# A Step Towards Transparent Integration of Input-Consciousness into Dynamic Program Optimizations

Kai Tian Eddy Z. Zhang Xipeng Shen
Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{ktian,eddy,xshen}@cs.wm.edu

## Abstract

Dynamic program optimizations are critical for the efficiency of applications in managed programming languages and scripting languages. Recent studies have shown that exploitation of program inputs may enhance the effectiveness of dynamic optimizations significantly. However, current solutions for enabling the exploitation require either programmers' annotations or intensive offline profiling, impairing the practical adoption of the techniques.

This current work examines the basic feasibility of transparent integration of input-consciousness into dynamic program optimizations, particularly in managed execution environments. It uses transparent learning across *production runs* as the basic vehicle, and investigates the implications of cross-run learning on each main component of input-conscious dynamic optimizations. It proposes several techniques to address some key challenges for the transparent integration, including randomized inspection-instrumentation for cross-user data collection, a sparsity-tolerant algorithm for input characterization, and selective prediction for efficiency protection. These techniques make it possible to automatically recognize the relations between the inputs to a program and the appropriate ways to optimize it. The whole process happens transparently across production runs; no need for offline profiling or programmer intervention. Experiments on a number of Java programs demonstrate the effectiveness of the techniques in enabling input-consciousness for dynamic optimizations, revealing the feasibility and potential benefits of the new optimization paradigm in some basic settings.

*OOPSLA'11,* October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10...$10.00

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

***General Terms*** Languages, Performance

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

## 1. Introduction

Dynamic program optimizations play a central role for enhancing the performance of applications in managed programming languages (e.g., Java and C#) [3], as well as scripting languages (e.g., Javascript) [13]. 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 [19, 23, 32]. 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. Recent studies [32] 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, Tian and others 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 [32], Li and others have observed 44% performance potential for sorting library construction when certain attributes of input data sets are used [23].

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. Some prior studies have proposed the use of annotations [25] or offline profiling-based solutions [19, 32]. 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 current study 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 input-conscious dynamic optimizations[1].

## 2. Overview of This Work

As prior work [32] describes, input-conscious dynamic optimizations 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 previous work, most parts of input-conscious dynamic 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 [32].

The solution investigated in this work 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.

[1] We use "input-conscious" rather than the previous term, "input-centric" [32], for its intuitiveness.

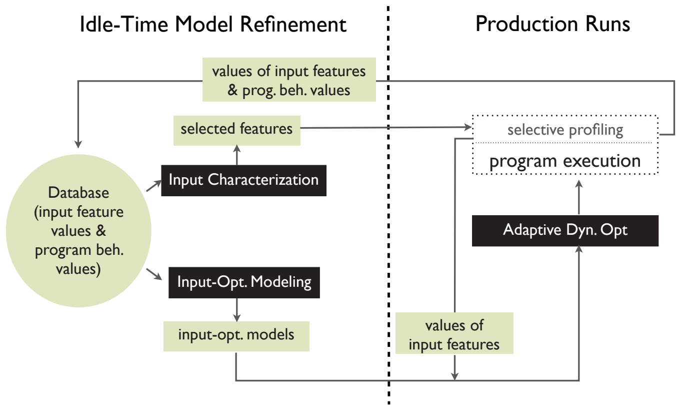

**Figure 1.**

As Figure 1 illustrates, in this paradigm, all three components of input-conscious 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 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 [19]. 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 work, we propose randomized inspection-instrumentation to solve the problem. The solution is based on a cross-user sampling scheme [24]—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 4.2 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 4.3 presents the algorithm.

The challenges to the other components of input-conscious 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 5 and 6 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 input-conscious optimizations, and ensembles them into a concerted continuous optimization system to proceed transparently and profitably.

Experiments based on a JVM, namely Jikes RVM [3], demonstrate that the proposed techniques are effective in addressing some major obstacles for transparent input-conscious dynamic optimizations. On 18 Java benchmarks, we observe 10–26% average speedup compared to their executions on the default Jikes RVM (Section 8), outperforming the previous input-oblivious approach substantially.

