second*, the instrumentation must happen on the fly. Most previous designs of cross-user sampling [50] are for static instrumentation, which inserts some monitoring instructions into the software before the deployment of the software. But our problem requires instrumentation to adapt to each run because of our focus on the influence of program inputs. As a consequence, the overhead of the sampling comes not only from the execution of the instrumented instructions, but also from the instrumentation process itself, making it more difficult than before to control the total overhead to a given limit—a requirement critical for production runs.

This sub-section describes two techniques,
inspection-instrumentation and randomization, designed to address these difficulties.

Behaviors to Profile Before looking at the solutions, we first explain the behaviors needed to profile. Following prior observations [41], we focus on two kinds of program behaviors, from which, seminal behaviors will be recognized later through statistical correlation analysis.

The first kind of behaviors is *interface behaviors.* (Please note that the name has nothing to do with Java Interfaces.) They consist of the values obtained directly from program inputs—such as the values of command-line options and values from file reading operations. We ignore a file operation that falls in a loop, the trip-count of which is either large or unknown during compile time. Such an operation tends to be accessing some massive data set, the values of which may not influence the coarse-grained behaviors much, but may significantly inflate the candidate behavior set and complicate the recognition of seminal behaviors. Interface behaviors have two appealing properties: They usually correlate with program inputs strongly; they tend to reside in the initialization part of a program, and hence their values often become known in the early stage of an execution—an important property for the uses of the predictive models built later on (Section 3.3.2).

The second kind of behaviors are the *trip-counts of all the loops* in the program thanks to the importance of loops and their strong correlations with other program behaviors [41].

3.2.2.1 Overhead Control through Inspection-Instrumentation

Collection of interface behaviors (that are not loop trip-counts) requires only the recording of some variable values, incurring negligible overhead. The focus of overhead control is on the collection of loop trip-counts.

Before describing the overhead control strategy, we first explain the source of overhead. There are two sources of overhead for collecting loop trip-counts.

- The first is compilation overhead. It relates to the way JIT works. We use Jikes RVM for explanation; many other managed environments have similar schemes. By default, the JIT in Jikes RVM compiles a Java method using a basic compiler when encountering the method for the first time. The compilation is essentially a simple byte code translation with little data or control flow analysis. Later recompilations are through an optimizing compiler. Only compilation by the optimizing compiler (at levels 0, 1, or 2) exposes loop structures. Our loop instrumentation is implemented in the optimizing compiler. So in order to collect loop information, the selected Java method is compiled by the optimizing compiler (at level 0) rather than by the basic compiler when it is loaded for the first time. Because compilations happen during runtime, the extra time incurred by the optimizing compilation over that by the default basic compilation is the first source of overhead.

- The second source is execution overhead. To get the trip-count of a loop, the compiler inserts a counter-increase instruction into the loop body. Executions of these instructions happen in every loop iteration, forming the second source of overhead.

Without a careful control, the two kinds of overhead may cause unacceptable slowdown to the program executions.

Our solution is a guarded adaptive scheme for instrumentation, named inspection-instrumentation. The basic idea is simple: If we can estimate the overhead of an instrumentation, we would be able to control the amount of instrumentations so that the total overhead is within an acceptable limit. But because the inspection and selective instrumentation both have to happen over production runs, they must be carefully designed to work hand-in-hand over the often incomplete view exposed by production runs on program behaviors.

The designed inspection and instrumentation mingle together through all production runs. But for clarity of explanation, we describe them separately as follows.

Inspection The purpose of inspection is to estimate the compilation and execution overhead that an instrumentation may incur.

1) Compilation Overhead. Compilation overhead mainly relates with the size of a Java method. Typically, a JIT is able to estimate the time needed to compile a method at each optimizing level. For instance, there is a table in the default Jikes RVM that lists the compilation speeds of the JIT at various compilation levels (Section 3.1.2). So with the size of a Java method revealed, the compilation overhead can be easily estimated from the compilation speeds.

The size of a Java method is obtained incrementally across runs. In each run, as the JIT compiles a Java method, it gets the size of the method for free. It records the size into a database if it is not there yet. The first-time runs by all the users (likely on many different inputs) typically give a good coverage of all the Java methods in the program. If a later run encounters some new methods, these methods are excluded from instrumentation in that run. Their size will be added to the database for guiding the instrumentations in future runs. The recording of a method size happens once per Java method, incurring negligible overhead.

2) Execution Overhead. The estimation of execution overhead is based on the following assumption:

> *After a loop is instrumented, it becomes* $1/S$ *or less slower than its default run, where* S *is the size of the loop body in terms of the number of instructions.*

This assumption comes from the fact that the instrumentation inserts only one counter increase instruction into the loop body. We acknowledge that the assumption may not hold in certain cases (e.g., with early returns). But most other parts of the overhead estimation algorithm are conservative. Overall, the assumption causes no noticeable effects as experiments show (Section 3.5).

Estimation of execution overhead takes place during the compilation of a Java method by the optimizing compiler. The compiler lists the loops in an ascendingly ordered sequence based on their body size. For a nested loop, the size of the outer loop does not include the inner loops. Let L_i be the ith loop in that ordered sequence ($i = 1, 2, \cdots, M$), with M for the total number of loops in the sequence. The overhead estimation mainly uses the following proposition:

Proposition 3.1 *For any given i, when all loops, L_j ($i <= j <= M$), are instrumented, their incurred execution overhead (normalized by the execution time of the program's default run) is no more than $1/size(L_i)$. ($size(L_i)$ is the number of instructions in L_i.)*

To see the correctness, one needs to notice that because of our assumption described two paragraphs earlier, after the instrumentation, the total execution time of the program becomes $T' = T_{rest} + \sum_{j=i,\cdots,M} T_{L_j} * (1 + 1/size(L_j))$, where, T_{rest} is the time the non-loop parts of the program take in the default run of the program, and T_{L_j} is the time loop L_j takes in the default run of the program with the time spent in its inner loops excluded. Because $size(L_j) >= size(L_i)$ ($i <= j <= M$), we have

$$\begin{aligned} T' &<= T_{rest} + (1 + 1/size(L_i)) \textstyle\sum_{j=i,\cdots,M} T_{L_j} \\ &<= (1 + 1/size(L_i))(T_{rest} + \textstyle\sum_{j=i,\cdots,M} T_{L_j}) \\ &= (1 + 1/size(L_i))T_{def}, \end{aligned}$$

where T_{def} is the execution time of the program in its default run. The correctness of the proposition follows.

Instrumentation Aided by the inspection component, the algorithm of instrumentation ensures that the ratio between the estimated total overhead and the default running time does not exceed a predefined threshold, H (a small number between 0 and 1; 2% in our experiments).

Figure 3.3 outlines the algorithm. For simplicity of explanation, first assume that the default execution time of the current run, T_{def}, is known beforehand.

```
static int totalCost=0;

Procedure methodProcess (Method$_j$){
  if ($C_j$ == null){
   recordSize (Method$_j$);
   defaultCompile (Method$_j$);
  }
  else
   if (totalCost + $C_j$ > H* T)
    defaultCompile (Method$_j$);
   else{
    totalCost += $C_j$;
    compileWithInstrument (Method$_j$);
   }
}

Procedure compileWithInstrument (Method$_j$){
  optCompile (Method$_j$);
  LoopList = sortLoops (Method$_j$); // from small to large
  instruB = false;
  foreach e in LoopList{
   if (instruB) instrument (e);
   else if (totalCost/T + 1/e.size < H){
    instruB = true;
    instrument (e);
   }
  }
}
```

Figure 3.3: The online algorithm for guarded adaptive instrumentation.

The runtime system (JVM) uses a variable *totalCost* to track the total estimated overhead (normalized by the default execution time) that may be incurred by instrumentations that have been done in the current run. Its value is zero at the beginning of an execution. The instrumentation algorithm consists of three steps:

Step 1) When a method, M_j, is loaded, the JIT checks whether its size has been recorded (by the inspection in previous runs). If not, the method will be excluded from the instrumentation, and compiled in the default way. Otherwise, this method may need to be instrumented; the algorithm proceeds to the second step.

Step 2) Recall that instrumentation can only happen through optimizing compilation. In this step, the JIT computes the overhead (C_j) that may be incurred by compiling M_j with the optimizing compiler. By comparing ($totalCost + C_j/T_{def}$) against H, it determines whether using the optimizing compiler to compile this method is affordable. If not, the method is compiled in the default way with no instrumentation. Otherwise, tries to instrument M_j by following the next step.

Step 3) The JIT increases $totalCost$ by C_j/T_{def}, and does the optimizing compilation, during which, it tries to instrument the loops in the method selectively as follows. It examines the loops in the method in an ascending order of their body size. Its examination stops when it encounters a loop, denoted as L, that meets the condition

$$(1/size(L) + totalCost) < H. \tag{3.1}$$

From Proposition 3.1, we know that the instrumentation of loop L and all loops that are larger than it incurs no more overhead (normalized by T_{def}) than $1/size(L)$. Therefore, meeting condition 3.1 means that all these loops can be instrumented without incurring too much overhead. The JIT increases $totalCost$ by $1/size(L)$, and then instruments all these loops. The program execution continues.

In the above description, we assume that T_{def} is known. It is rarely true in reality. To circumvent the problem, we instead use the approximated shortest execution time, T_{short}, of the program.

The approximation of T_{short} is over the first-time runs of all users. The JVM records the execution times of those runs (likely on various inputs). The idle-time analyzer computes the mean (m) and standard deviation (d) of those times. The value $m - 3d$ is taken for T_{short}. Using $m - 3d$ rather than the minimum of all run times is to avoid the noise from abnormal executions of the program;

it is a standard way in statistics for outliers filtering (including the use of "3") [35].

As typically $T_{short} <= T_{def}$, that replacement only inflates the estimated overhead, hence adding no risks but extra conservativeness to the selective instrumentation.

Coverage Maximization through Randomization A factor critically determining the coverage of the lightweight profiling is the time when the instrumentation algorithm starts to run in an execution. For instance, if it always starts at the beginning of an execution, due to the limited affordability, only the methods invoked early in the executions would get a chance to be instrumented.

To help achieve a large coverage quickly, we design a randomized scheme. For each copy of a Java application (likely owned by different users), the JVM maintains a variable, $insStart$, which determines the time when the instrumentation algorithm starts to run.

After the first execution of the application, the JVM assigns a random value to $insStart$. The value is an integer between zero and N (the number of methods in the program). In an execution of the program (except the first-time run), the instrumentation algorithm starts after the number of methods that have been loaded equals $insStart$. After each run, the value of $insStart$ is updated to $(insStart + m)\%N$, where m is the number of methods getting instrumented in the just-finished run. If no methods were instrumented (e.g., when $insStart$ is greater than the number of methods loaded), m is set to be 1 to encourage the continuation of the instrumentation in the next run.

The randomization of the initial value of $insStart$ helps to diversify the instrumentation coverage of the executions by different users. Meanwhile, the regular updates to $insStart$ within the executions by the same user ensure a systematic coverage of the entire program among the executions by that user.

3.2.3 Correlation Propagation for Seminal Behavior Recognition over Sparse Data

Based on the data accumulated through cross-user sampling, the idle-time analyzer tries to identify seminal behaviors. Recall that the goal is to find a small set of behaviors that strongly correlate with other behaviors by processing the collected data set.

A special complexity imposed by the transparent data collection is that the collected data set tends to be sparse because of the low tolerance of overhead by production runs. The sparsity complicates the correlation analysis. For example, as Figure 3.4 illustrates, $Loop_1$ and $Loop_2$ are never sampled in one common run. So even if the trip-counts of the two loops actually correlate with each other strongly, a direct correlation calculation on their sampled trip-counts cannot uncover that.

	run_1	run_2	run_3	run_4	run_5	run_6	run_7	run_8
$loop_1$	x	x			x			x
$loop_2$			x	x		x	x	
$loop_3$	x		x	x	x		x	x

Figure 3.4: An illustration of the difficulty for seminal behavior recognition caused by data sparsity.

We circumvent the difficulty by exploiting the transitivity of correlations. The basic observation is that if event A has strong statistical correlation with event B and event B strongly correlates with event C, event A and event C tend to correlate. For the example in Figure 3.4, as the samples of $Loop_3$ overlap with those of both $Loop_1$ and $Loop_2$, we can use the overlapped runs to compute the correlation between $Loop_3$ and $Loop_1$, and the correlation between $Loop_3$ and $Loop_2$. If the two correlations are both high, it can be inferred that $Loop_1$ and $Loop_2$ have strong correlations as well.

Figure 3.5 outlines our algorithm for identifying seminal behaviors. The algorithm iteratively partitions all sampled loops into a number of families. The loops in a family have strong correlations with one another in terms of trip-counts. During this process, the algorithm examines every pair of

```
//IB: the interface behavior set
//$H_c$:  a predefined correlation threshold
Procedure SemRec( ){
  semBeh = {};
  LoopFam = buildLoopFam( );
  foreach f in LoopFam {
   l = getRepresentive (f);
   c = calCor (l, IB);
   if (c< $H_c$) {
    s = getEarliest (f);
    semBeh = semBeh ∪ s;
   }
  }
}

Procedure buildLoopFam (){
  loopList = sortLoops (); // based on ID
  foreach $l_1$ in loopList {
   loopList = loopList - $l_1$;
   for each $l_2$ in loopList {
    c = calCor ($l_1$, $l_2$);
    if (c > $H_c$) {
     f = getFamily ($l_1$); // create one if none
     addToFam ($l_2$, f);
    }
   }
  }
}
```

Figure 3.5: Algorithm for seminal behavior recognition.

loops in order of loop ID. For each pair, it feeds the data collected from their overlapped runs to a correlation analyzer (a component of the idle-time analyzer). If the analyzer regards that the loops have strong correlations (either linear or non-linear), the loop with the larger ID is added to the family to which the loop with the smaller ID belongs. A new family is created if there is no such family.

After the loops are partitioned into families, the next step is to determine the seminal behaviors. This step starts with the interface behaviors, which are put into seminal behavior set by default. Recall that interface behaviors are typically cold behaviors and are collected in every run. The entire set of interface behaviors is regarded as one predictor. The algorithm examines the correlation between this predictor and one representative loop in each family. The representative is selected to be the loop that has the largest samples in the family for the stableness of the correlation analysis results. When a strong correlation is found, the whole family of loops are removed from further considerations as they are predictable from the current seminal behavior set. If the correlation is low or uncomputable (when there are too few overlapped runs), the earliest loop of that family is taken as a new seminal behavior and added into the seminal behavior set.

We elaborate on two details. First, the earliness of a loop is defined as the earliest time that its trip-counts is known. We make changes to the JIT so that the instrumentation inserts a load and store bytecode before and after each sampled loop to record how much time (in timerTicks in Jikes RVM) has passed since the start of the program. The earliness of a loop is computed as the average earliness of all samples of the loop in all runs. Selecting the earliest loop from a family as a seminal behavior helps the early use of the to-be-built input-opt models (see Section 3.3.2). Second, the correlation analyzer is a statistical tool we have developed. It consists of standard statistical functions for correlation analysis: Least Median of Squares (LMS) regression for linear regression, Regression Trees for non-linear regression, step-wise function and principal component analysis (PCA) for feature selection. They are similar to the analyzer in previous work [41]. Details are elided.

