Cross-Input Learning and Discriminative Prediction in Evolvable Virtual Machines

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

***Abstract*—Modern languages like Java and C# rely on dynamic optimizations in virtual machines for better performance. Current dynamic optimizations are reactive. Their performance is constrained by the dependence on runtime sampling and the partial knowledge of the execution. This work tackles the problems by developing a set of techniques that make a virtual machine evolve across production runs. The virtual machine incrementally learns the relation between program inputs and optimization strategies so that it proactively predicts the optimizations suitable for a new run. The prediction is discriminative, guarded by confidence measurement through dynamic self-evaluation. We employ an enriched extensible specification language to resolve the complexities in program inputs. These techniques, implemented in Jikes RVM, produce significant performance improvement on a set of Java applications.**

***Index Terms*—Cross-Input Learning, Java Virtual Machine, Evolvable Computing, Adaptive Optimization, Input-Centric Optimization, Discriminative Prediction**

I. Introduction

The recent decade has seen continuously growing interests in dynamic optimizations, reflected especially in the widely used Java and C# virtual machines. For programs written in those languages, dynamic optimizations are essential to their performance.

Compared to traditional static compilation, dynamic optimizations are advantageous in adapting to runtime program behaviors. However, the optimizers in most current virtual machines are reactive: They make decisions based on the just observed behaviors. The reactivity limits the optimizations on two aspects. First, it causes delay in optimizations because many runtime samplings are required to recognize the behavior of the stable stage of an execution. Second, because the observed behaviors are of the past interval rather than the entire execution, they may mislead the optimizer into an inferior decision. Some recent work has revealed significant potential benefits from the removal of those limitations [2], [6].

In this work, we attempt to address these limitations by making dynamic optimizations *proactive*. Our approach departs from previous techniques in being input-centric with program input characterization and cross-input behavior prediction at the core. The underlying mechanism is to incrementally build some predictive models during production runs of an application to capture the relation between the inputs (their values and any other attributes) and the runtime behavior of the application. When the application is launched with an arbitrary (legal) input, the virtual machine will use those models to proactively predict the suitable optimization decisions for the application and optimize the application accordingly.

More specifically, this approach consists of three techniques. The first is *input feature extraction*. One of the major challenges in bridging program inputs with runtime behaviors is the complexity in program inputs. An input may include many options and various structures, such as trees, graphs, even programs. The complexity hides critical features and complicates the recognition of the correlation between program inputs and optimization strategies. In this work, we use an enriched **e**xtensible **i**nput **c**haracterization **l**anguage (XICL) to resolve input complexity and implement an efficient XICL translator in Jikes RVM. The enriched XICL allows a programmer to specify the input format and potentially important input features for programs with or without interactive features, and an XICL translator automatically converts a program input into a well-formed feature vector. The implementation of the translator enables the capitalization of both the computation in the original application and the knowledge of programmers, making feature extraction efficient.

The second technique is *incremental input-behavior modeling*, which gradually reveals the statistical relation between program inputs and runtime behaviors by learning across the *production* runs of an application. The learning is transparent to users, requiring no offline profiling. Moreover, it attains typical inputs for free: The inputs used in production runs embody the typical inputs used by the specific user. In this work, we choose incremental Classification Trees [7] as the modeling technique to recognize the input-behavior correlations and refine input-feature vectors. The learning adds negligible runtime overhead by separating the task into two stages—online lightweight data collection and offline model construction.

The third technique is *discriminative prediction.* For every predictive input-behavior model, the technique uses cross-validation to compute a confidence level that reflects the quality of the model. The dynamic optimizer uses a model only when the confidence level of the model is high enough. Otherwise, the optimizer falls back to the default (reactive) optimization strategy. This discriminative prediction is especially important for cross-run incremental learning because the quality of the predictive models is usually inferior initially and

improves across runs gradually. This discriminative scheme prevents inferior predictions from misleading the optimizer and causing performance degradations.

With these three techniques integrated, a virtual machine can evolve across runs: As it learns more knowledge on an application through runs, the virtual machine is able to predict the application's behavior better and better. In comparison to the learning in existing virtual machines and profiling-directed-feedback compilations, evolvable virtual machines are unique in that their learning is across production runs, transparent to users, and protected by a self-evaluation mechanism. The evolvement opens many opportunities for proactive dynamic optimizations.