We stress that creating a complete transparent input-conscious 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 7) 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 paper 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 work makes four main contributions.

- As the first exploration towards transparent input-conscious 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.

$$\text{opt-level} = \underset{i \in \{0,1,2\}}{\text{argmax}} \; (\text{Benefit}(i) - \text{Cost}(i)).$$

$$\text{Benefit}(i) = T_{\text{future}} * (1 - 1/\text{speedup}(i)).$$

$$\text{Cost}(i) = \text{compileSpeed}(i) * \text{method\_size}.$$

**Figure 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 $i$.)

- It proposes a continuous learning framework that enables incremental evolvement of input-conscious dynamic optimizations with risks tightly controlled.

We organize the rest of this paper around the challenges each of the three key components of input-conscious 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. Platform: Jikes RVM

We use Jikes RVM [3], 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.

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 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 [3].

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.

## 4. 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.

### 4.1 Review on Program Seminal Behaviors

The concept of program seminal behaviors is proposed by Jiang and others [19]. 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 input-conscious 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 [19], 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.

### 4.2 Randomized Inspection-Instrumentation for Data Collection

There have been much work on lightweight runtime profiling [7, 9, 17, 20]. In this work, we use cross-user sampling [24] 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 [24] 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 [19], 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 5.2).

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 [19].

### 4.2.1 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). 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 8).

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 $i$th 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 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.

```
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.** The online algorithm for guarded adaptive instrumentation.

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

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

$$(1/size(L) + totalCost) < H. \qquad (1)$$

From Proposition 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 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") [16].

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

| | 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 4.** An illustration of the difficulty for seminal behavior recognition caused by data sparsity.

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.

## 4.3 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 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.

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 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 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 loops in order of loop ID. For each pair, it feeds the data

```
//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 $\cup$ 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 5.** Algorithm for seminal behavior recognition.

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 5.2). 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 [19]. Details are elided.

The technique designed in this work for seminal behavior identification shares certain commonality with the prior technique [19] 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 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 8). 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.

## 5. 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 input-conscious 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 input-conscious dynamic optimizations to work effectively.

### 5.1 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 $f()$.

The previous work [32] 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 [26, 29]. 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 paper.

***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 [16] 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.

## 5.2 Adaptive Dynamic Optimizations

The third component of input-conscious 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).

## 6. 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 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.

## 7. 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 paper. 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. [38]) 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. [30]) 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 input-conscious 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.

## 8. Evaluation

Our evaluation focuses on the effectiveness of the techniques for overhead control, and the feasibility and potential of the transparent input-conscious 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?

## 8.1 Methodology

***Platform*** Our implementation is on Jikes RVM (v. 3.1), which has been briefly described in Section 3. 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 a previous study on offline input characterization [32]. 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 all the benchmarks in the previous suite, we add all the other programs from the Dacapo (2006) benchmark suite, except Chart, for comprehensiveness of the test. (We have not figured out how to get new inputs for Chart.) 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, consulting the authors of the Dacapo suite (Our special thanks to Blackburn!), 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, Antlr, Bloat) are modified to reactivate some of their command-line options that were disabled by the benchmark suite interface.

Table 1 lists all the benchmarks. These benchmarks cover a variety of domains, from utility tools to compiler tools to computational applications. The inputs exhibit large variations, reflected by the large differences in the corresponding running times shown in the 5th and 6th columns of the table.