The technique designed in this chapter for seminal behavior identification shares certain commonality with the prior technique [41] in that both are based on statistical correlation analysis. However, there are two important differences. First, the previous technique works on dense data sets rather than sparse data sets. Second, the previous work builds affinity lists in a greedy manner. It cannot exploit the correlation transitivity and hence is not amenable to sparse data sets. For instance, for the example in Figure 3.4, the prior technique fails in recognizing the correlations between Loop_1 and Loop_2.

In the implementation of the algorithms, we use 0.8 as the value for the threshold H_c to judge whether a correlation is high enough. It is the same as the threshold value used in the previous work, easing the comparison between the two techniques (in Section 3.5). The complexity of the algorithm is $O(N^2)$, where N is the number of loops in the program. As this step happens during idle time of a machine, the complexity is typically tolerable.

3.3 Input-Opt Modeling and Adaptive Dynamic Optimizations

This section briefly discusses some issues the cross-run learning paradigm brings to the other two components of of transparent transparent input-centric dynamic optimizations. Although these issues are not as difficult as those discussed in the previous section, appropriate treatment to them is no less important for the transparent transparent input-centric dynamic optimizations to work effectively.

3.3.1 Input-Opt Modeling

Recall that the objective of input-opt modeling is to build up a predictive model mapping from the values of input features (i.e., seminal behaviors) to the appropriate optimization decisions (e.g., appropriate unrolling levels for a loop, suitable optimizing levels for a method, etc.) for an execution. The mapping can be represented as $B_{target} = f(B_{sem})$, with B_{target} for the value of a prediction target, B_{sem} for the values of seminal behaviors, and $f()$ for the predictive models. The goal of input-opt modeling is to determine $f()$.

Chapter 2 has treated this problem as a statistical learning problem. The solution is to collect a data set consisting of the values of B_{target} and B_{sem} in many runs of a program on different inputs, and then apply a statistical learning tool to the data set to compute $f()$.

The paradigm of learning across production runs imposes new implications to input-opt modeling in two aspects.

Data Collection The first array of implications relate with the collection of the data sets (B_{target} and B_{sem}). The seminal behavior set (B_{sem}) consists of mostly interface behaviors (which are usually outside hot code regions) and a small number of loop trip-counts. The overhead for collecting those behaviors is typically negligible.

Collection of target behaviors (B_{target}) is more complex; the overhead depends on what the target optimizations are. We categorize various optimizations into three classes.

- *Class 1)* For some optimizations, the default runtime environment eventually exposes the appropriate decisions. An example is the appropriate level for optimizing a Java method. Even though the default Jikes RVM cannot determine the appropriate level for a method during an execution because it does not know how much time that method takes in the entire execution, it can do so at the end of the execution. So, the final optimizing level the JVM decides on a method is usually the appropriate level for the entire run. For this class of optimizations, the collection of B_{target} is simple, just recording the final decisions at the end of an execution. As the overhead occurs only after the execution, it is typically negligible.

- *Class 2)* For some optimizations, the default runtime environment does not directly expose the appropriate decisions, but can produce such decisions as long as some necessary information is provided. One example is function inlining. The inlining decisions made by Jikes RVM during an execution may be inappropriate due to the lack of information. However, Jikes

RVM contains a model that produces the appropriate inlining decisions as soon as the hotness of all methods and their sizes are provided. Often, the information needed is recorded through the execution by default; the method hotness in Jikes RVM is such an example. If not, the behaviors have to be collected using the overhead-controlled sampling scheme as described in the previous section.

- *Class 3)* There are some other optimizations, the appropriate decisions of which are hard to model, and are often better to resort to empirical cross-trial comparisons. An example is the best unrolling levels of a loop. As the trials may negatively affect the production run performance, the number of trial runs must be minimized. For some target optimizations, the trials of different decisions can happen in one execution, such as loop unrolling for a loop that is invoked many times in a run. For others, the trials of different decisions may need to happen on different runs of the program. An example is the selection of the garbage collection that best fits an execution [53,69]. In this case, the comparison of the quality of the different decisions is tricky. If the trials happen on the same inputs, the comparison is simple. But because the trials are on production runs, the same inputs may not be seen until many runs later (or ever). Fortunately, from seminal behaviors values, one can infer the similarity or relations among different inputs, and hence make approximated comparison. Detailed explorations are out of the scope of this dissertation.

Model Self-Assessment and Evolvement The second fold of implications are on model construction. Because now training data come incrementally across runs, it becomes especially important to track the quality of the current models so that wrong predictions can be prevented from hurting the optimizations.

We use ten-fold cross-validation [35] to compute the confidence of each constructed model. Ten-fold cross-validation is a standard statistical approach. It uses nine tenth of all training data for model construction and the rest for testing. This process repeats for ten times. A standard statistical analysis is then applied to the testing results to derive a confidence value for that model.

Meanwhile, the model construction step records the boundaries of the part of the input feature space that has been covered by the training data. Prediction inside these regions is typically safer than outside. The usage of the boundaries is seen in the next section.

As more runs finish, more data are collected. With the input-opt models reconstructed periodically using the updated data set, the confidence levels and covered regions boundaries are updated accordingly. The self-assessment process happens in the idle-time analyzer and do not interfere with the production executions.

3.3.2 Adaptive Dynamic Optimizations

The third component of transparent input-centric optimizations is to employ the constructed input-opt models to guide runtime optimizers. During runtime, as soon as the seminal behaviors values become known, the runtime environment invokes the already constructed input-opt models to attain the appropriate optimization decisions and use them for runtime optimizations.

As an implication from the paradigm of learning across production runs, the usage of the models must be select as the model quality takes some runs to enhance. The confidence levels and region boundaries described in the previous sub-section come at handy. The principle is that the runtime environment uses a model only if the seminal behavior values of the current run falls into the covered region and at the same time, the confidence of the model is high enough (over 70% in our experiments).

3.3.3 A Concerted Assembly That Evolves Continuously

Another implication from the new paradigm is that unlike the prior offline schemes, the three components now must happen throughout the entire life time of an application. It is important to assemble them together into a concert to work synergistically.

As Figure 3.1 shows, a continuous learning framework unifies the three components together. Although all three components remain active through the life time of an application, the degrees of their activeness differ in different stages of the life time.

The initial certain number of runs of a program are purely dedicated to the first component for identification of the seminal behaviors. The second and third components do not need to be invoked as the seminal behaviors set is not available yet.

The criterion we use for the initial activation of the second component is as follows. Let $n_{i,j}$ be the number of runs during which both loop i and j are sampled. Let $\bar{n}$ be the average of all $n_{i,j}$. The value of $\bar{n}$ reflects the density of the accumulated data set. When it exceeds a predefined threshold (e.g., 3 in our experiments), the second component, input-opt modeling, gets activated. All the runs before the reaching of the predefined threshold form the *initial stage* of the optimization paradigm.

In every following run, the runtime optimizer tries to use the current input-opt model for optimizations. After every such run, the collected seminal behaviors and the observed learning target (e.g., method optimization levels) are put into the local database. Periodically, the local databases of different users are accumulated together (e.g., into a remote server), upon which, the idle-time analyzer refines the seminal behavior set and the input-opt models. So over time, the seminal behavior set may become smaller (as more correlations among seminal behaviors are discovered), the input-opt model may become more accurate, and the program is likely to run faster.

3.4 Other Complexities

The previous sections have described our solutions to some core obstacles. This section lists some other complexities related with practical deployment of the optimization paradigm. Resolving these complexities is beyond the scope of this dissertation. We list them, hoping that they may trigger some research interest of the community so that the new optimization paradigm can be practically materialized in the near future.

Data Communication and Profile Management Conceptually, the idle-time analyzer resides in a machine that connects with all the users of a target software. All the sampled data of that software are sent to this central machine periodically (when the local machine is idle) from all users for the analyzer to process. The processing results, including the IDs of the recognized seminal behaviors and the input-opt models, are sent back to all the users for helping their respective runtime optimizers (e.g., a JVM) to optimize the future executions of the software. As neither the samples nor the models are large, the amount of data transfer should be modest, unless the customer base is massive. The frequency of the communication can be configured to strike a good tradeoff between the timeliness of the model update and the communication cost. The concrete design of the communication system and the efficient way to manage profiles on the servers may depend on the scale of the problem, the frequencies of required updates, and so on.

Differences in Platforms and Libraries The second complexity for real-world deployment of the paradigm is the differences among platforms and software copies. Two users may happen to run a program on two different architecture or libraries; the data collected may have to be reconciled. Studies (e.g. [89]) in matching profiles across platforms may be helpful. Another possible solution is to concentrate on behaviors that are largely platform-independent (e.g. method calling frequency) during input-opt modeling. Prediction from such models may still be useful as the runtime optimizer has the knowledge of the specific platform, and hence may translate the predicted program-level behaviors into platform-specific optimization decisions.

Software Update The third complexity comes from software update. Software update may cause changes to the behaviors of the program, hence invalidating some results learned so far. But on the other hand, an update to a software rarely changes the program entirely. It is worth exploring how the continuous learning framework can adapt to the changes smoothly, without discarding the entire profile database and starting from scratch. Some techniques (e.g., code matching) in software test prioritization (e.g. [74]) may be helpful to solve this problem.

Server Applications and Program Phases The inputs to a server application typically come continuously through an entire execution. The transparent input-centric continuous optimizations may need to happen at the arrival of each input. Similarly, for a program with phase shifts, the integration of the phase knowledge into the paradigm may be necessary.

3.5 Evaluation

Our evaluation focuses on the effectiveness of the techniques for overhead control, and the feasibility and potential of the transparent transparent input-centric paradigm for dynamic optimizations in some basic settings. Specifically, we aim at answering three-fold questions:

1) Control of Overhead. Can the runtime data collection quickly collect many samples without causing too much interference to production runs?

2) Potential for Optimizations. How effective is the paradigm in characterizing program inputs and exerting the power of input-consciousness? Can the paradigm continuously enhance program performance? Is the enhancement significant?

3) Prevention of Risks. How effective is the selective prediction in preventing wrong predictions from hurting program performance?

3.5.1 Methodology

Platform Our implementation is on Jikes RVM (v. 3.1), which has been briefly described in Section 3.1.2. All experiments happen on machines equipped with Intel Xeon E5310 processors that run Linux 2.6.22; the heap size ("-Xmx") is 512MB for all.

Benchmarks A special obstacle for our experiments is in finding benchmark suites. Because of the focus on input influence, we require many different inputs per benchmark. However, most existing benchmark suites come with no more than three inputs.

A Java benchmark suite that comes with many inputs is the one developed in Chapter 2 on offline input characterization. The suite contains 10 Java programs selected based on the criterion that extra inputs for these benchmarks are relatively easier to collect than for other benchmarks in the original suites, and meanwhile, the benchmark comes with source code as it is necessary for the previous analysis. Despite the previous efforts, some of the programs in the suite (e.g., Search) still have only a small number of inputs. We include them for completeness.

In addition to including the benchmarks in the previous chapter, we also add some other complex Java programs from the Dacapo (2006) benchmark suite. For each of the added benchmark, we try to collect extra inputs that are typical in the normal executions of the benchmarks. More specifically, we collect or derive the inputs by searching the real uses of the corresponding applications, and reading the source code of the programs and example inputs. For the usage of the benchmarks to be close to that of real applications, some programs (e.g., Mtrt, Eclipse) are modified to reactivate some of their command-line options that were disabled by the benchmark suite interface. Table 3.1 lists all the benchmarks used. These benchmarks cover a variety of domains, from utility tools to compiler tools to computational applications.

Table 3.1: Benchmarks and Their Properties

Program	Description	Code lines	# Inputs	#Runs in init stage	Sampled loops per run (%)
Compress^j	compression tool	927	20	618	6.7
Db^j	database tool	1028	54	80	18
Mtrt^j	multithreaded ray tracer tool	3842	100	1161	4.2
Euler^g	computational fluid dynamics	1179	20	55	17.4
MolDyn^g	molecular dynamics simulation	583	20	38	21
MonteCarlo^g	Monte Carlo simulation	3073	21	204	11.1
Search^g	Alpha-Beta pruned search	712	9	106	20
RayTracer^g	3D ray tracer	1224	21	83	16.7
Antlr^d	parser generator	32263	175	1270	4.8
Bloat^d	bytecode-level optimization	73563	100	1815	3.9
Eclipse ^{d}	multi-language IDE	1903219	80	1856	1.5
Fop ^{d}	print formatter	88846	70	943	3.4
Hsqldb ^{d}	SQL relational database engine	151915	75	1146	2.9
Jython ^{d}	python interpreter in Java	91982	60	1258	1.9
Luindex ^{d}	text indexing tool	8570	50	645	3.2
Lusearch ^{d}	text search tool	12709	63	1630	1.8
Pmd ^{d}	Java source code analyzer	49331	53	1051	4.3
Xalan ^{d}	transform XML documents	243516	60	924	3.1

j: jvm98 [3]; d: dacapo [13]; g: grande [2]

Table 3.2 lists the number of inputs used for each benchmark in the 2nd column. The inputs exhibit large variations, reflected by the large differences in the corresponding running times shown in the 3th and 4th columns of the table. To further examine the input sensitivity among different inputs, we run the 18 Java benchmarks on all the inputs using Jikes RVM 3.1.0 (enabling option *enable_advice_generation=true*) and collect all the advice files. The advice files contain all the method optimization levels generated by the adaptive optimization system in Jikes RVM using a cost-benefit model [10]. In each advice file, the optimization levels indicate the hotness of all the methods that have been executed in that run, which intuitively show a holistic view of method level program behaviors. The 5th Column in Table 3.2 reports the total number of methods executed for each Java benchmark. Column 6 to Column 10 show a detailed breakdown of the methods based on their optimization levels. Column 6 reports the percentage of the methods whose optimization levels change across different program inputs. Column 7 reports the percentage of methods whose optimization levels are always -1 across all inputs. Similarly the 8th to the 10th column report the percentage of methods whose optimization levels are always 0, 1 and 2 respectively.