There has been some research on alleviating the limitations of the reactiveness in dynamic optimizations, such as cross-run repository-based optimizations [2], and phase-based adaptive (re)compilation [6].

This work is distinct on three aspects. First, it tailors the optimization strategy to every input of an application, rather than selects the strategy that maximizes the *average* performance of the history executions [2]. Second, it uses a self-evaluation scheme to enable discriminative rathe than unconditional prediction [2], [6] on optimization parameters or aggressiveness, thus preventing most wrong predictions from misleading program optimizations. Finally, it selects the optimization strategy based on the prediction of the behavior of the entire execution rather than phase-level observations [6].

We evaluate the techniques on the selection of the optimization levels for Java methods in the Jikes Research Virtual Machine (RVM). Experiments on 11 programs show that the evolvable Java virtual machine can gradually learn the relation between program inputs and the appropriate optimization level of each Java method. On average, it can predict the correct levels with 87% accuracy. The prediction helps the runtime optimizer improve the performance of the applications by 7-21% on average, outperforming previous repository-based RVM [2] by more than 10%.

In the rest of this paper, Section II outlines the learning framework, Section III focuses on the techniques for handling program inputs, Section IV describes the scheme of incremental learning with discriminative prediction, Section V reports experimental results, followed by some discussions, related work, and conclusions.

II. Overview of the Evolvable Virtual Machine

Figure 1 shows the 3 major components related to the evolvement. When a program is launched on the virtual machine, the *feature extractor* resolves the complexity in the program's input, and converts the input into a well-formed feature vector. From the vector, the *strategy predictor*, if confident enough, uses current predictive models to produce an optimization strategy that likely suits the new run. The virtual machine then executes the program. During the execution, it optimizes the program using the predicted optimization strategy if there is one. Otherwise, the virtual machine falls back to its default optimization scheme. In both cases, the virtual machine monitors the runtime behavior of the program through the default sampling schemes in the virtual machine. After the execution, with all the samples obtained, the virtual machine generates a (posterior) optimization strategy—which is treated as an ideal strategy for the execution—and sends it to the *model builder* to enhance the models. The evolvement relies on an XICL specification for input characterization. (Without it, the system would fall back to the default adaptive optimizer and behave the same as the default virtual machine.)

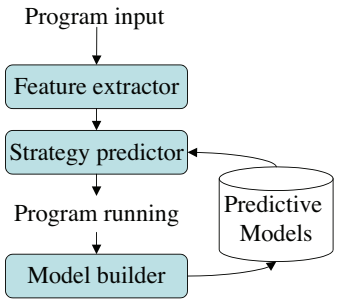


Fig. 1. Overview of the major components for evolvement.

The following two sections describe the three major components respectively. Section III presents how the feature extractor addresses the complexity in program inputs and derives the important input-features. Section IV describes how the strategy predictor predicts with confidence, and how the model builder incrementally learns the relation between program inputs and optimization strategies.

III. Input Characterization

The first challenge in tailoring optimizations to program inputs is to recognize and extract critical features from program raw inputs. It is difficult if not impossible to convert inputs to feature vectors in a fully automatic manner, mainly because of the complexity of program inputs in both format and content.

The complexities lie in three aspects. First, in many cases it is the hidden features (e.g. the size of an input file) rather than the explicit values (e.g. the name of an input file) of an input component that determine a program's behavior. Unfortunately, without domain-specific knowledge, it is often impossible to automatically recognize the format and semantic—a graph or a tree or a bag of numbers or something else—of an input component, let alone its important features. Second, the separation between categorical and quantitative features is important for behavior modeling; but automatic separation is not always possible. Finally, input components may overshadow each other, and may have default values that are difficult to recognize by automatically analyzing program code due to aliases and other difficulties in inter-procedure analysis.

On the other hand, it is apparently impractical to ask users of an application to manually convert every input to feature vectors. In this work, we propose a tradeoff solution. We develop a scheme that automates most of the process, and meanwhile, allows programmers to incorporate their knowledge without

many efforts. It requires no involvement from the program users.