***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 work, 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.

## 8.2 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 work, 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 [14] 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 6 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

**Table 1.** Benchmarks and Their Properties

| Program | Description | Code lines | # Inputs | Running time (s) Min | Running time (s) Max | #Runs in init stage | Sampled loops per run (%) |
|---|---|---|---|---|---|---|---|
| Compress[j] | compression tool | 927 | 20 | 0.94 | 9.33 | 618 | 6.7 |
| Db[j] | database tool | 1028 | 54 | 0.59 | 98.16 | 80 | 18 |
| Mtrt[j] | multithreaded ray tracer tool | 3842 | 100 | 0.26 | 6.37 | 1161 | 4.2 |
| Euler[g] | computational fluid dynamics | 1179 | 20 | 0.93 | 7.79 | 55 | 17.4 |
| MolDyn[g] | molecular dynamics simulation | 583 | 20 | 0.11 | 63.05 | 38 | 21 |
| MonteCarlo[g] | Monte Carlo simulation | 3073 | 21 | 9.07 | 15.81 | 204 | 11.1 |
| Search[g] | Alpha-Beta pruned search | 712 | 9 | 2.74 | 210.36 | 106 | 20 |
| RayTracer[g] | 3D ray tracer | 1224 | 21 | 3.10 | 236.57 | 83 | 16.7 |
| Antlr[d] | parser generator | 32263 | 175 | 0.15 | 0.19 | 1270 | 4.8 |
| Bloat[d] | bytecode-level optimization | 73563 | 100 | 0.08 | 41.46 | 1815 | 3.9 |
| Eclipse [d] | multi-language IDE | 1903219 | 80 | 0.572 | 86.648 | 1856 | 1.5 |
| Fop [d] | print formatter | 88846 | 70 | 0.385 | 2.039 | 943 | 3.4 |
| Hsqldb [d] | SQL relational database engine | 151915 | 75 | 0.455 | 8.888 | 1146 | 2.9 |
| Jython [d] | python interpreter in Java | 91982 | 60 | 0.594 | 34.02 | 1258 | 1.9 |
| Luindex [d] | text indexing tool | 8570 | 50 | 0.363 | 7.299 | 645 | 3.2 |
| Lusearch [d] | text search tool | 12709 | 63 | 0.482 | 1.443 | 1630 | 1.8 |
| Pmd [d] | Java source code analyzer | 49331 | 53 | 0.323 | 4.475 | 1051 | 4.3 |
| Xalan [d] | transform XML documents | 243516 | 60 | 0.229 | 5.723 | 924 | 3.1 |

j: jvm98 [2]; d: dacapo [6]; g: grande [1]