Table 3.2: Benchmarks and Variances of Their Inputs

Program	# Inputs	Running time (s)		# Methods	Methods with Opt Level (%)				
		Min	Max		nonConst	-1	0	1	2
Compress^j	20	0.94	9.33	160	6.25	91.88	1.88	0	0
Db^j	54	0.59	98.16	104	29.81	69.23	0	0	0.96
Mtrt^j	100	0.26	6.37	581	18.07	81.93	0	0	0
Euler^g	20	0.93	7.79	117	58.97	41.03	0	0	0
MolDyn^g	20	0.11	63.05	144	29.17	70.83	0	0	0
MonteCarlo^g	21	9.07	15.81	232	51.29	46.55	0	2.16	0
Search^g	9	2.74	210.36	383	7.18	92.55	0	0	0.27
RayTracer^g	21	3.1	236.57	126	32.54	66.67	0	0.79	0
Antlr^d	175	0.266	3.865	1403	41.70	58.09	0.14	0.07	0
Bloat^d	100	0.08	41.46	1984	26.31	73.59	0.10	0	0
Eclipse ^{d}	80	0.572	86.648	12284	72.44	27.55	0.01	0	0
Fop ^{d}	70	0.385	2.039	4218	59.98	40.02	0	0	0
Hsqldb ^{d}	75	0.455	8.888	1731	60.77	39.23	0	0	0
Jython ^{d}	60	0.594	34.02	13202	83.25	16.75	0.01	0	0
Luindex ^{d}	50	0.363	7.299	1070	53.55	46.45	0	0	0
Lusearch ^{d}	63	0.482	1.443	887	57.16	42.84	0	0	0
Pmd ^{d}	53	0.323	4.475	3471	59.03	40.97	0	0	0
Xalan ^{d}	60	0.229	5.723	2966	57.79	42.21	0	0	0

j: jvm98 [3]; d: dacapo [13]; g: grande [2]

For the Java benchmarks that have fewer methods, it's feasible to show a complete picture of the method optimization levels using a checkerboard graph. For example, Figure 3.6 shows the checkerboard graph of method optimization levels for euler, raytracer and montecarlo. The color of a rectangle in the checkerboard graph represents the optimization level of a method (X-axis) when the program runs with a program input (Y-axis). Each rectangle can have one of five colors. White color indicates that this method is not executed in that run. Grey color represents that a method has an optimization level -1 in that run. Blue color corresponds to method optimization level 0. Green and red color correspond to method optimization levels 1 and 2 respectively. The checkerboard graph of the three Java programs exhibit a broad range of diversities for different program inputs. Overall, Table 3.2 and Figure 3.6 show that most of the programs exhibit a considerable degree of input sensitivity.

Experimental Setting We use a controlled environment for experiments. It helps us concentrate on the main goal of the evaluation (i.e., the questions listed at the beginning of this section), without getting distracted by the complexities beyond the scope of this dissertation, such as the design of the distributed communication system, variations in platforms and library versions.

In the controlled setting, there are 100 virtual users, running a benchmark on identical platforms. Instead of using 100 machines and getting distracted by complexities in data communications, we put all runs on a single machine. Each time, one virtual user runs the benchmark once, on an input randomly selected from the input set. The profiles from all runs are accumulated into a single database.

We acknowledge that the setting has apparent distance from practical settings; but we maintain that the setting is still usable for answering the three-fold questions in the focus of this study. For instance, the overhead incurred by the sampling scheme is about the current execution by the current user, largely independent of how all users are connected, how profiles are managed, or any other complexities excluded by the controlled setting; the same for the evaluation of risks prevention. We acknowledge that the exact benefits from the optimizations may differ from those in real settings. However, the measurement in this controlled setting can still indicate whether the paradigm is promising in continuously enhancing program performance, and whether this direction is worth further investigations.

3.5.2 Data Collection Efficiency and Incurred Overhead

This sub-section concentrates on the efficiency of the data collection scheme. Specifically, it examines the effectiveness of the two sampling techniques proposed in this chapter, randomization and inspection-instrumentation, in helping achieve a large coverage quickly without causing too much sampling overhead.

Overhead Among the executions of the continuous optimization paradigm, the initial stage is subject to the largest risks of exhibiting slowdowns due to the instrumentation for data collection. Our evaluation of overhead concentrates on that stage.

Because multiple runs of a Java program tend to show considerable variations of running times even if all those runs are on the same input, we use a statistical approach advocated by some previous studies [31] to examine the influence of the overhead. For each program, we randomly pick one input. We run the program on that input for 20 times using the default Jikes RVM, and record the times. We then use the same program and input to conduct 20 runs with the randomized sampling based on the inspection-instrumentation scheme. Figure 3.7 shows the distribution of the running times in the two scenarios. The times are normalized with the average time of the 20 default runs.

We use the standard statistical hypothesis testing to examine whether a program's performance in the two scenarios differs significantly. The approach applies T-testing to the time samples to compute a statistical metric, *p-value*. The higher the p-value is, the less likely the two kinds of runs differ significantly in time. A typical statistical practice is to reject the hypothesis that the two differ significantly if p-value is greater than 0.05 [35]. As shown in the bottom of Figure 3.7, only the p-values of *Euler*, *Search*, and *Eclipse* are less than 0.05. The average time differences of the three programs are respectively 0.85%, 0.97%, and 4.7%, confirming that the inspection-instrumentation scheme effectively limits the overhead of most programs to be negligible.

Collection Efficiency The second to the rightmost column in Table 3.1 reports the total number of sampling runs that the initial stage of our continuous optimization paradigm requires before the second and third components can start. (Recall that the criterion is that on average, a pair of loops must have been sampled in at least three common runs.) As we have 100 users, on average each user needs to have 0.38 to 18.56 runs to reach that coverage. As a comparison, if the randomized scheme is not used and every user's sampling starts from the beginning of the program, based on the rightmost column in Table 3.1, the estimated number of total runs would be 1.4X to 7.5X more for them to cover every loop just at least once.

It is worth noting that because of the randomization in our sampling scheme, the average number of needed runs per user decreases almost linearly as the number of users increases, as Figure 3.8 shows. (For legibility, the figure shows only 5 benchmark curves. The others have the similar trend.) But the average number remains virtually constant when the randomization scheme is not used, because no matter how many users there are, the sampling window always moves through the entire program gradually and sequentially for every user's executions.

3.5.3 Prediction Accuracy and Performance Enhancement

As Section 3.3.1 describes, the transparent input-centric dynamic optimizations may be applied to help different classes of optimizations. In this experiment, we take a specific optimization decision problem as a concrete example to examine the basic effectiveness of the new optimization paradigm.

3.5.3.1 Target for Enhancement

In this example use, the objective is to enhance the compilation strategy in Jikes RVM. In Section 3.1.2, we have mentioned that Jikes RVM realizes the importance of a Java method only gradually after some invocations of the method. The weakness causes two kinds of inefficiency. First, because the JIT recompiles a method at a higher optimizing level when it sees the increased importance of the method, a method may be recompiled multiple times in one run, causing unnecessarily large compilation overhead. Second, the highly optimized code is produced late, throttling the benefits of the optimizations. An extreme case is that many methods that are used heavily in the initialization stage of an application may get highly optimized at the end of the stage; but after that, the methods are never invoked again [34].

As many dynamic optimization systems use the similar strategy, this weakness is shared by almost all of them. Several studies [9, 34, 78] have reported the importance of this weakness. For instance, Arnold and others [9] have reported over 47% potential speedup when the weakness can be overcome in IBM commercial JVM, J9. Despite many recent efforts on this issue, the state-of-the-art solutions require either extensive offline profiling as in Chapter 2 or are subject to input-obliviousness [9].

3.5.3.2 A Solution from the New Paradigm

In this experiment, we try to apply the transparent transparent input-centric paradigm to overcome the limitations of existing solutions.

Specifically, we set the appropriate optimization level for each Java method as the prediction target in the input-opt models. As described earlier, the models are transparently built across production runs of the program. When a Java method is encountered and the values of the seminal behaviors of the current run are known already, the modified Jikes RVM uses the input-opt models to predict the best optimization level for the method. If the prediction is confident, the JIT optimizes the method at that level immediately. Otherwise, the default compilation scheme is applied to the method.

This approach helps avoid repetitive recompilations of a method. At the same time, as seminal behaviors typically become known at the early stage of an execution, this approach helps the JIT produce optimized code early, hence alleviating both kinds of inefficiency of the default strategy. It overcomes the limitations of prior solutions in Chapter 2 by removing the needs for offline profiling and enabling input-consciousness.

3.5.3.3 Results

Figure 3.9 reports how the programs performance changes across runs as the knowledge base (i.e., the input-opt models) grows incrementally. The X-axis starts with the first run following the finish of the initial stage of the paradigm. In the experimental setting, the idle-time analyzer refines the knowledge base after every five runs. For lack of space, it contains only the figures for the top 12 benchmarks listed in Table 3.1. The results of the other benchmarks show the similar trend.

The confidence and accuracy curves in each figure show the quality of the constructed input-opt models. The Y-axis value of each point on the accuracy curve is the percentage of the Java methods whose appropriate optimization levels are predicted correctly in the corresponding run. Being correct here means that the predicted optimization level of a method equals the ground truth, which is obtained through the default Jikes RVM as explained in the "Class 1" bullet in Section 3.3.1. The confidence value is computed using cross-validation on the existing data base, as described in Section 3.3.2. The arising trend exhibited by the curves indicates that the continuous learning framework is able to incrementally increase the quality of the input-opt models. Some runs' prediction accuracies are zero because in those runs, the runtime system finds that their seminal behavior values fall out of the space that the previous runs have covered. As Section 3.3.2 describes, thanks to the self-assessment and selective prediction scheme, in such cases, the runtime system does not do prediction and falls back to the default execution; no performance penalty is incurred. Similar fallback executions happen for those runs in which the confidence is lower than the threshold (0.7).

As the model becomes good enough, the JIT starts to use the predicted levels to do optimizations. The resulting speedup starts to show. On different inputs, the speedup differs. Overall, for most of the programs, significant speedups are exhibited.

Comparisons Even though the speedup brought by the transparent input-centric optimizations is quite significant as Figure 3.9 shows, the benefits come from multiple sources. It is unclear how much benefits the input-consciousness really brings. Will a simple input-oblivious refinement of the default recompilation scheme be sufficient? And how much benefit of transparent input-centric optimizations is compromised because of the data loss caused by the cross-run sampling scheme?

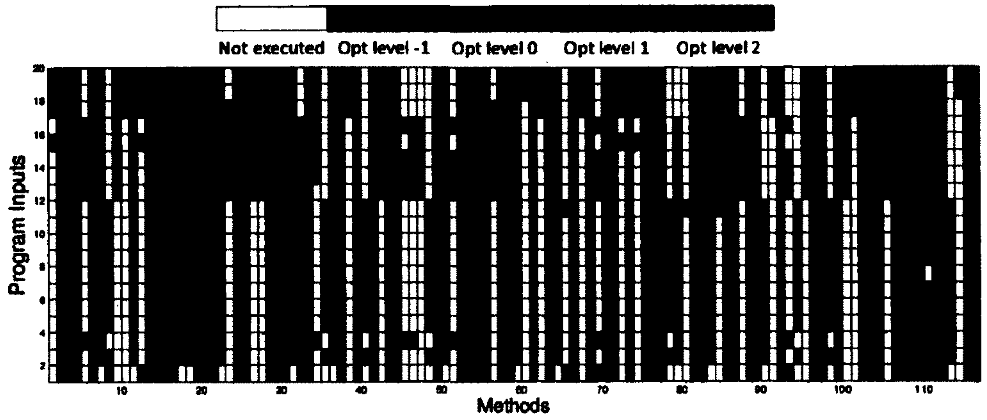
To answer the two questions, we compare the speedups brought by our technique with two other results. One is from the repository-based approach by Arnold and his colleagues [9]. It learns from a repository of history runs, but does not tailor optimization strategies to program inputs. More specifically, it produces an optimization strategy for each method in a program based on some optimization histograms that are built through history runs of the program. The optimization strategy contains a number of pairs. Each pair, say $< k, o >$, indicates that the method should be (re)compiled using level o when the sampler in the RVM encounters the kth samples of the method. The cross-run learning in the technique ensures that the produced optimization strategy produce the best average performance for history runs. Prior studies have shown that this scheme enhances the optimization by Java Virtual Machines (J9) substantially. Despite being a good refinement to the default recompilation scheme, it is input-oblivious. The comparison with this approach will indicate the value of being transparent input-centric. The authors of the technique did their implementation in IBM J9; we implement their approach on Jikes RVM by following their paper.

The other result to compare with is from the offline profiling approach in Chapter 2. We use detailed instrumentation to run each program on all its inputs to collect a complete training data set, then apply the techniques proposed in Chapter 2 to characterize the inputs and build predictive models for optimization level selection. After that, we use the models to help JIT in the same way as in our technique. As the complete data set is used for training, the obtained performance enhancements are expected to be the upper-bound for our approach. A comparison with these results will indicate the benefit compromise caused by the lightweight data collection in our approach.

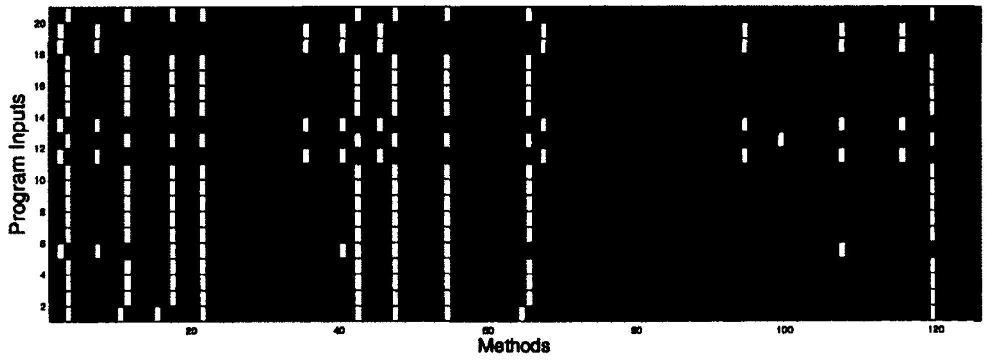
Figure 3.10 shows the minimum, mean, and maximum speedups from the three techniques on the benchmarks (40 runs per program). The cross-input adaptivity helps our technique to outperform the repository-based approach substantially, accelerating the programs over their default runs by 7–21% on average. Some of the results, especially those of "repository" results, show less than 1 speedup. Those indicate that some slowdown is caused due to wrong predictions. In most cases, the minimum speedups of our approach are higher than those of the "repository" approach, demonstrating that the selective prediction technique helps our technique to avoid negative effects from prediction errors. The small average distance from the offline profiling-based results indicates that the automatic components in our technique are able to well exert the potential of the transparent input-centric continuous optimization paradigm.