A. Extensible Input Characterization Language (XICL)

XICL is a mini-language for programmers to formally describe the format and the potentially important features of the possible inputs to an application. With an XICL specification, the XICL translator will automatically determine the role of each component in an arbitrary (legal) input, and convert the input to a feature vector.

1) Primary Constructs in XICL: A component in a program input is categorized as either an option or an operand, and XICL has a construct for each[1]. Consider the example shown in Figure 2 (a). It is a program to find the shortest routes in a given graph. It allows three options: "-n" determines the number of shortest paths to find; "-e" and "–echo" determine whether to print status messages. The application admits a number of operands, which each is a file containing a graph for route finding. Figure 2 (b) contains an XICL specification for this application.

The option construct specifies the option's name, type, potentially important features ("attr" stands for attributes), default value, and whether it requires an argument. XICL contains a set of predefined types and features; "VAL", for example, stands for the value of the option. The operand construct specifies the positions of operands in the command line ("$" stands for the end of the command line), its type and potentially important features. The features starting with "m" are the features defined by the programmer; the two such features in this example, "mNodes" and "mEdges", are respectively the numbers of nodes and edges in a graph. The allowance of programmer-defined features reflects the extensibility of XICL, elaborated in the next section.

Given the specification, the XICL translator will be able to convert an arbitrary (legal) input into a vector containing the potentially important features. Suppose that we have an invocation of the route program: "route -n 3 graph1", where, graph1 contains 100 nodes and 1000 edges. The output of the XICL translator is a feature vector as (3, 0, 100, 1000) (the second element is the default value of the "-e" option.)

2) Extensibility: XICL is extensible by allowing programmer-defined features. There are two ways for a programmer to define a feature. First, the programmer can extend the XICL translator by adding some methods that extract certain features from an input. The name of those methods can then be used as the values of "attr" in the constructs. During production runs of the application, the XICL translator will automatically recognize such features and invoke the corresponding methods.

The second way is to use runtime values of the original application. The rationale is that the initialization stage of a program often exposes many useful input features. For instance, when the route program starts, it reads and parses the input graph file, during which process, the number of nodes and edges are assigned to two member variables of a class named graph. Passing the values of those variables to XICL translator as features will save some computation in feature extraction.

[1]The current design of XICL is for programs whose interfaces conform POSIX conventions.

B. XICL Translator

This section describes the design of XICL translator, the implementation of the extensibility of XICL, and its integration in Jikes RVM.

1) Basic Design of XICL Translator: At the core of the implementation is a class, named XICLTranslator. Figure 3 shows the critical components of the class. The member, fVector, is a vector recording the translation result (i.e., extracted input features.) A hash table, xfMethodsMap, maps the names of feature-extraction classes—either predefined or programmer-defined—to the corresponding class instances. Each of these classes is an implementation of the interface XFMethod (on the bottom right of Figure 3.) Their respective definitions of the xFeature() method compute the features of the corresponding input components.

The workflow of the XICL translator is as follows[2]. As an application is launched, the virtual machine creates an instance of the XICL translator, and then invokes the buildFVector method with the command line of the application as the parameter. The buildFVector method parses the XICL specification file and creates an op instance for every option or operand construct in the specification. It then iterates through the op instances; in each iteration, it applies the feature extraction methods (instances of XFMethod) of the op to the corresponding values in the command line to compute the features corresponding to op. At the end, the translator obtains a feature vector containing all features of the given input.

The procedure, getMethod, enables the mapping from the names of feature extraction methods (the attr fields) in an XICL specification to the real method instances. Along with the method hash table, this procedure offers a bridge linking programmer-defined feature extraction methods and the XICL translator, as explained next.

2) Extensibility: The design of the XICL translator makes it easy to be extended. Programmers can incorporate into the XICL translator their knowledge on the extraction of certain features by simply implementing an instance of XFMethod and putting it into the package that contains the XICL translator. An example is shown in Figure 4 with the assumption that org.jikesrvm.xicl is the package including the XICL translator. Programmers can then use "mFeatureFoo" as the value of certain att fields in the XICL specification files. The XICL translator will automatically use the defined method to extract the feature from program inputs.