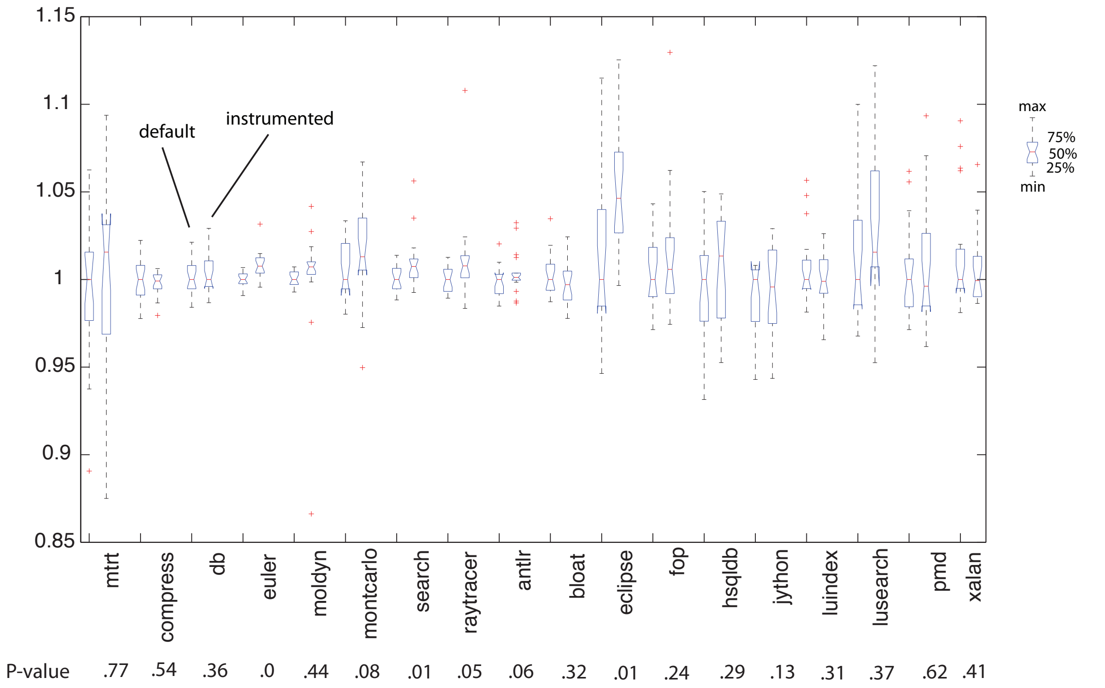

**Figure 6.** Distributions of the running times of the default and instrumented runs. The P-values at the bottom indicate whether the two differ significantly ($< 0.05$) or not ($> 0.05$).

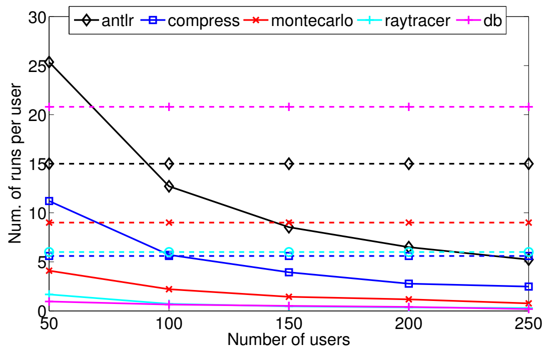

**Figure 7.** The number of runs per user in the initial stage. The solid-line curves are the results when randomization is used; the broken-line curves are when randomization is not used.

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 [16]. As shown in the bottom of Figure 6, 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 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 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 7 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.

### 8.3 Prediction Accuracy and Performance Enhancement

As Section 5.1 describes, the input-conscious 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.

#### 8.3.1 Target for Enhancement

In this example use, the objective is to enhance the compilation strategy in Jikes RVM. In Section 3, 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 [15].

As many dynamic optimization systems use the similar strategy, this weakness is shared by almost all of them. Several studies [4, 15, 32] have reported the importance of this weakness. For instance, Arnold and others [4] 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 [32] or are subject to input-obliviousness [4].

#### 8.3.2 A Solution from the New Paradigm

In this experiment, we try to apply the transparent input-conscious 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 [4, 32] by removing the needs for offline profiling and enabling input-consciousness.

#### 8.3.3 Results

Figure 8 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 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 5.1. The confidence value is computed using cross-validation on the existing data base, as described in Section 5.2. 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 5.2 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 input-conscious optimizations is quite significant as Figure 8 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 input-conscious 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 [4]. 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 $k$th 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 input-conscious. 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 [32]. 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 a recent work [32] 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 9 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 10–26% 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 input-conscious continuous optimization paradigm. (In several cases, our approach shows even better performance than the offline one. It is due to the imperfect design of the compiler, just another indication of the complexity and sub-optimality of the current compiler construction.)

## 9. Related Work

Given the large body of literatures on program optimizations, this section concentrates on the studies closest to cross-run program optimizations, input-based optimizations, and sampling.

There have been some proposals on continuous compilation across runs, including the design of CoCo by Childers and others [10] and the CPO framework by Wisniewski and others [37]. 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