3.6 Summary

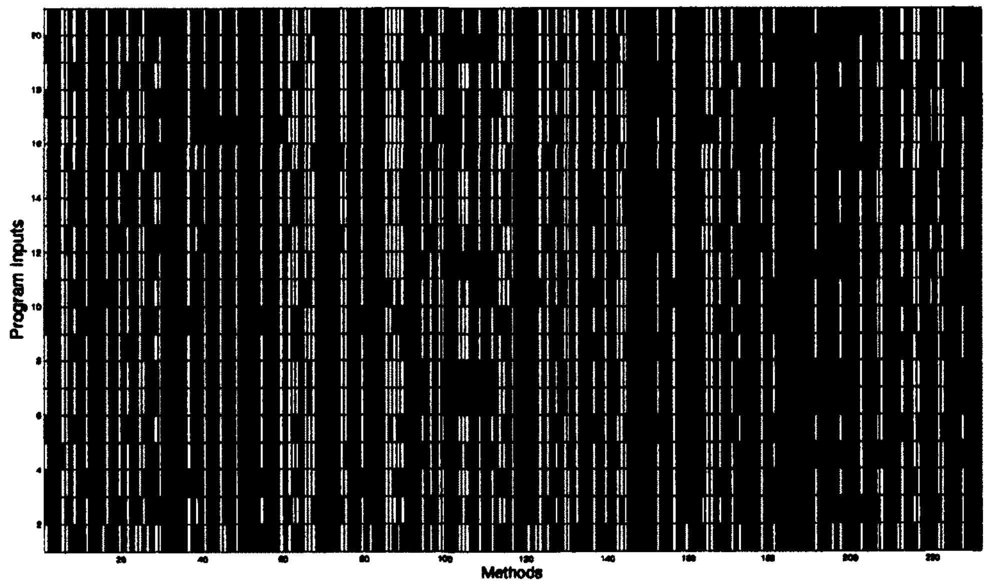
In this chapter, we report an investigation in the basic feasibility of transparent integration of input-consciousness into dynamic program optimizations, particularly in managed execution environments. The underlying vehicle of the new approach is transparent learning across *production runs*. After examining the implications of the new paradigm on each main component of transparent input-centric dynamic optimizations, we propose several techniques to address some key challenges, including randomized inspection-instrumentation for cross-user data collection, a sparsity-tolerant algorithm for input characterization, and selective prediction for efficiency protection. Together, these techniques make it possible to automatically recognize the relations between the inputs to a program and the appropriate ways to optimize it. The new approach eliminates the needs for offline profiling or programmers' annotations, overcoming some limitations of prior solutions. Meanwhile, the paper points out some complexities that require further explorations. Experiments in a JVM demonstrate the feasibility and potential benefits of the new optimization paradigm in some basic settings.



(i) euler



(ii) raytracer



(iii) montecarlo

Figure 3.6: The checkerboard graph of method optimization levels across different program inputs.

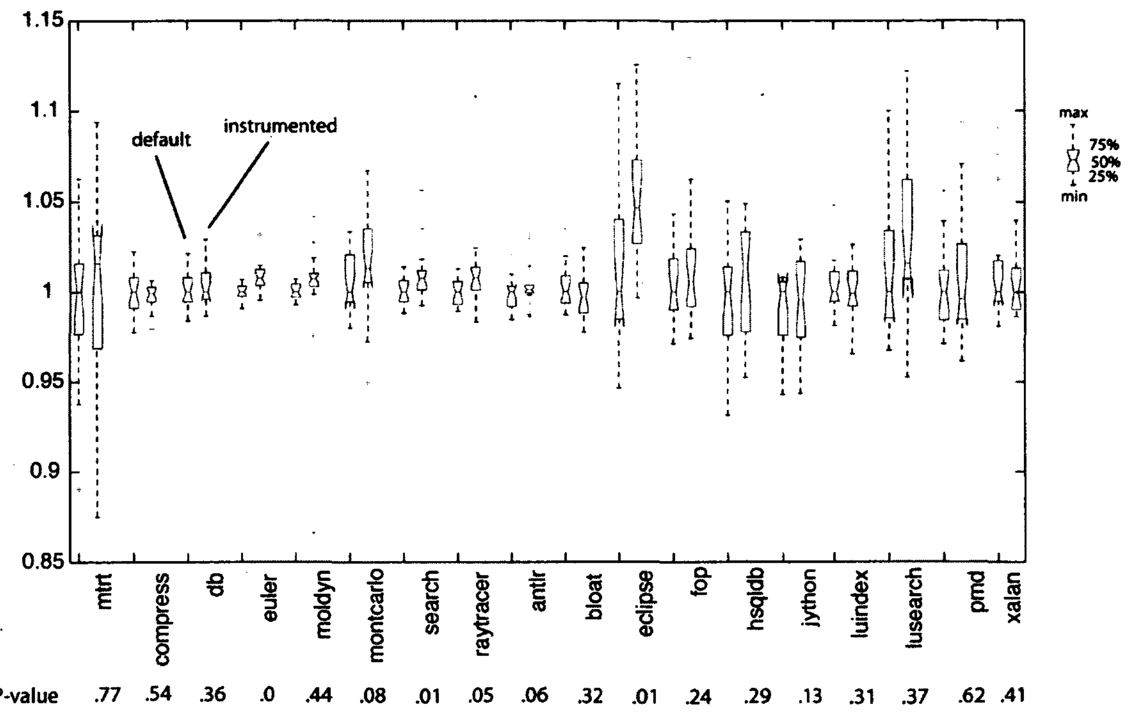


Figure 3.7: Distributions of the running times of the default and instrumented runs. The P-values at the bottom indicate whether the two differ significantly (< 0.05) or not (> 0.05).

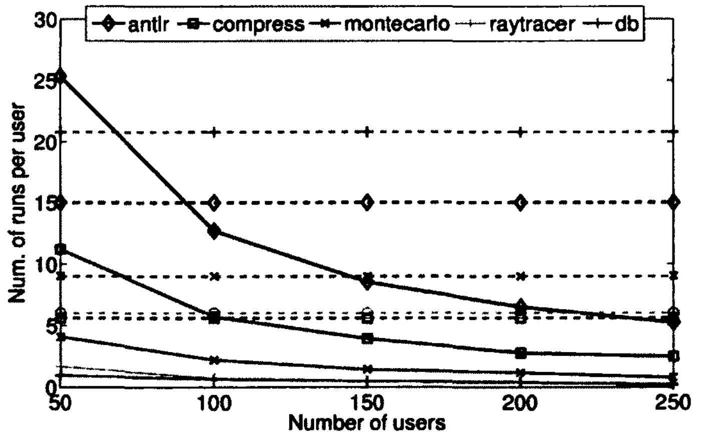


Figure 3.8: The number of runs per user in the initial stage. The solid-line curves are the results when randomization is used; the broken-line curves are when randomization is not used.

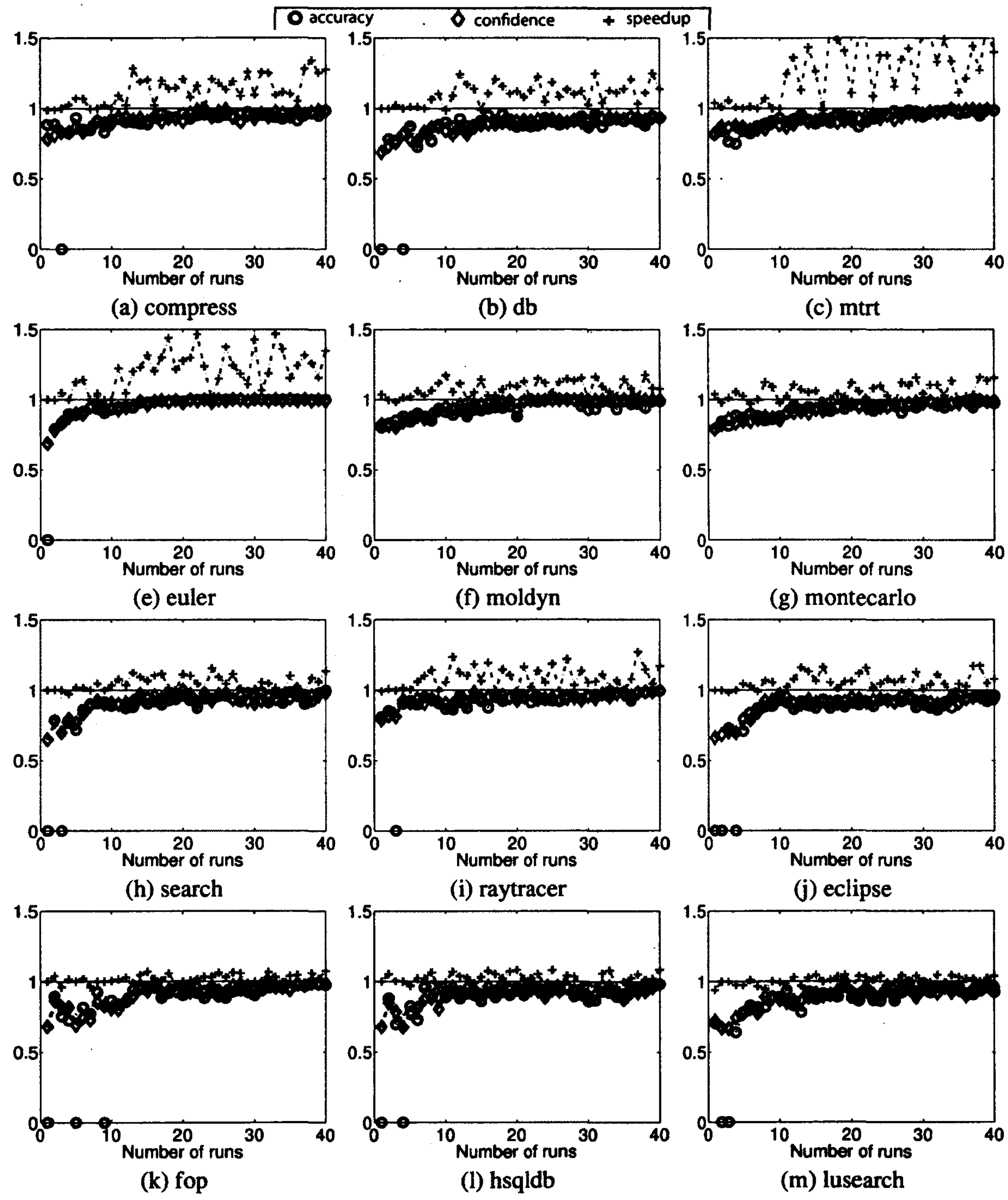


Figure 3.9: The cross-run changes of the prediction confidence and accuracy of the input-opt models, along with the corresponding performance enhancement over the executions in the default Jikes RVM.

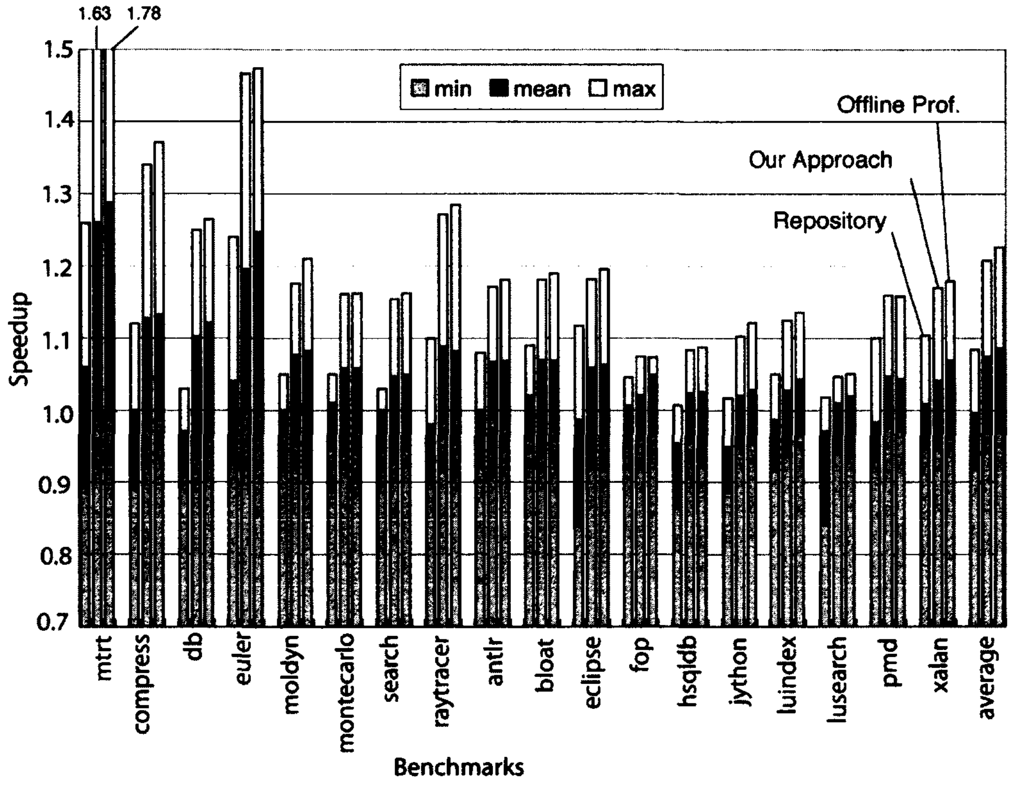


Figure 3.10: The overall speedups from repository-based approach, transparent input-centric continuous optimizations, and the upper-bounds obtained through offline-profiling–based experiments.

Chapter 4

Optimal Job Co-scheduling on CMP

The previous chapters focus on program optimizations and performance improvement at program level. This chapter wants to address the new challenges in performance improvement on job scheduling in modern Chip Multiprocessor (CMP) architectures. It gives a study on optimally co-scheduling jobs of different lengths on Chip Multiprocessors.

Cache sharing in Chip Multiprocessors brings cache contention among corunning processes, which often causes considerable degradation of program performance and system fairness. Recent studies have seen the effectiveness of job co-scheduling in alleviating the contention. But finding optimal schedules is challenging. Previous explorations tackle the problem under highly constrained settings. In this chapter, we show that relaxing those constraints, particularly the assumptions on job lengths and reschedulings, increases the complexity of the problem significantly. Subsequently, we propose a series of algorithms to compute or approximate the optimal schedules in the more general setting.

Specifically, we propose an A*-based approach to accelerating the search for optimal schedules by as much as several orders of magnitude. For large problems, we design and evaluate two approximation algorithms, A*-cluster and local-matching algorithms, to effectively approximate the optimal schedules with good accuracy and high scalability. This study contributes better understanding to the optimal co-scheduling problem, facilitates the evaluation of co-scheduling systems, and offers insights and techniques for the development of practical co-scheduling algorithms.

4.1 Introduction

With the popular adoption of Chip Multiprocessors (CMP) and Simultaneous Multithreading (SMT) in modern processors, it is typical for multiple computing units to share a single cache and other resources. The sharing, although helpful for the reduction of inter-thread latency, often causes resource contention among *co-runners* (referring to the processes or threads running together with some on-chip resources shared with one another), resulting in considerable degradation in program performance and system fairness [17,26,28,36,62,76]. As processor-level parallelism grows continuously, the problem becomes increasingly important.

In recent studies, researchers have demonstrated the promise of job co-scheduling in alleviating the problem of contention. Job co-scheduling capitalizes on the differences among jobs. By cleverly assigning compatible jobs to cores (or hyperthreads), researchers have seen considerable improvement in computing efficiency [26,70,87], fairness [71], and performance isolation [28].

However, most previous studies have concentrated on the empirical aspect of the problem: How to conduct runtime explorations to heuristically infer the goodness of different co-run schedules and select a good one. Although such explorations are critical for producing practical co-scheduling systems, they are not enough for the assessment of the potential of co-scheduling—that is, the benefits an optimal schedule can bring.

Finding optimal schedules is important for two reasons. First, it enables a more thorough evaluation of co-scheduling systems than before. The baseline of most evaluations of current co-scheduling systems is just random schedulers. But in the design of a practical co-scheduling system, it is important to know the room left for improvement—that is, the distance from the optimal solution—to determine the efforts needed for further enhancement and the tradeoff between scheduling efficiency and quality. Without finding an optimal co-schedule, such an assessment is impossible to do. Second, although optimal co-scheduling algorithms may not be efficient enough for direct deployment in production uses, they can help understand the computational complexity of the co-scheduling problem, and offer insights to the enhancement of practical co-scheduling systems.