3) Efficient Feature Attainment: Some values computed at the early stage of the application program may well characterize certain features of inputs. If those values can be

[2]For simplicity, we assume that the application is launched by a command line that includes input arguments. We discuss interactive applications at the end of this section.

SYNOPSIS: route [options] FILE ... OPTIONS: -n N: find N shortest paths. N is 1 by default. -e, --echo: status message. Off by default.	option {name=-n; type=NUM; attr=VAL; default=1; has_arg=Y} option {name=-e:-echo; type=BIN; attr=VAL; default=0; has_arg=N} operand {position=1:$; type=FILE; attr=mNodes:mEdges}
(a) Usage of an example application	(b) XICL specification

Fig. 2. An example application with its XICL specification.

```
// XICL Translator
public class XICLTranslator {
  ...
  __XICLFeatureVector fVector;
  HashMap <String, XFMethod> xfMethodsMap;

  public XICLTranslator (){
   xfMethodsMap =
      new HashMap<String, XFMethod>();
  }

  // find the method from its name
  XFMethod getMethod (String mname){
   xFMethod md = xfMethodsMap.get(mname);
   if (md==NULL){
    Class cls = Class.forName (md);
    md = (XFMethod) cls.newInstance();
    xfMethodsMap.AddtoHashMap(mname, md);
   }
   return md;
  }//getMethod()

  //kernel for feature vector construction
  public void buildFVector(String cmdLine){
   String m; __XICLFeatureVector fv;
   //parse XICL specifications
   ops = getAllOptsOprds();
   //get features of each option/operand
   while (op = ops.next()){
    //get the value of op in the real input
    String value = op.getValue(cmdLine);
    //compute the features of op
    while (m = op.nextFeatureMethod()){
     XFMethod c = getMethod(m);
     fv = c.xFeature (value);
     fVector.append (fv);}}
  }//buildFVector()
}//XICL Translator

// Interface for feature-extraction methods
public interface XFMethod {
  __XICLFeatureVector xFeature (String s);
}//XFMethod
```

Fig. 3. XICL Translator

```
package org.jikesrvm.xicl;
public class mFeatureFoo implements XFMethod {
  public __XICLFeatureVector xFeature(String s) {
   // extracting the feature foo from s
   ... ...
  }
}
```

Fig. 4. An example illustrating how to extend XICL to extract programmer-defined features by implementing an instance of class XFMethod.

```
import org.jikesrvm.xicl.*;
public class AnApplication {
  ... ...
  public foo() {
   ... ...
   __XICLFeatureVector subv= new Vector (v);
   __XICLFeatureVector.updateV(''1mFeature'',subv);
   ... ...
  }
}
```

Fig. 5. Illustration of passing variable values in an application to XICL translator as input features.

passed to the XICL translator, they would save some feature extraction operations by the translator. The passing requires communications from the application to the virtual machine. In RVM, such communications are easy to do, because RVM is written in Java and the RVM thread shares the same address space as the application thread. Figure 5 shows an example of how to pass the value of variable v in an application into XICL translator as part of the feature vector.

In the example, the updateV method puts subv at certain positions (determined by 1mFeature) in the feature vector, which, as a static member of class __XICLFeatureVector, is the same vector filled by the XICL translator (through accesses to fVector, an instance of the class __XICLFeatureVector.) This mechanism offers a way for programmers and the XICL translator to work together for efficient input characterization. There is another interface (not shown in the example), __XICLFeatureVector.done(), which tells the virtual machine that no more values will be passed as features, so that the virtual machine can go ahead to start behavior prediction if necessary.

4) Handling Interactivity: The XICL translator is applicable to interactive applications with or without GUI (graphical user interface.) Programmers can insert __XICLFeatureVector.updateV() at the interactive points to obtain new features, and then use __XICLFeatureVector.done() to trigger a new prediction. An alternative way is to extend the grammar of XICL to allow programmers to specify, in the XICL description file, the locations of interactive points in the application (along with the potentially important attributes of the corresponding inputs.) A tool can then automatically insert the corresponding "updateV()" and "done()" function invocations at those interactive points.