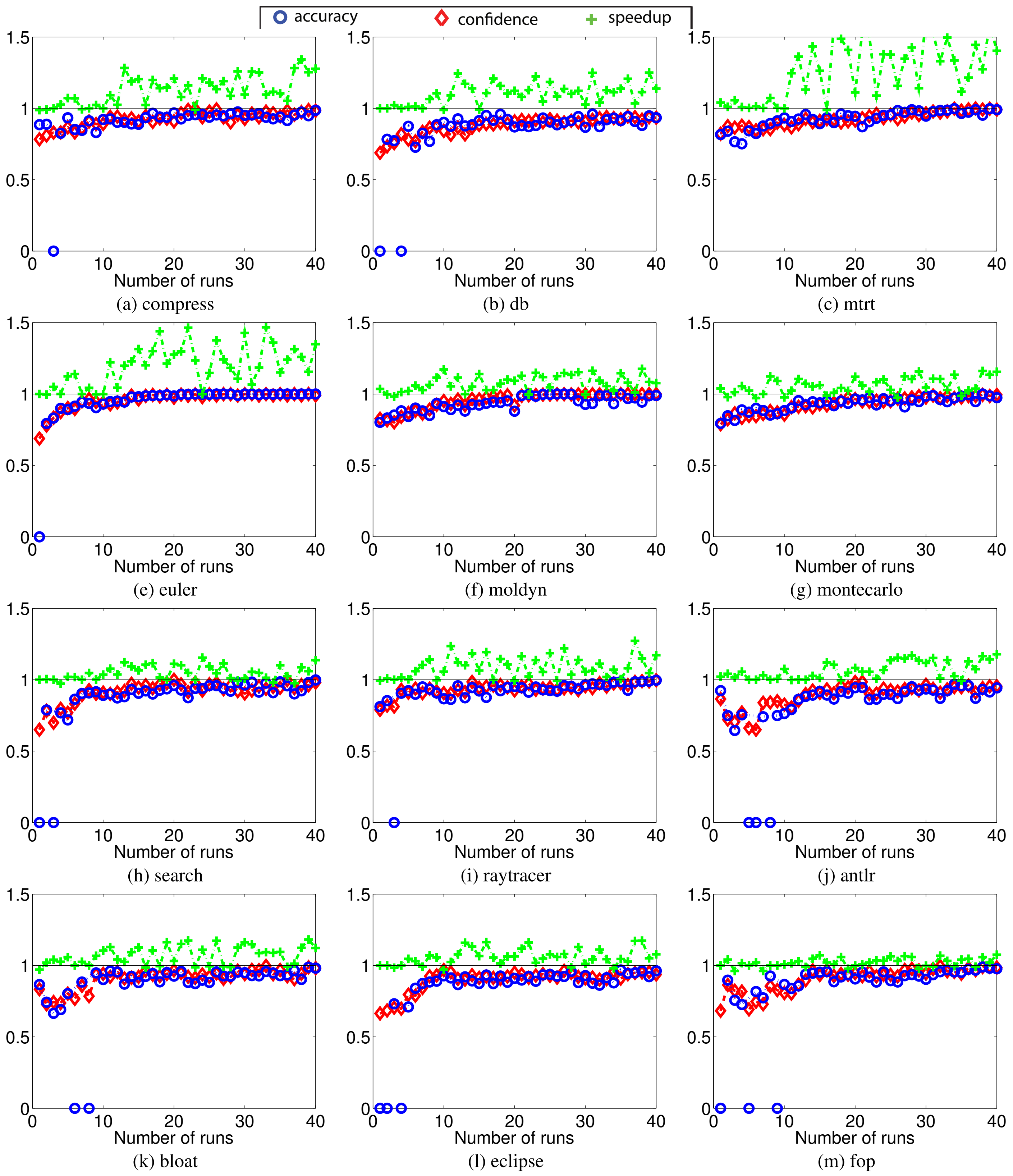

**Figure 8.** The cross-run changes of the prediction confidence and accuracy of the input-opt models, along with the corresponding performance enhancement over the executions in the default Jikes RVM.