Unfortunately, the current exploration of optimal
co-scheduling is still preliminary. A recent study [40] has started to tackle this problem, but in a highly constrained setting, where all jobs are of the same length and no reschedulings are allowed. The two conditions make the problem tractable, but meanwhile, cause a significant departure from real scenarios.

The *goal* of this chapter is to uncover the complexity of, and develop scalable solutions to, the optimal co-scheduling problem in a more general setting, where jobs may have different lengths and can be rescheduled multiple times. The complexity of the problem increases significantly as the schedule space in this setting is exponentially larger than that in the special case. The schedules produced by the previously developed optimal co-scheduling algorithms [40] lose the optimality in this more general setting (showed in Section 4.5.1.)

This chapter presents our two-fold attack to the optimal co-scheduling problem. First, we formulate optimal co-scheduling as a tree-search problem and develop an A*-search-based algorithm to determine optimal co-schedules for small problems, with orders of magnitude less overhead than a brute-force approach. A* search [63] is a technique stemming from artificial intelligence. Its appeals include the guarantee of the optimality of both the search result and the search efficiency. One of its key components is a cost estimation function, which determines the effectiveness of the algorithm in pruning the search space. In this chapter, we formulate the cost function as a linear programming problem, which helps the co-scheduling algorithm outperform a brute-force approach by 5 orders of magnitude in efficiency, with an exactly optimal solution produced. We are not aware of any previous A*-search-based algorithms for job co-scheduling.

Second, we develop two approximation algorithms and examine their scalabilities and effectiveness in co-scheduling large problems. The A*-search-based algorithm mentioned in the previous paragraph is hard to be applied to large problems due to memory requirement. But with the insights shed from it, we design two approximation algorithms, namely A*-cluster and local-matching algorithms. The A*-cluster algorithm integrates online adaptive clustering into the A*-search-based approach to trade accuracy for scalability. The local-matching algorithm, driven by local optimum, applies graph theory at every scheduling time to find the schedule that best fits the remaining jobs without future rescheduling considered. Experiments on Intel quad-core and hyperthreading machines show that both algorithms produce near-optimal results, significantly outperforming random schedules. The local-matching algorithm, in particular, consistently outperforms the state-of-the-art co-scheduling algorithms [40], and meanwhile, shows excellent scalability.

The two-fold explorations in this work contribute better understandings to the optimal co-scheduling problem. The proposed A*-search-based algorithm and the local-matching algorithm offer useful tools for computing (or approximating) optimal co-schedules. They not only enable the assessment of the potential of practical co-scheduling algorithms, but also shed insights to the development of future practical co-schedulers on CMP and SMT systems.

In the rest of this chapter, Section 4.2 defines the problem setting. Section 4.3 analyzes the complexity of the optimal co-scheduling problem and presents the A* search based approach. Section 4.4 presents two approximation algorithms for solving large problems. Section 4.5 reports the experimental results. Section 4.6 discusses the limitations of this work and future extensions. Section 4.7 reviews the related work, followed by a short summary.

4.2 Problem Setting and Notations

4.2.1 Concept of Co-Run Degradation

On a CMP system, a process often runs slower when there are other corunning processes that compete with it for shared resources, such as on-chip cache or bus. This degradation is called *co-run degradation*, defined as the difference between its corun time and its single-run time, denoted as $(cT - sT)$ (c for corun, s for single-run.) The *co-run degradation rate* is defined as

$$corun\ degradation\ rate\ of\ job\, i = \frac{cCPI_i - sCPI_i}{sCPI_i},$$

where $cCPI_i$ and $sCPI_i$ are the numbers of cycles per instruction (CPI) when job i has or has no cache sharers (i.e., jobs co-running on the same cache) respectively; c stands for co-run and s stands for single-run.

In optimal co-scheduling, the degradation of every possible co-run and the length of the single run of each job are known beforehand. They may be obtained by offline profiling runs. Recall that the direct goal of optimal co-scheduling is to find the upper limit of co-scheduling for facilitating the evaluation of practical co-scheduling systems, rather than to produce a co-scheduler directly applicable in real runtime systems. Therefore, offline profiling—adopted in this work—is usually affordable. (But the trial of all possible co-schedules is typically infeasible because of the exponential increase of the co-scheduling space.)

It is worth noting that many recent studies have attempted to approximate single-run and co-run information through either runtime analysis [28,43,61,70] or statistical models [17,75]. Combined with those techniques, efficient optimal co-scheduling algorithms may show the promise of practical applicability (even though practical applicability is not the main goal of this research.) Detailed studies are out of the scope of this chapter.

4.2.2 Optimal Co-Scheduling Problem Tackled in this Work

The problem this work attempts to solve is defined as follows. There are **N** processes to be assigned to *M* cores, and $N = M^1$, one process per core at each time. Every **K** (a factor of *N*) cores reside on a chip, *sharing* a cache. Let **I** be the number of chips, so $I = N/K$. The processes may finish at different times, therefore, reassignments may be necessary. There are various goal functions in job co-scheduling. In this work, the goal is to find a schedule that minimizes the total completion time of all jobs [2], as expressed as

$$\arg\min_{S} \sum_{i=1}^{N} cT_i^{(S)},$$

where, $cT_i^{(S)}$ is the time job i takes to finish in a co-schedule S.

To avoid distractions from other complexities, we make the following assumptions. First, the performance of the processes on a chip is independent on how the other jobs are assigned on other chips. Second, all processes start at the same time running till their finish and there are no phase

[1]The techniques in this chapter are applicable to the cases when $N < M$ as well; one can simply consider that there are $M - N$ processes that consume no resource at all.

[2]It is assumed that the clock starts at time 0 for all jobs no matter whether they are actually running.

changes in a process. Third, as there are no phase changes, we assume that rescheduling occurs only when one process terminates, and process migration has negligible overhead. These assumptions keep the fundamental challenges of the general co-scheduling problem (Section 4.6 discusses them further.) The techniques developed under these assumptions lay the necessary foundations for the more general cases.

In the following description, we call each scheduling or rescheduling point as *a scheduling stage*. So, N jobs have at most N scheduling stages. If all degradations are non-negative (which is typical), the scheduling can stop at stage $N - I$ because the remaining I jobs can run alone with no degradations. We use *an assignment* to refer to a group of K jobs that are to run on the same chip. We use *a sub-schedule* to refer to a set of assignments that cover all the unfinished jobs and have no overlap with each other. We use *a schedule* for the set of all sub-schedules that are used from the start to the end of the executions of all processes. A schedule is a solution to the co-scheduling problem.

4.3 A*-Search for Co-Scheduling

In this section, we first examine the co-schedule space, revealing the challenges in finding optimal co-schedules. Then we present an A*-search-based approach to pruning the space and finding optimal co-schedules efficiently.

4.3.1 Co-Schedule Space

The optimal co-scheduling problem tackled in this work can be formulated as a tree-search problem as Figure 4.1 illustrates. For N jobs, there are at most N scheduling stages; each corresponds to a time point when one job finishes since the last stage. Each node in the tree, except the starting node, represents a sub-schedule of the remaining jobs. The nodes at a stage, say stage i, correspond to all possible sub-schedules for $N - i + 1$ remaining jobs. There is a cost associated with each edge, equal to the total execution time spent between two stages by all jobs. Let n_2 represent a child of node n_1. Given the state at n_1, we assign the remaining jobs according to the sub-schedule contained in n_2; let t be the time required for the first remaining job to finish; the cost on the edge from n_1 to n_2 is $t * m$, where m is the number of jobs that are alive during that period of time.

The goal of optimal co-scheduling is to find a path from the starting node to any leaf node so that the sum of the costs of all the edges on the path is minimum. As a comparison, the previously explored co-scheduling problem [40] contains only the starting node and the first stage in the tree (without rescheduling), the search space of which contains $n = \prod_{i=0}^{\frac{N}{K}-1} \binom{N-iK-1}{K-1}$ nodes. Whereas, the search space in the problem tackled in this work is exponentially larger with $O(n^N)$ nodes contained.

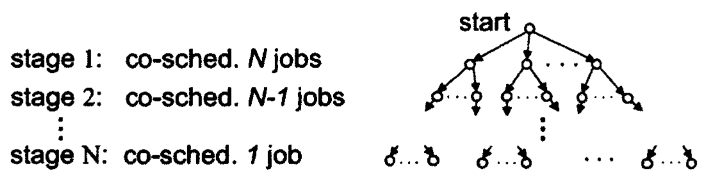


Figure 4.1: The search tree of optimal job co-scheduling with rescheduling allowed at the end of a job. Each node in the tree, except the starting node, represents a sub-schedule of the remaining jobs.

The significantly increased complexity makes previous approaches difficult to apply. As an example, Jiang et al. have used perfect matching on a fully connected graph to formulate the optimal co-scheduling [40]. That model, however, captures no influence from the difference among program lengths and rescheduling.

4.3.1.1 A* Search Algorithm

In this work, we treat the optimal co-scheduling problem as a tree-search problem, and apply A*-search algorithm to determine optimal co-schedules efficiently.

A* search is an appealing algorithm stemming from artificial intelligence [63]. It is designed for fast graph search. Under certain conditions, A* search guarantees the optimality of the solution, and meanwhile, effectively avoids the visit to some portion of the search space that are impossible to contain the optimal solutions. In fact, it has been proved that A* search is optimally efficient for any given heuristic function. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*, for a given heuristic function [63]. Its completeness, optimality, and optimal efficiency are the major reasons for us to choose A* search for the acceleration of the search for optimal schedules.

We use Figure 4.1 to explain the basic algorithm of A*. Each edge in the graph has a cost. The problem is to find the cheapest route in terms of the total cost from the starting node to the goal node. In A* search, each node, say node n, has two functions, denoted as $g(n)$ and $h(n)$. Function $g(n)$ is the cost to reach node n from the starting node. Function $h(n)$ is the estimated cost of the cheapest path from n to the goal. Therefore, the sum of $g(n)$ and $h(n)$, denoted as $f(n)$, is the estimated cost of the cheapest route that goes from the start to the goal and passes through n.

A* search has a priority list, which initially contains only the starting node. Each time, A* search removes the top—that is, the node with the highest priority—from the priority list, and expands that node. After an expansion, it computes the $f(n)$ values of all the newly generated nodes, and put them into the priority list. The priority is proportional to $1/f(n)$. The expansion continues until the top of the list is a goal node, which implies that its cost is no greater than the $f(n)$ of any other nodes in the list. The use of $f(n)$ is the key for A* search to prune the search space effectively. Figure 4.2 shows the algorithm of A* search.

```
/* start node contains all jobs to be scheduled */
Procedure Astar (){
  priorityList = { start };
  while priorityList.notEmpty() {
   x = priorityList.topItem();
   if (x.unfinishedJobs == 0)
    return x; // the solution route can be
              // traced from x
   priorityList.remove(x);
   while (y = x.nextSubSchedule())!= null {
    y.g_score = x.g_score + totalTime_between(x,y);
    y.h_score = estimatedTime_to_finishAll_from(y);
    y.f_score = y.g_score + y.h_score;
    priorityList.insert(y);}}
}
```

Figure 4.2: A* search for optimal schedules.

A good definition of function $h(n)$ is critical for the solution's optimality and the algorithm's efficiency. There are two important properties of A*:

- A* is optimal if $h(n)$ is an admissible heuristic—that is, $h(n)$ must never overestimate the cost to reach the goal[3].

- The closer $h(n)$ is from the real lowest cost, the more effective A* search is in pruning the search space.

Therefore, the key in applying A*-search is to develop a good definition of the function $h(n)$. Next, we describe how linear programming can be used to form a suitable definition of the function, and how A*-search is used in job co-scheduling.

[3]We assume that the search scheme is a tree-search. There are some subtle complexities for other schemes of search [63].

4.3.1.2 A*-Search-Based Job Co-Scheduling Algorithm

To use A* search in job co-scheduling, the key problem is to define the two functions, $g(n)$ and $h(n)$. The definition of function $g(n)$ is simple, just the total cost from the start to node n.

The definition of $h(n)$ is more interesting. When all corun degradations are non-negative, a simple definition is the sum of the single-run times of all the unfinished parts of the remaining jobs. This definition is legal as $h(n)$ must be no more than the real cost. But for efficiency, we need more sophisticated definition to make $h(n)$ closer to the real lowest cost.

Definition of $h(n)$ through Linear Programming A more sophisticated definition of $h(n)$ is through the form of linear programming. Suppose at node n there remain U unfinished jobs. We define $h(n) = T_s + T_d$. T_s is the time the U jobs need to finish their remaining parts if they each run alone. T_d is the estimated minimum of the total degradation that the U jobs have during their execution from node n to any child of n.

In this section, we concentrate on the common case when all degradation rates are non-negative. In this case, when U is not greater than the number of chips I, T_d is clearly 0 (no degradations as each chip has only one or no jobs.) Our following discussion is focused on the scenario where $U > I$.

Consider a sub-schedule represented by one of the children nodes of n. The degradation of all U jobs in the sub-schedule equals the sum of the degradations on all chips. The minimum degradation on one chip with b jobs can be estimated as follows. Let $T_{min(n)}$ represent the minimum of the single-run times of the unfinished part of all the remaining U jobs. Notice that the time lasting from n to any of its children must be no less than $T_{min(n)}$ because of corun degradations. Let $r_{b_{min}}$ be the minimum of the degradation rates of all jobs when a job coruns with $b - 1$ other jobs. It is clear that the degradation on the chip must be no less than $b * r_{b_{min}} * T_{min(n)}$, which is taken as the estimation of the minimum degradation of the chip. Therefore, the lower bound of the degradation of a sub-schedule j is $d_j = \sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(n)}$; b_i refers to the number of jobs assigned to chip i in the sub-schedule.

The value of T_d should be the minimum of d_j of all sub-schedules of node n. To determine the sub-schedule that has the smallest d_j, we need to find the values of b_i so that $\sum_{i=1}^{I} b_i * r_{b_{i_{min}}} * T_{min(n)}$ is minimized under the constraint that $\sum b_i = U$. This analysis leads to the integer linear programming shown in Figure 4.3. By relaxing the constraint on x_i to $0 \leq x_i \leq 1$, the problem becomes a linear programming problem, which can be solved efficiently using existing tools [1].

Definitions: (details at Page 3 bottom section)
U: number of unfinished jobs,
I: number of chips,
K: cores per chip, $I < U \leq I * K$,
$r_{x_{min}}$: minimum degradation rate,
$T_{min(n)}$: minimum single run time.

$$x_i = \begin{cases} 1 & : \text{the ith core has a job assigned} \\ 0 & : \text{otherwise} \end{cases}$$

The number of jobs on the c-th chip:

$$m(c) = \sum_{i=(c-1)*K+1}^{c*K} x_i$$

Objective function:

$$min \sum_{c=1}^{I} m(c) * r_{m(c)_{min}} * T_{min(n)}.$$

Linear constraint:

$$\sum_{i=1}^{N} x_i = U;$$

Figure 4.3: Integer linear programming for computing T_d, the lower bound of degradation.