IV. Incremental Learning and Discriminative Prediction

This section describes the other two major components in the evolvable virtual machine: the model builder and the strategy predictor. Both center on the mapping functions between program inputs and optimization strategies. The builder tries to construct such functions, and the predictor attempts to use those functions to predict the input-specific optimization

strategies. Before describing those components, we first explain the goal of the prediction: the optimization strategy in Jikes RVM.

A. Optimizations in Jikes RVM

The decisions in the optimizer in Jikes RVM roughly fall into two categories. High-level decisions include the compilation level for a method and whether to inline at a call site; low-level decisions include the values of the parameters used in each kind of optimizations (e.g., loop unrolling levels.)

The techniques proposed in this work may be applicable to both levels of optimizations. This paper, in particular, concentrates on one kind of high-level decisions, the compilation level of a method.

In Jikes RVM, a method can be compiled on 4 levels: -1, 0, 1, 2. The higher the level is, the longer the compilation takes, and often (not always) the faster the generated code runs. During an execution, the default Jikes RVM compiles a method using the base compiler (level "-1") at the first encounter of the method. It continuously observes the hotness (the number of samples) of a method through runtime sampling, and uses a cost-benefit model to determine whether a method should be recompiled at a higher optimization level. In the cost-benefit model, the cost is the compilation time, and the benefit is the expected time savings in the rest of the execution because of this recompilation. Jikes RVM estimates the duration of the future execution by assuming that the times a method will run is the same as the times it has already run.

The objective of the evolvable virtual machine is to predict the appropriate compilation level for a method at the early stage of the program execution. With the prediction, the JIT (Just-In-Time) compiler will be able to optimize a method on the right level much earlier than in the default RVM. It has two benefits: It saves compilation time by avoiding unnecessary recompilations of a method, and it saves execution time by generating efficient code early. We concentrate on this type of optimization because previous studies have shown that it is especially important and also challenging [2], [6].

B. Learning Technique: Classification Trees

Learning is a process to build a function mapping from inputs to outputs. In this work, the input to the function is an input feature vector, and the output is a categorical value—the good optimization level for a method. When outputs are categorical, learning is a classification problem. We select Classification Trees as the learning technique for its simplicity, flexibility, and interpretability.

A classification tree is a hierarchical data structure implementing the divide-and-conquer strategy [7]. It divides the input space into local regions, and assigns a class label to each of those regions. Figure 6 shows such an example. Each non-leaf node asks a question on the inputs and each leaf node has a class label. The class of a new input equals to the class of the leaf node it falls into.

The key in constructing a classification tree is in selecting the best questions in the non-leaf nodes. The goodness of a question is quantified by purity: A split is desirable if after the split, the data in each subspace has the same class label. Many techniques have been developed to automatically select the questions based on the entropy theory [7].

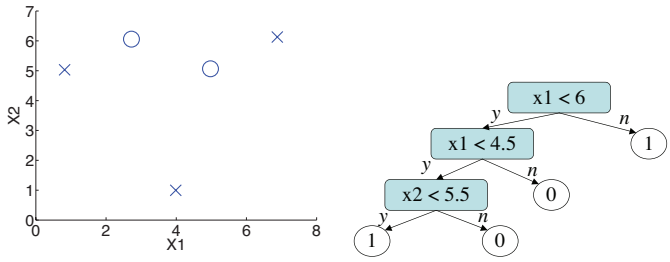


Fig. 6. A training dataset (o: class 1; x: class 2) and the classification tree.

```
/* M: the predictive model to build or refine. */
/* conf: confidence of the current model M. */
/* When a java program launches with input I. */
if (conf > THc)
  v = GetInputVector(I);
  ô = PredictOptStrategy(M, v);
  run the program with ô to its end, meanwhile
    collecting sampled profile p;
else
  run program in the default way to its end,
    meanwhile collecting sampled profile p;
  v = GetInputVector(I);
  ô = PredictOptStrategy(v);
end
o = GetIdealOptStrategy(p);
acc = CalAccuracy(ô, o, p);
conf = (1−γ)*conf + γ*acc;
UpdateModel(M, v, o);
```

Fig. 7. Incremental learning with confidence.