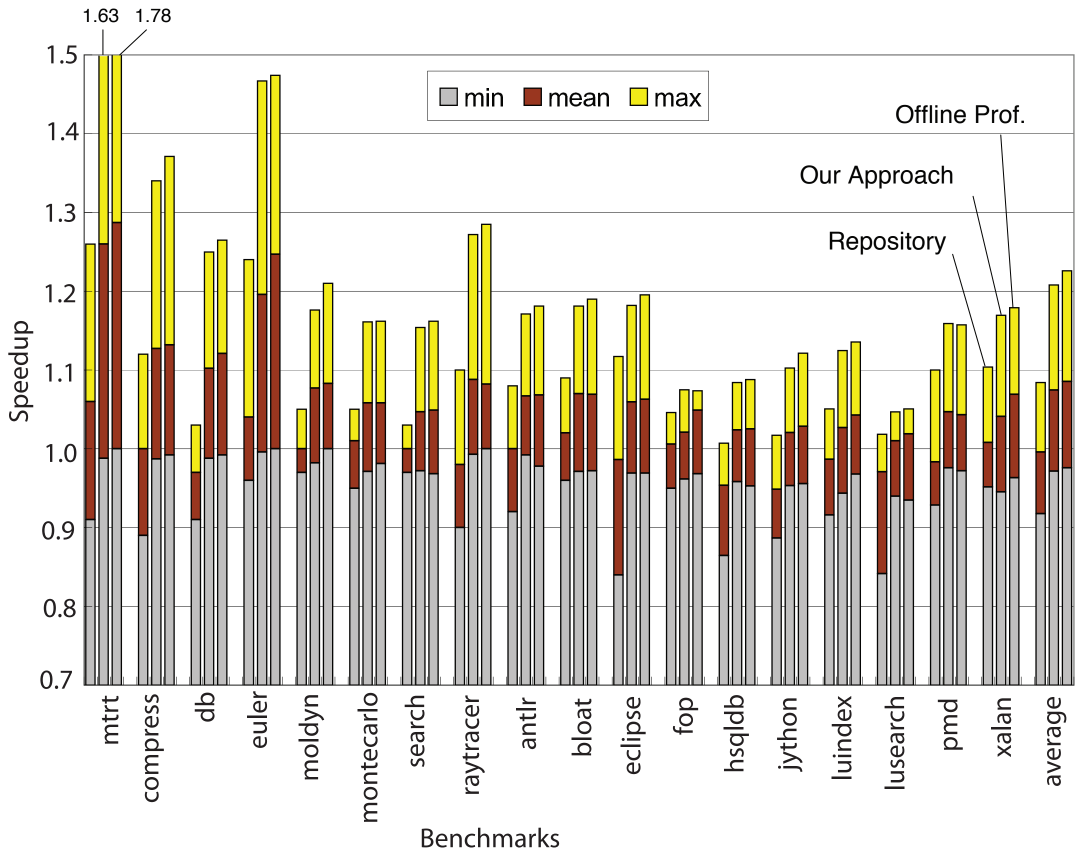

**Figure 9.** The overall speedups from repository-based approach, input-conscious continuous optimizations, and the upper-bounds obtained through offline-profiling–based experiments.

no systematic treatment to program inputs. Our work differs from the repository-based cross-run learning system by Arnold and his colleagues [4] 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 [25]. 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 current work is the first that enables fully automatic cross-run optimizations with input-adaptivity.

Some prior studies have noticed the importance of program inputs and tried to exploit them for optimizations. Tian and others propose an input-centric framework [32]. Their technique is heavily based on offline profiling for both input characterization and the recognition of the relations between inputs and optimizations. There are some other 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 [35], machine learning-based compilation [22], adaptive sorting [23], and some library constructions [5, 12, 18, 28, 31, 36]. 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 [8]. Their work also relies on a large number of offline training runs.

The term, continuous program optimizations, was also used to refer to pure runtime adaptive optimizations [3, 11, 21, 27, 33]. 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. [9, 17]) and debugging (e.g., [7, 20]). Cross-user data collection has been used for bug isolation [24] and compilation [34]. 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.

## 10. Conclusion

In this paper, 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 input-conscious 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.

## Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. We are especially grateful for Steve Blackburn and John Zigman's help on the collection and creation of extra inputs of the Dacapo benchmark suite. We thank Michael Bond for answering our questions about Jikes RVM, and Matthew Arnold for commenting on the repository-based approach. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and 0954015. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

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

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of International Symposium on Software Testing and Analysis*, 2002.

[31] 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.

[32] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[33] 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.

[34] B. Wagner. *Collaborative compilation*. PhD thesis, Computer Science Dept., MIT, 2006.

[35] 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.

[36] 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.

[37] 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.

[38] 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.