As a special case, when $K = 2$, the solution to the integer linear programming is equivalent to the following simple formula:

$$T_d = 2 * (U - I) * r_{2_{min}} * T_{min}(n). \tag{4.1}$$

The rationale behind the formula is that in any sub-schedule of this scenario, there must be at least $(U - I)$ chips that have a pair of the unfinished jobs assigned. Otherwise, some chips must have more than 2 jobs assigned, which is not allowed in the problem setting (Section 4.2.) The application of the definition of T_d to such a sub-schedule leads to Equation 4.1.

With the definitions of the two functions, $g(n)$ and $h(n)$, we implement the A*-search-based optimal co-scheduling algorithm. Experiments, reported in Section 4.5, verify the optimality of the schedules produced by the algorithm.

4.4 Approximation of Optimal Schedules for Large Problems

One typical drawback of A*-search-based algorithms is the high requirement of memory space. The reason is that the algorithm requires to keep all open nodes (i.e. the nodes that still have unexpanded children nodes) into the priority list. As the problem size increases, the number of open nodes may become too large to fit into the memory.

In this section, we describe two approximation algorithms for solving the optimal co-scheduling problem in a scalable manner. One algorithm, the A*-cluster algorithm, comes from the insights shed by the A*-search-based algorithm; the other algorithm, the local-matching algorithm, is a generalized version of graph-matching-based co-scheduling algorithms.

4.4.1 A*-Cluster Algorithm

The first algorithm, A*-cluster, combines A* algorithm with clustering techniques. Through clustering, the algorithm controls the number of rescheduling stages by rescheduling only when a cluster of jobs finish. Also through clustering, the algorithm avoids the generation of sub-schedules that are similar to one another. Together the two features reduce the time complexity of the problem significantly.

One option to cluster jobs is to group them based on their single-run times. However, jobs with similar single-run times may need very different times to finish in co-run scenarios. Our solution is an online adaptive strategy.

The integration of clustering into A* search is implemented inside procedure *nextSubSchedule()* (invoked in the middle of procedure *Astar()*) as shown in Figure 4.4. The A* algorithm uses this procedure to generate a child of the current node in the search tree. Suppose the current node is not the starting node. At the first invocation of procedure *nextSubSchedule()* by this node, the procedure computes the state of the job set when the first *cluster* of the unfinished jobs complete under the current sub-schedule (**to reduce the number of scheduling stages**), and then regroups the other jobs into certain clusters. Based on the clustering results, during the generation of children nodes, each time the procedure *nextVeryNewSubSchedule* returns a sub-schedule that is substantially different from the already generated sub-schedules (**to reduce the number of nodes at a stage**.) A sub-schedule is substantially different from another one if they are not equivalent when we regard all jobs in a cluster as the same. As an example, suppose 4 jobs fall into 2 clusters as {{1 2}, {3 4}}. The sub-schedule (1 3) (2 4) is regarded as equivalent to (1 4) (2 3), but different from (1 2) (3 4) (each pair of the parentheses contains a corun group.) Finding those novel sub-schedules only needs to solve a first-order linear equation system. The unknowns are the numbers of instances of different mixing patterns of clusters; they must be non-negative. Each equation corresponds to one job cluster: on the left side is the sum of the number of the jobs falling into that cluster in each mixing pattern, on the right side is the total number of jobs belonging to that cluster. Each solution of the equation system corresponds to one novel sub-schedule. Details are skipped for lack of space.

The two reduction strategies decrease the search space significantly. The first strategy reduces the height of the search tree, while the second strategy reduces the width. They reduce the number of nodes at a stage from factorial, $\prod_{i=0}^{\frac{N}{K}-1} \binom{N-iK-1}{K-1}$, to polynomial, $O(n^{\gamma})$ ($\gamma = C + (C^K - C)/K!$), for given C and K values (C is the number of clusters.)

```
/* Jobs contains unfinished jobs; */
/* isFirstInvoke is 1 initially.  */
Procedure nextSubSchedule() {
  if (isFirstInvoke) {
   foreach job in jobs
    estimate_timeToFinish(job);
   if (this! = start) {
    C1 = getEarliestCluster(Jobs);

    /* update to the state when C1 finishes */
    Jobs = Jobs - C1;
    update_timeToFinish(Jobs);
   }
   Cs = ReCluster(Jobs);
   isFirstInvoke=0;
  }
  /* get a substantially new sub-schedule */
  nextVeryNewSubSchedule(Jobs, Cs);
}
```

Figure 4.4: Integration of clustering into A* search algorithm for approximation of optimal co-scheduling.

The clustering in our implementation is based on the time needed for a job to finish under the current sub-schedule. (For the starting node, we use single-run times.) The times are estimated by the single run time and corun degradation rates of each job. Machine learning researchers have proposed many clustering techniques, such as K-means, hierarchical clustering [35]. In our problem, the data are one dimensional and we do not know the number of clusters beforehand. So, we use a simple distance-based clustering approach. Given a sequence of data, we first sort the data in an ascending order. Then, we compute the differences between every two adjacent data items in the sorted sequence. Large differences are considered as indiction to cluster boundaries. A difference is regarded as large enough if its value is greater than $m + \delta$, where, m is the mean value of the differences in the sequence and δ is the standard deviation of the differences. An example is as follows.

times to finish : 10 15 18 32 35 51 53 56
differences : 5 3 $\underline{14}$ 3 $\underline{16}$ 2 3
job clusters : (x x x) (x x) (x x x)
mean difference = 6.5; std. = 5.9

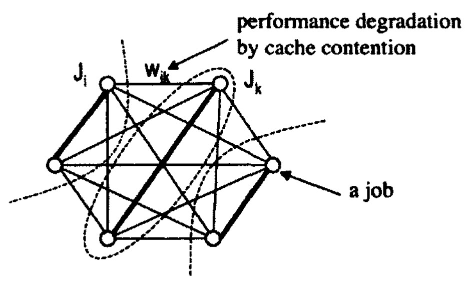
The time complexity of the clustering algorithm is $O(J)$, where J is the number of remaining jobs.

4.4.2 Local-Matching Algorithm

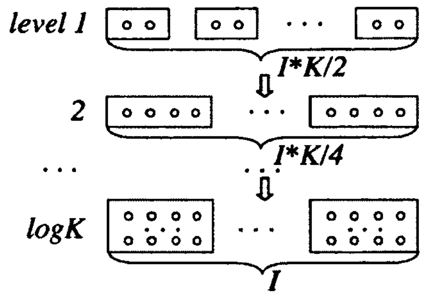
For even higher efficiency, we design a second approximation algorithm, which explores only one path from the root to the goal in Figure 4.1. At each scheduling point, it selects the schedule that minimizes the total running time of the remaining part of the unfinished jobs under the assumption that no reschedules would happen. The assumption leads to local optimum at each scheduling stage.

The key component of the algorithm is the procedure to compute the local optimum. The problem is NP-complete when a chip has more than two cores. Previous work has demonstrated the effectiveness of hierarchical perfect matching in approximating the optimal schedule when the number of jobs equals the number of cores and all jobs are of the same length [40]. In this current work, we relax those two conditions to make the algorithm more generally applicable.

Our explanation of the algorithm starts with a simple case, to co-schedule N jobs on a system



(a) Degradation graph for 6 jobs on 3 dual-cores. Each partition contains a job group sharing the same cache. Bold edges compose a perfect matching.



(b) Hierarchical perfect matching algorithm. Each box represents a virtual chip except the chips at the bottom, which are real chips. Each circle represents a core; K is the number of cores per real chip.

Figure 4.5: Illustration of perfect-matching-based co-scheduling algorithms.

with I dual-core chips and $N = I * 2$ (the number of jobs equals the number of cores.) On such a system, the local optimal co-scheduling problem can be modeled in a degradation graph, illustrated in Figure 4.5 (a). Every vertex represents a job. Unlike the previous exploration [40], in this work, the weight on each edge equals the sum of the times required for the remaining parts of the two jobs represented by the two vertices to finish if they corun on a chip. The optimal co-scheduling thus becomes a *minimum-weight perfect matching* problem. A *perfect matching* in a graph is a subset of edges that cover all vertices, but no two edges share a common vertex. A *minimum-weight perfect matching* problem is to find a perfect matching that has the minimum sum of edge weights in a graph. Clearly, a valid job schedule must be a perfect matching in the graph, and a minimum-weight perfect matching must minimize the total time for remaining jobs to finish. Hence a minimum-weight perfect matching must correspond to an optimal schedule of the remaining jobs when there is no rescheduling on those jobs.

The minimum-weight perfect matching problem can be solved by the polynomial-time *blossom* algorithm proposed by Edmonds [25]. In job co-scheduling context, the time complexity of the algorithm is $O(N^4)$ (N is the number of jobs.)

On systems with more than two cores per chip, the optimal co-scheduling becomes NP-complete [40]. Hierarchical perfect matching approximates the optimal schedule by applying minimum-weight perfect matching iteratively. Figure 4.5 (b) illustrates the basic idea. Given I K-core chips (still assuming $N = I * K$), the algorithm first treats each chip as $K/2$ dual-core virtual chips, and the on-chip cache is evenly partitioned into $K/2$ portions with every dual-core sharing one portion. By applying the minimum-weight perfect matching algorithm, we can find the assignment of jobs to those virtual chips, which partitions the job set into $N/2$ pairs. Next, by treating one pair as one scheduling unit, in the similar manner, we couple the pairs into $N/4$ pairs of pairs, or in another word, $N/4$ 4-member job groups. This process continues until the size of job group becomes K, when the schedule finishes. Each $K-$member group is one assignment in a final sub-schedule.

The hierarchical matching algorithm, although invoking the minimum-weight perfect matching algorithm $logK$ times, has the same time complexity as $O(N^4)$, thanks to that the number of vertices in the degradation graphs decreases exponentially.

In the above description, we assume that the number of remaining jobs equals the number of cores in the computing system. This assumption is needed for perfect matching algorithm to work. However, it does not hold in this work when some jobs complete early. We take a simple strategy to handle this case: treat the jobs that have finished as pseudo-jobs, which exist but consume no computing resource. Therefore, if the co-runners of a job are all pseudo-jobs, that job has no performance degradation at all. As the pseudo-jobs have to be scheduled every time, this strategy introduces some redundant computation. However, it provides an easy way to generalize the perfect matching algorithm by keeping the number of jobs always equal to the number of cores.

It is easy to see that the time complexity of the local-matching algorithm, in both dual-core and larger systems, is $O(N^5)$: The co-scheduling algorithm on a stage has complexity of $O(N^4)$, and there are N stages.

Application in SMT Co-Scheduling Although the description of the three co-scheduling algorithms assumes that the co-scheduling is on different cores in a CMP system, all those algorithms are applicable to SMT co-scheduling as well. The only change is that the co-run degradations are of jobs that run together on multiple hyperthreads that reside on the same processing unit, rather than on multiple cores on the same chip. The next section reports the experimental results on both CMP and SMT co-scheduling.

4.5 Evaluation

In this section, we first present the evaluation of the scheduling algorithms on a mix of 14 parallel and sequential programs, and then show a study of their scalability. We use two kinds of architecture. For CMP co-scheduling, the machines are equipped with quad-core Intel Xeon 5150 processors running at 2.66 GHz. Each chip has two 4MB L2 cache, each shared by two cores. Every core has a 32KB dedicated L1 data cache. For SMT co-scheduling, the machines contain Intel Xeon 5080 processors (two 2MB L2 cache per chip) clocked at 3.73 GHz with Hyper-Threading enabled (two hyperthreads per computing unit.)

The 14 test programs consist of 2 parallel programs from SPLASH-2 [73] and 12 programs randomly selected from SPEC CPU2000. As we use two threads for each of the two parallel programs, we have 16 jobs in total. We did not use the programs from the entire benchmark suites because the large problem size would make it infeasible to compare the scheduling algorithms, especially with the brute-force search algorithm. We use the two parallel programs (two threads per program) to examine the applicability of the co-scheduling algorithms for parallel (in addition to sequential) applications. Table 4.1 lists the programs with their corun degradation ranges on the Intel Xeon 5150 processors. The big ranges of degradations suggest the potential for co-scheduling.

Table 4.1: Benchmarks

Benchmark	single-run	corun degrad rate		
	time (s)	min %	max %	mean %
fmm*	5.63	0.77	11.28	3.67
ocean*	13.52	2.13	58.81	19.73
ammp	21.10	1.66	30.24	12.62
art	2.22	2.31	75.42	27.78
bzip	10.90	0.00	38.95	3.31
crafty	6.75	0.07	12.33	4.95
equake	11.05	6.42	78.00	26.46
gap	2.90	2.09	34.34	11.02
gzip	14.10	0.00	13.06	2.19
mcf	7.86	8.23	125.36	42.37
mesa	15.33	0.65	15.15	5.18
parser	3.74	1.74	37.75	13.51
twolf	5.42	0.00	15.73	5.21
vpr	4.58	3.31	42.52	18.30

* : from SPLASH-2. Others from SPEC CPU2000.

In the collection of co-run degradation rates, we follow Tuck and Tullsen's practice [79], wrapping each program to make it run 10 times consecutively, and only collecting the behavior of co-runs which are the runs overlapping with other programs.

The exponentially growing co-scheduling space makes it infeasible to determine the optimal schedule for even 16 jobs through exhaustive search. So, we first use 8 jobs to reveal the detailed comparisons among the co-scheduling algorithms, verifying the optimality of the solution provided by the A*-search-based algorithm. We then use all the 16 jobs to examine the performance and scalability of the two approximation algorithms.

4.5.1 Comparison to the Optimal

This experiment runs on Intel Xeon 5150 processors. We use the top 6 programs (8 jobs as *fmm* and *ocean* have two threads each) in Table 4.1 to compare the performance of 6 different scheduling algorithms: brute-force, A*, A*-cluster, local-matching, no-resch, and random schedulers. The *brute-force scheduler* conducts an exhaustive search of the entire schedule space to find the best schedules. The *no-resch scheduler* implements the optimal co-scheduler proposed in the previous work [40], which considers no job-length differences or possibilities of job rescheduling. The *random scheduler* schedules jobs in a random manner, corresponding to the default schedulers in most existing systems, which are oblivious to on-chip resource sharing. We obtain the random scheduling results by conducting random scheduling for 100 times and picking the one with median performance.