The technique of Classification Trees suits the learning tasks in this work because it handles both discrete and numeric features, is efficient and easy to use, and has good interpretability. The last but not least reason is that it automatically selects the important features. The input vectors constructed by XICL interpreter tend to contain more features than necessary. This is because the XICL specification usually includes all possible input options of the program. However, in a user's real uses of the program, many of the options are typically not invoked. The features corresponding to those options will thus have the default values in the feature vectors of all runs. During the construction of classification trees, because those features won't cause any impurity reduction, they will not appear in the trees. This automatic feature selection allows programmers to specify the features about whose importance they are not sure. Moreover, by excluding the unimportant features, the prediction accuracy is typically better due to less noise.

C. Learning and Prediction Strategy: Incremental Learning with Discriminative Prediction

The strategy to learn the relation between inputs and optimization levels is to build and refine the predictive models (i.e. the classification trees) gradually across production runs of the application. Figure 7 shows the pseudo code of the learning.

A key feature of the learning scheme is the confidence

measurement, conf, as a guard to prevent poor predictions. The confidence is quantified as the decayed average of the accuracies of the prediction on previous executions. It enables discriminative prediction—only predicts when confident.

The learning scheme works in this way. Initially, the predictive model M is empty and the confidence level conf is 0. During an execution, the default runtime sampler collects a profile p including the numbers of samples for every Java method. At the end of the execution, the virtual machine, based on p, computes the ideal optimization strategy o, for every method based on the default cost-benefit model in Jikes RVM as explained in Section IV-A. Also at the end (or beginning if conf is high enough) of the execution, the virtual machine uses the current model M to generate a predicted optimization strategy $\hat{o}$. The confidence level conf is then updated using the prediction accuracy, obtained by comparing $\hat{o}$ and o. The model M is updated with the new input and the ideal strategy o.

At the beginning of an execution, only if conf is large enough, the virtual machine applies the model to the new input and uses the predicted optimization strategy to optimize the new run.

The prediction accuracy used for the calculation of conf is defined as the fraction of the total time that the program spends on the methods whose optimization levels are predicted correctly. The formula is

$$accuracy = \frac{\sum_{m \in C} T_m}{\sum_{i \in A} T_i},$$

where, A is the set of all the methods in the program, C is the set of methods whose optimization levels are predicted correctly (i.e., $C = \{m : o(m) = \hat{o}(m)\}$.) T_m and T_i are the running times of a method, which, in Jikes RVM, are the numbers of times the method is sampled during an execution.

There are two parameters in the algorithm. The decay factor (γ) determines the relative weights of the recent and the older running history. The larger it is, the more the recent runs weight. The confidence threshold (TH_c) determines the risks of applying the predictive model; the larger it is, the more conservative the scheme is. Both values should be between 0 and 1. Like many parameters used in virtual machines, those values should be selected through empirical experiments. We use 0.7 for both in our experiments.

V. Evaluation

We select 11 programs from 3 benchmark suites to form a mix of different types of applications, shown in Table I. The use of a subset of the benchmark suites is mainly due to the difficulty in the creation and collection of inputs. We do not choose a benchmark if its input is too difficult to collect or create. (Note that in the real uses of evolvable virtual machines, such a difficulty does not exist because the inputs are automatically provided by users in the real uses of the application.) Furthermore, it is common in the construction of a benchmark suite that some benchmarks are obtained by simplifying the original applications. Many input options of the original applications may be disabled to make the benchmark interface simple. Given that input is the focus of this work, we select the 11 programs that are close to the original application in terms of the usage and interface.

The machine we use is equipped with 1.6GHz Intel Xeon E5310 processors, running Linux 2.6.9. All experiments use Jikes RVM 2.9.1 as the virtual machine.

A. Program Input Characterization

In the benchmark suites, each program usually comes with only one or two inputs, which are insufficient for a systematic study of input influence. We collect more inputs as shown in the second column of Table I. For some programs, such as Search, we have only few inputs due to the special requirements of the programs on their inputs. 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. To make the benchmarks close to real applications, for some programs (e.g., Mtrt, Antlr, Bloat), we enable some of their command-line options that were disabled by the benchmark suite interface.

As described in Section IV-B, the construction of classification trees can automatically select the important features from feature vectors. The third and fourth columns of Table I show the numbers of raw features and the selected features.