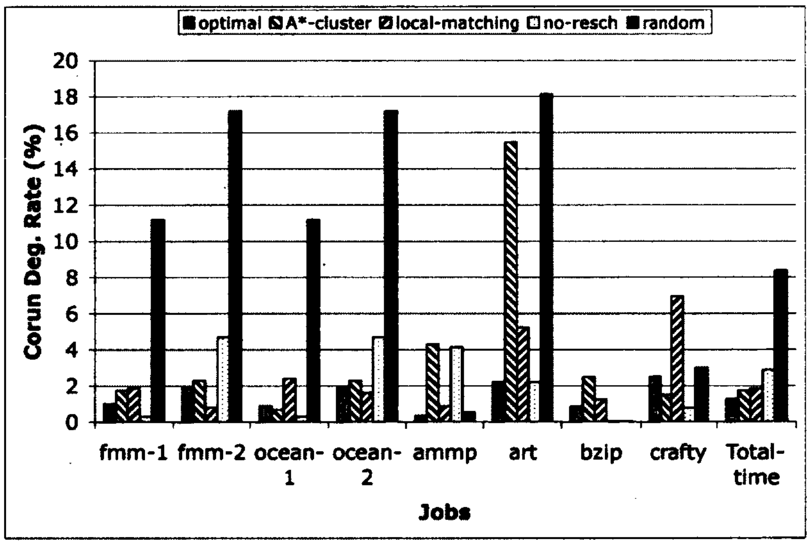


Figure 4.6: Performance degradation rates of 8 jobs co-running on quad-core Intel Xeon 5150 processors.

The results verify the optimality of the scheduling results from the A* scheduler. It produces the same schedule as the brute-force search scheduler does. Figure 4.6 shows the corun degradation rates of the 8 jobs in different schedules. The "optimal" bars represent the results of the brute-force search and the A* scheduler. The random scheduler causes 8.4% degradation to the total running time. The schedule by the no-resch scheduler is 2.9% worse than the optimal, confirming that the scheduling algorithm, although able to produce optimal schedules for the previously explored special setting, cannot guarantee the optimality in this more general scenario. The two approximation algorithms, A*-cluster and local-matching algorithms, both achieve close-to-optimal results, only 0.4–0.5% away from the optimal performance.

It is important to notice that the optimal schedule is a schedule that minimizes the total running time, but not the running time of each individual program. Therefore, it is normal to see that the optimal schedule causes larger degradation to some programs (e.g., *crafty*) than other schedulers do in Figure 4.6. By degrading the performance of some programs a little more, the optimal scheduler succeeds in decreasing the degradations of other more significant programs, and hence achieve the overall optimum.

Table 4.2 compares the schedulers in other aspects. The A* scheduler finds the optimal schedule by visiting only 0.05% of the nodes that brute-force search visits. It cuts the search time from 470 seconds to 0.3 seconds. The significant reduction demonstrates its effectiveness in space pruning. The two approximation algorithms use even less time for scheduling. The right-most two columns report the total running times and corun degradation rates of the 8 jobs under those schedules. The random scheduling results include both the median and the worst performance of 100 random schedules to show the potential risks of current sharing-oblivious scheduling.

Table 4.2: Comparison of co-scheduling algorithms on 8 jobs on quad-core Intel Xeon 5150 processors

algorithm	visited nodes	scheduling time (s)	total exec time (s)	deg. rate (%)
brute-force	16 M	470	80.3	1.3
A*	7760	0.3	80.3	1.3
A*-cluster	11	0.008	80.6	1.7
local-matching	4	0.06	80.7	1.8
no-resch	1	0.02	81.5	2.9
random	-	-	85.9–89.2	8.4–12.5

4.5.2 Results on 16 Jobs

For 16 jobs, the brute-force algorithm would take years. Our implementation of A* algorithm (in Java) is subject to memory shortage when scheduling more than 12 jobs. (A memory-bounded version [63] of the algorithm may help.) In this section, we concentrate on the evaluation of the two approximation algorithms on both multi-cores and hyperthreads.

4.5.2.1 Co-Scheduling on Multi-Cores

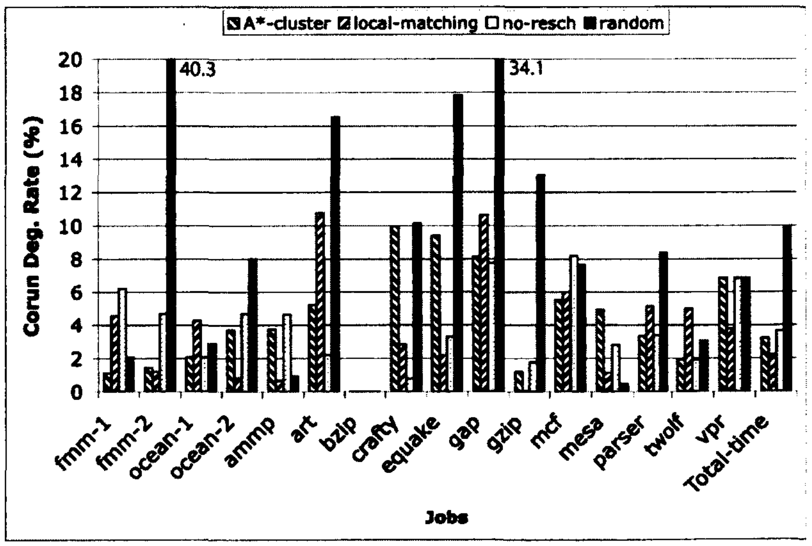


Figure 4.7: Performance degradation rates of 16 jobs co-running on quad-core Intel Xeon 5150 processors.

Figure 4.7 show the degradation rates on quad-core Intel Xeon 5150 processors; Table 4.3 reports the corresponding summary data. The random schedules cause 9.9% (up to 19.2%) degradation to the total running time. The no-resch algorithm reduces the degradation to 3.7%, while the A*-cluster and the local-matching algorithms further reduce the degradation to 3.2% and 2.2%. It is remarkable that the local-matching algorithm achieves the better result by taking less than 0.6% time of what the A*-cluster algorithm takes. This result indicates that even though the A*-cluster algorithm visits more nodes in the schedule space, the inaccuracy due to the clustering has caused considerable errors to the scheduling results.

Table 4.3: Co-scheduling 16 jobs on quad-core Intel Xeon 5150 processors

algorithm	visited nodes	sched. time (s)	total exec time (s)	deg. rate (%)
A*-cluster	721	109	149	3.2
local-matching	8	0.63	147	2.2
no-resch	1	0.03	150	3.7
random	-	-	159–172	9.9–19.2

4.5.2.2 Co-Scheduling Performance on Hyper-Threads

Figure 4.8 and Table 4.4 shows the experimental results when the 16 jobs run on the Intel Xeon 5080 processors with hyperthreads enabled. The schedule from A*-cluster reduces the median degradation rates of random schedules from 31.7% to 25.9%. The local-matching algorithm reduces the degradations to 22%, outperforming the no-resch algorithm by 2.8%.

Compared to the results in the multi-core experiments in Table 4.3, the degradation rates are clearly higher in this hyperthreading experiments because of the more extensive sharing of on-chip resource among jobs. The A*-cluster algorithm takes more time than in the multicore experiments, even though it visits fewer nodes; this is because of the difference in cluster sizes.

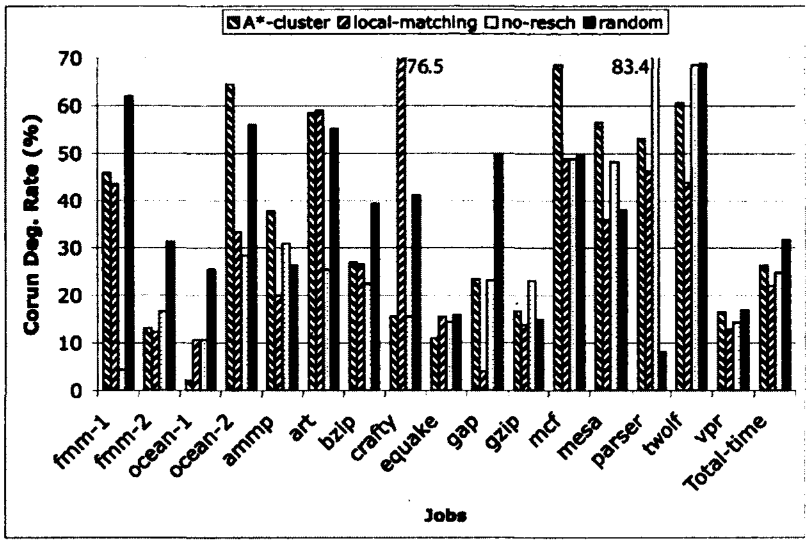


Figure 4.8: Performance degradation rates of 16 jobs co-running on the hyperthreads of Intel Xeon 5080 processors.

Table 4.4: Co-scheduling 16 jobs on hyperthreads of Intel Xeon 5080 processors

algorithm	visited nodes	sched. time (s)	total exec time (s)	deg. rate (%)
A*-cluster	315	198	325	26
local-matching	8	0.24	315	22
no-resch	1	0.03	322	25
random	-	-	340–382	32–48

4.5.2.3 Co-Scheduling Scalability

We use 32 to 128 jobs to measure the running times of the two approximation algorithms ($K = 2$). The jobs are artificial jobs with random values as their single-run times and corun degradations. Figure 4.9 depicts the running times of the algorithms on the Intel Xeon 5150 processors. The local-matching algorithm shows much better scalability than the A*-cluster algorithm does: It takes only about 10 seconds to schedule 128 jobs, whereas, the A*-cluster algorithm needs more than 2000 seconds. The reason for the difference is that the number of paths A*-cluster needs to explore in the schedule tree increases as the number of jobs increases, while the local-matching algorithm always explore a single path. The time increase of local-matching algorithm is merely due to the increased computation for obtaining the best sub-schedule at each scheduling stage.

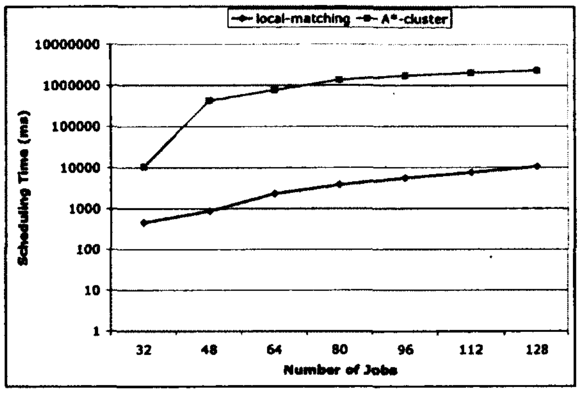


Figure 4.9: Scalability of the approximation algorithms.

Short Summary We draw the following conclusions from all the experimental results:

- The A*-search-based algorithm effectively prunes search space. When the problem size is small, it can produce optimal schedules efficiently.
- The local-matching algorithm show consistently better results than other approximation algorithms. Together with its good scalability, this algorithm is a desirable choice for large co-scheduling problems.
- The previously proposed optimal co-scheduling algorithm loses the guarantee of the optimality of its scheduling results when job lengths are different and rescheduling is allowed. Even though it still produces good results, it is consistently outperformed by the local-matching algorithm.
- The combination of clustering with A*-search shows good scheduling results, but is not as scalable as the local-matching algorithm.

4.6 Discussions

The requirement of all co-run degradations may seem to be an obstacle preventing the direct uses of the proposed algorithms in practical co-scheduling systems. However, that requirement does not impair the main goals of this work.

This work is a limit study. The primary goal is to offer feasible ways to uncover the optimal solutions in job co-scheduling, rather than to develop another heuristics-based runtime co-scheduler. Besides offering theoretical insights into co-scheduling, this work enables better evaluation of co-scheduling systems than before, offering the facility for efficiently revealing the potential of a practical co-scheduler in a general setting, which has been infeasible in the past for even small problems.

Furthermore, proactive co-scheduling (i.e., scheduling before executions) may directly benefit from the algorithms proposed in this work. There has been some work on predicting co-run performance from program single runs (e.g., [43]). Some recently proposed techniques in locality analysis has advanced the efficiency and accuracy in locality characterization [66, 75]. Those studies make it possible to predict corun performance efficiently, and thus pave the way for proactive scheduling. The algorithms proposed in this work may serve as the base for proactive scheduling algorithms. They may also provide the insights for the development of more effective online (reactive) scheduling algorithms in both operating systems and the runtime of parallel applications.

Some assumptions made in this work are not difficult to resolve. For instance, if jobs start at different times or contain phase shifts, it may suffice to add rescheduling at those changing points in the scheduling algorithms. But, some other complexities, such as process migration overhead and rescheduling at any time, may need further explorations to resolve.

4.7 Related Work

The closest work to this research is the analysis and approximation of optimal job co-scheduling from Jiang et al. [40]. The analysis and algorithms in that work is based on the assumptions of same job lengths and no rescheduling allowed. This research eliminates those assumptions and offers algorithms applicable to more general co-scheduling problems.

As Early as in 1960s, scheduling has been used for improving the usage of memory system. Denning proposed balance-set scheduling to improve virtual memory usage by grouping programs based on their working set size [22]. Recent studies employ the similar idea for CMP cache usage but use different program features, including estimated cache miss ratios [28], and hardware performance counters [26, 87]. Kim et al. study cache partitioning for fairness in CMPs [43]. DeVuyst et al. exploit unbalanced thread scheduling for energy and performance [23]. Architecture designs for alleviating cache contention have focused on cache partitioning [38, 61], cache quota management [62], cache policies [36], and heterogeneous design [45].

Cache sharing also exists in Simultaneous Multithreading (SMT) processors. Parekh et al. introduce thread-sensitive scheduling. They demonstrate that by greedily selecting jobs with the highest IPC can improve system throughput significantly [58]. Snavely et al. propose symbiotic scheduling [70, 71]. It uses sample-optimization-symbios (SOS) scheme to try different combinations of jobs and pick the best one as the optimal schedule [70]. Their later work also considers process priorities in the symbiotic scheduling [71]. Settle et al. use an L2 cache activity vector to enhance the scheduler [64]. Some other work changes process affinity according to hardware performance counters [15, 56]. The algorithms presented in this chapter may benefit SMT co-scheduling as well if co-run performance is attainable.

To the best of our knowledge, this work is the first systematic study devoted to finding the *optimal* schedules for CMP systems *with rescheduling allowed.* We are not aware of any prior work that uses A* search to reduce shared-cache contention. The combination of A* and clustering and the local-matching algorithms for co-scheduling are also novel to the best of our knowledge.

4.8 Summary

This chapter explores the problem of optimally co-scheduling jobs of different lengths. We propose an A*-based approach to accelerating the search for optimal schedules by as much as orders of magnitude. For large problems, we design and evaluate two approximation algorithms, A*-cluster and local-matching algorithms, to effectively approximate the optimal schedules with good accuracy and scalability. Experiments on both multicores and hypterthreads demonstrate that when the number of jobs is small, the A*-search-based algorithm can find the optimal co-schedule efficiently. For large problems, the local-matching algorithm is a desirable choice. It achieves better co-scheduling results than state-of-the-art algorithms, meanwhile, showing remarkable scalability.

The analysis and approximation algorithms in this chapter offer the insights and practical support for the evaluation of co-scheduling systems. The algorithms can be directly used in proactive co-scheduling when co-run performance is predictable, and may serve as the base for enhancing online co-scheduling and the runtime of parallel applications as well.

Chapter 5

Related Work

5.1 Inputs Handling in Program Optimizations.

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 estimations for dynamic behavior [5, 6]. For example, a loop is considered ten times more frequently accessed than others [21].