The development of the XICL descriptions for those benchmarks has taken us some efforts, but some of them are not necessary in real uses. We first read the code and documentations to understand each benchmark and its required inputs. This step took us most of the time. Fortunately, in real uses of XICL, these efforts are typically unnecessary, given that programmers are likely to understand the programs they developed well. The second step, writing the XICL description, is more straightforward and took us no more than 2 hours for a benchmark: XICL is mini, containing only two constructs. Besides XICL-predefined features, there are totally 4 user-defined features: the sizes of database and queries for Db, the number of rules for Antlr, and the lines of code for Bloat. None of them needs more than 50 lines of coding. Some fine tunings to the descriptions may yield better characterization of the benchmarks inputs, but for unveiling the basic effectiveness of the evolvable virtual machines, we avoid posterior tunings in our experiments.

B. Selection of Optimization Levels

In this experiment, we run each program 30 times (or 70 times for programs with many inputs) under each of three scenarios. Every run uses one randomly picked input from the program's input set. The three scenarios include those when the programs run in our evolvable RVM (denoted as *Evolve*), when they run with the repository-based optimizer (denoted as *Rep*) [2], or when they run in the default Jikes RVM (denoted as *Default*.) These scenarios represent three

TABLE I
BENCHMARKS

Program	# Inputs	Running time (s)		Input features		Prediction	
		Min	Max	Total	Used	conf	acc
Compress^j	20	0.94	9.33	3	1:file size	0.92	0.94
Db^j	54	0.59	98.16	11	2:sizes of database and queries	0.84	0.86
Mtrt^j	62	0.26	6.37	2	2:input values	0.81	0.82
Antlr^d	65	0.15	0.19	25	3:number of rules, output format, language type	0.81	0.83
Bloat^d	55	0.08	41.46	23	2:lines of code, operation type	0.82	0.85
Fop^d	57	0.59	1.53	27	2:file lines, output format	0.82	0.84
Euler^g	6	0.93	7.79	1	1:input value	0.86	0.91
MolDyn^g	12	0.11	63.05	1	1:input value	0.77	0.81
MonteCarlo^g	14	9.07	15.81	1	1:input value	0.78	0.83
Search^g	5	2.74	210.36	2	1:length of input string	0.91	0.96
RayTracer^g	12	3.10	236.57	1	1:input value	0.80	0.85

j: jvm98; d: dacapo; g: grande

different strategies in handling program inputs during dynamic optimizations. *Evolve* learns across runs, making dynamic optimizations proactive and input-specific. *Rep* learns from history repository, but does not tailor optimization strategies to program inputs. *Default* uses no history runs, employing a pure reactive optimization scheme.

More specifically, the technique in *Rep*, proposed by Arnold and his colleagues [2], produces an optimization strategy for each method in a program. The 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. Our implementation is based on the authors' description in their paper [2] (with the compilation bound included.)

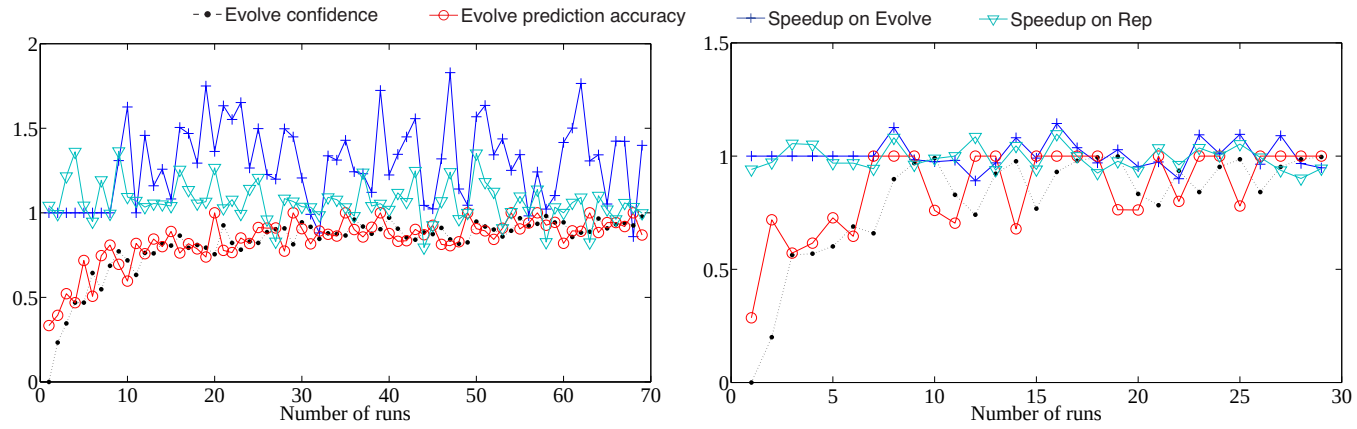
In contrast, the predictor in *Evolve* produces only one number (l) for each method, indicating the best optimization level the method should be compiled in the *current* run. For all methods, the evolvable RVM will compile them on the "-1" level at the first time the method is invoked. For a method whose value of l is greater than "-1", the RVM generates a recompilation event right after the first-time compilation of the method so that the JIT compiler will recompile the method on level l. Using level "-1" for the first-time compilation is important for avoiding too-early optimizations, which may miss some optimization opportunities due to unresolved references. (*Rep* uses the similar strategy to avoid too-early optimizations.)