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 [83], PHiPAC [12], SPARSITY [39], SPIRAL [60], FFTW [29], STAPL [77]. 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 [59], Tempo from Consel's group [54], and DyC from Chambers and Eggers' group [32]. Others monitor execution through runtime profiling to optimize a program; examples include the dynamic feedback work by Diniz and Rinard [24], the continuous program optimizations by Kistler and Franz [44], the ADAPT project by Voss and Eigenmann [80], the CoCo project by Childers, Davidson and Soffa [19], the continuous program optimization (CPO) project by Wisniewski and his colleagues [84], the dynamic optimizations on LLVM [46], the run-time support for managed languages like Java and C# [8,18,47,57,72,86]. 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 2.2.2. There has been some work that uses input features, such as the computation offloading by Wang and Li [82], the adaptive algorithm selection from Li et al. [49] and Rauchwerger et al. [77]. Their explorations are mainly on a specific class of applications and use manually defined input features. Amaral et al. have investigated the influence of inputs on benchmark design [11].

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.

Techniques in branch prediction (e.g., [85]) and value prediction (e.g., [51]) 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. It also differs from dynamic invariant detectors, such as Daikon [27], whose goal is finding invariants rather than statistical relations.

Some prior studies have noticed the importance of program inputs and tried to exploit them for optimizations. There are some prior studies that 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 [82], machine learning-based compilation [48], adaptive sorting [49], and some library constructions [12,29,39,60,77,83]. Because of the required manual efforts, those explorations have been focused on some particular applications or kernels. The optimization strategy is constructed through a large number of offline profiling runs. A complementary approach to helping JIT is to enhance the compilation decisions by training over a large number of code features. An example is the method-specific dynamic compilation by Cavazos and others [16]. Their work also relies on a large number of offline training runs.

Mao and Shen [52] propose a description language, XICL, to facilitate the characterization of program inputs for program optimizations. However, that approach requires programmers to manually specify the format and potentially important features of the inputs. This process adds extra burden to the programmer and is error-prone.

In a recent work, Jiang and others [41] reveal the wide existence of statistical correlations among behaviors, and introduce the concept of seminal behaviors. This dissertation adopts seminal behaviors as an indirect but fully automatic way to characterize program inputs. By integrating it into the three-layer framework, this work demonstrates the value of seminal behaviors in supporting a new paradigm of program behavior analysis and optimizations.

5.2 Cross-run Continuous Program Optimizations

Given the large body of literatures on program optimizations, the second part of related work concentrates on the studies closest to cross-run continuous program optimizations, including memorization based optimizations, continuous program optimizations, input-based optimizations, and sampling based optimizations.

In memorization based optimizations, a memo function remembers the results corresponding to some set of specific inputs to avoid repeating the calculation [4,55]. Input-centric optimization does not just remember, but also predict, and it concentrates on statistical behavior patterns, rather than function results.

Input-centric optimization differs from previously proposed continuous program optimizations. Most of the prior techniques confine the analysis and transformations to a single program run [24, 44, 80], rather than across runs. But none of them has systematically explored the influence of program inputs or built predictive models to connect inputs with program behaviors, which are the foundations and distinctive features of our technique.

There have been some proposals on continuous compilation across runs, including the design of CoCo by Chlders and others [19] and the CPO framework by Wisniewski and others [84]. Their focuses are on the design of high-level architectures, loop transformations, or the exploitation of multiple levels of the software stack. Their design contains no systematic treatment to program inputs. Our work differs from the repository-based cross-run learning system by Arnold and his colleagues [9] in three main aspects. First, we propose an automatic way to tackle input complexity over production runs, which is not addressed in their study. Second, our technique tailors the optimization strategy for every input rather than producing a single strategy that maximizes the average performance of all past runs. Finally, our technique uses self-evaluation to selectively predict optimal strategies with confidence; their technique applies the learned strategy to new inputs with no guarding.

Mao and Shen have developed a framework for cross-input learning and optimizing programs [52].

Their work uses manually characterized input features without addressing the difficulties in automatic input characterization through production runs. The required manual efforts are extra burden that impairs the adoption of cross-input learning and optimizations. To the best of our knowledge, this dissertation is the first that enables fully automatic cross-run optimizations with input-adaptivity.

This current study draws on the observations on behavior correlations and the concept of seminal behaviors contributed by Jiang and others [41]. However, their work has only showed the existence of the correlations and seminal behaviors, but has not used them either for input characterization, or for any kind of proactive dynamic optimizations.

The term, continuous program optimizations, was also used to refer to pure runtime adaptive optimizations [8, 24, 44, 57, 80]. They typically use runtime lightweight profiling to guide dynamic optimizers. They do not use cross-run knowledge, and do not deal with input complexities explicitly.

Much work has used sampling for program optimizations (e.g. [18,37]) and debugging (e.g., [14, 42]). Cross-user data collection has been used for bug isolation [50] and compilation [81]. The data collection scheme used in this current work differs from the previous work in that it uses randomization to speedup coverage, and employs the overhead pre-inspection to guard instrumentation. The two techniques show effectiveness for both coverage maximization and overhead control; we are not aware of prior uses of these two techniques.

Chapter 6

Conclusion

In summary, the dissertation makes the following major contributions.

- It develops the first input-centric paradigm for program behavior analysis and optimizations. Some recent studies 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.

- It 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. The dissertation demonstrates that a systematic treatment to these properties may significantly improve the learning results and yield substantial enhancement to the optimization results.

- It proposes and implements the first lifelong continuous optimizer with cross-input adaptivity. It presents a technique for transparent input characterization, with which, for the first time, it shows that input characterization is feasible through a transparent training over production runs without any manual intervention or offline profiling. Given the importance of inputs on influencing program behaviors, this contribution paves the path to many kinds of program input-based performance prediction and maximization.

- Finally, it investigates the major considerations for lifelong continuous optimizations to work profitably. It introduces a continuous learning framework to connect different components into a synergy, working cooperatively to enrich the knowledge base and enhance program performance continuously and confidently.

To the best of our knowledge, this dissertation is the first systematic exploration on how to automatically exploit program inputs for proactive dynamic optimizations. Based on the explorations, we propose a new optimization paradigm, namely input-centric optimizations, synthesizing a set of techniques. This dissertation presents this paradigm in two stages. The first one, input-centric optimization, enables the whole paradigm through offline training; The second one, input-conscious cross-run optimizations, attempts to bring the whole paradigm online.

For input-centric optimization through offline training, we address the challenges in each of the three layers, including input characterization, input-behavior modeling and input-centric adaptation. We identify seminal behaviors through offline training and build the models between seminal behaviors and the program behaviors of interest. Then during runtime, the program will be automatically optimized in a proactive, dynamic fashion based on the predictive models;

For input-conscious cross-run optimizations, we proposes a set of techniques to automatically characterizing program inputs without requirements for offline profiling runs. It incrementally accumulates observations in history runs by many users and builds up the knowledge base on the relation between program inputs and runtime behaviors. With the help of this online optimization scheme, a program is able to continuously evolve (in terms of performance) through its life time, requiring no offline training process.

Fundamentally, the two optimization schemes provide a brand new solution for dynamic program behavior prediction and proactive program optimization. The techniques proposed in this dissertation together resolve the adaptivity-proactivity dilemma that has been limiting the effectiveness of existing optimization techniques. The results on both Java and C/C++ applications demonstrate the effectiveness of the new paradigm. It may open many new opportunities for a broad range of runtime optimizations and adaptations, and it's promising in advancing the current state of program optimizations into a new level.

Bibliography

[1] Gnu linear programming kit. Available at `http://www.gnu.org/software/glpk/glpk.html`.

[2] Java Grande benchmark. http://www2.epcc.ed.ac.uk/javagrande/.

[3] Spec jvm98. http://www.spec.org/jvm98/.

[4] U. ACAR, G. BLELLOCH, AND R. HARPER. Selective memorization. *SIGPLAN Notice*, 38(1):14–25, 2003.

[5] A. V. AHO, M. S. LAM, R. SETHI, AND J. D. ULLMAN. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[6] R. ALLEN AND K. KENNEDY. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[7] 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*, pages 47–65, Minneapolis, MN, October 2000.

[8] 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.

[9] M. ARNOLD, A. WELC, AND V.T. RAJAN. Improving virtual machine performance using a cross-run profile repository. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 297–311, 2005.

[10] MATTHEW ARNOLD, STEPHEN FINK, DAVID GROVE, MICHAEL HIND, AND PETER F. SWEENEY. Adaptive optimization in the jalapeo jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 47–65. ACM Press, 2000.

[11] P. BERUBE AND J. N. AMARAL. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

[12] 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.

[13] S. M. BLACKBURN ET AL. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2006.

[14] M. D. BOND, K. E. COONS, AND K. S. MCKINLEY. Pacer: Proportional detection of data races. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[15] JAMES R. BULPIN AND IAN A. PRATT. Hyper-threading aware process scheduling heuristics. In *2005 USENIX Annual Technical Conference*, pages 103–106, 2005.

[16] J. CAVAZOS AND M. O'BOYLE. Method-specific dynamic compilation using logistic regression. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[17] D. CHANDRA, F. GUO, S. KIM, AND Y. SOLIHIN. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] P. DENNING. Thrashing: Its causes and prevention. In *Proceedings of the AFIPS 1968 Fall Joint Computer Conference*, volume 33, pages 915–922, 1968.

[23] M. DEVUYST, R. KUMAR, AND D. M. TULLSEN. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[24] 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.

[25] J. EDMONDS. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[26] ALI EL-MOURSY, R. GARG, D. H. ALBONESI, AND S. DWARKADAS. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of the International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[27] M. ERNST, J. PERKINS, P. GUO, S. MCCAMANT, M. TSCHANTZ C. PACHECO, AND C. XIAO. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[28] A. FEDOROVA, M. SELTZER, AND M. D. SMITH. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, 2007.

[29] M. FRIGO AND S. G. JOHNSON. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[30] A. GAL, B. EICH, M. SHAVER, D. ANDERSON, ET AL. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference On Programming Language Design and Implementation*, 2009.

[31] A. GEORGES, D. BUYTAERT, AND L. EECKHOUT. Statistically rigorous Java performance evaluation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2007.

[32] 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.

[33] 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.

[34] 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.

[35] T. HASTIE, R. TIBSHIRANI, AND J. FRIEDMAN. *The elements of statistical learning*. Springer, 2001.

[36] L. R. HSU, S. K. REINHARDT, R. LYER, AND S. MAKINENI. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 13–22, 2006.

[37] X. HUANG, S. M. BLACKBURN, K. S. MCKINLEY, J. E. MOSS, Z. WANG, AND P. CHENG. The garbage collection advantage: improving program locality. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[38] J. HUH, C. KIM, H. SHAFI, L. ZHANG, D. BURGER, AND S.W. KECKLER. A nuca substrate for flexible cmp cache sharing. In *Proceedings of International Conference on Supercomputing*, pages 31–40, 2005.

[39] EUN-JIN IM, KATHERINE YELICK, AND RICHARD VUDUC. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[40] 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.

[41] 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.

[42] G. JIN, A. V. THAKUR, B. LIBLIT, AND S. LU. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2010.

[43] S. KIM, D. CHANDRA, AND Y. SOLIHIN. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2004.

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

[45] R. KUMAR, D. M. TULLSEN, AND N. P. JOUPPI. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2006.

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

[47] 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.

[48] 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.

[49] 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.

[50] B. LIBLIT, A. AIKEN, A. X. ZHENG, AND M. I. JORDAN. Bug isolation via remote program sampling. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] D. MICHIE. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[56] NAKIJIMA AND PALLIPADI. Enhancements for hyperthreading technology in the operating system — seeking the optimal scheduling. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[57] 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.

[58] S. PAREKH, S. EGGERS, H. LEVY, AND J. LO. Thread-sensitive scheduling for smt processors. Technical Report 2000-04-02, University of Washington, June 2000.

[59] M. POLETTTO, W.C. 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.

[60] M. PUSCHEL, J.M.F. MOURA, J.R. JOHNSON, D. PADUA, M.M. VELOSO, B.W. SINGER, JIANXIN XIONG, F. FRANCHETTI, A. GACIC, Y. VORONENKO, K. CHEN, R.W. JOHNSON, AND N. RIZZOLO. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[61] M. K. QURESHI AND Y. N. PATT. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*, pages 423–432, 2006.

[62] N. RAFIQUE, W. LIM, AND M. THOTTETHODI. Architectural support for operating system-driven CMP cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 2–12, 2006.

[63] S. RUSSELL AND P. NORVIG. *Artificial Intelligence*. Prentice Hall, 2002.

[64] A. SETTLE, J. L. KIHM, A. JANISZEWSKI, AND D. A. CONNORS. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 63–73, 2004.

[65] 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.

[66] X. SHEN, J. SHAW, B. MEEKER, AND C. DING. Locality approximation using time. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, pages 55–62, 2007.

[67] 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.

[68] 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.

[69] 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.

[70] A. SNAVELY AND D.M. 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.

[71] A. SNAVELY, D.M. TULLSEN, AND G. VOELKER. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 66–76, 2002.

[72] 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.

[73] SPLASH. Stanford parallel applications for shared memory (SPLASH) benchmark. Available at `http://www-flash.stanford.edu/SPLASH/`.

[74] A. SRIVASTAVA AND J. THIAGARAJAN. Effectively prioritizing tests in development environment. In *Proceedings of International Symposium on Software Testing and Analysis*, 2002.

[75] G.E. SUH, S. DEVADAS, AND L. RUDOLPH. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, 2001.

[76] G.E. SUH, S. DEVADAS, AND L. RUDOLPH. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2002.

[77] 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.

[78] K. TIAN, Y. JIANG, E. ZHANG, AND X. SHEN. An input-centric paradigm for program dynamic optimizations. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[79] N. TUCK AND D. M. TULLSEN. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 26–35, 2003.

[80] 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.

[81] B. WAGNER. *Collaborative compilation.* PhD thesis, Computer Science Dept., MIT, 2006.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] X. ZHANG, S. DWARKADAS, G. FOLKMANIS, AND K. SHEN. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[88] Y. ZHONG, X. SHEN, AND C. DING. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.

[89] X. ZHUANG, S. KIM, M. SERRANO, AND J. CHOI. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2008.

VITA

Kai Tian

Kai Tian is a PhD student in computer science department at The College of William and Mary, USA. His research area focuses on program dynamic optimization, input-centric program optimizations, program analysis and profiling for performance improvement, parallel computing, data mining and information retrieval. The ultimate goal of his research is to make softwares run faster and intelligently, so that they can benefit software developers and customers in more comprehensive aspects. Beyond that, he loves programming and has been trying to create some fantastic software products that can benefit a lot of people.

Before attending graduate school in the US, he spent four years in School of Software at Shanghai Jiao Tong University, where he received his bachelor degree of Computer Science with a specialization in Software Engineering. He was born in Feicheng City, Shandong Province, China, where he attended elementary school, middle school and high school. He is always trying to be the best of himself. His favourite quote is "I can accept failure, but I can't accept not trying.", by Michael Jordan. He plays a lot of sports during his spare time, including basketball, tennis, table tennis and badminton. Sports can keep him energetic to improve himself every day.