1) Experimental Results: Figure 8 uses Mtrt and RayTracer as examples to show the effects of the incremental learning in the evolvable virtual machine. In both graphs, as the virtual machine sees more runs of the programs, the prediction accuracy (the circles) of the predictive models increases gradually, and the confidence curves (the dots) show the similar ascending trend. In the first few runs, as the confidence is low, *Evolve* uses the default optimization scheme in RVM, and the performance of the programs (the pluses) are similar to their default runs. When the confidence is higher than the threshold (0.7), *Evolve* applies the predicted input-specific optimizations, which make the programs run significantly faster. The speedup of Mtrt reaches up to 180%, and up to 10% for RayTracer. The speedup demonstrates the benefits from the proactivity in the optimizations. For comparison, the graphs also show the speedup from *Rep* (the triangles.) *Evolve* outperforms *Rep* by up to 44% (25% on average) on Mtrt and slightly (2% on average) on RayTracer. This comparison shows the benefits from input-specific optimizations.

a) Correlations between Running Times and Evolve *Benefits:* The speedup curves in Figure 8 fluctuate considerably because the inputs to the applications arrive in a random order, and the speedups on different inputs are different from each other. One of the major effects from program inputs is on the program running times. In this section, we examine the correlations between running times and the benefits of *Evolve* using two representative programs, Mtrt and Compress.

Figure 9 (a) shows the speedup and running times of 92 runs of Mtrt, sorted in an ascending order of the running times of the program in the default Jikes RVM. The speedup data of *Evolve* are obtained in the incremental learning process. For *Rep*, we use the strategy obtained from the histogram of all runs to avoid warmup issues. (We exclude from the figure the 8 runs that happen at the beginning of the learning as *Evolve* conducts no prediction in those runs.) The solid curves in Figure 9 (a) shows a clear correlation between running times and the speedup from *Evolve*. The existence of the correlation is intuitive. As mentioned earlier, *Evolve* has two benefits: It saves compilation time by avoiding unnecessary recompilations, and it saves execution time by generating efficient code early. The two kinds of benefits tend to be more obvious on a relatively long execution. In those executions, more methods are likely to be compiled multiple times in the default JVM than in a very short execution, and more methods are likely to need high-level optimizations, thus resulting in larger potential benefits for the early compilation by *Evolve*. Given that *Rep* is similar to the default RVM in requiring multiple compilations to a method (but in a better strategy), the same reasons explain the increasing gap between the *Evolve* and *Rep* curves in Figure 9 (a).

However, as the program running time increases even further, the benefits from *Evolve* would eventually decrease and diminish. For Mtrt, when the inputs become "700 1000" (not shown in Figure 9 (a) for legibility), the default RVM takes more than 20 seconds to finish, and the speedup from *Evolve* falls to less than 1.08. The reason for the diminishing benefits is that both the compilation time savings and the execution time savings from the early compilation would



(a) Mtrt (b) RayTracer

Fig. 8. Temporal curves of the confidence and prediction accuracy of *Evolve*, along with speedup comparisons between evolvable RVM (*Evolve*) and repository-based RVM (*Rep*).

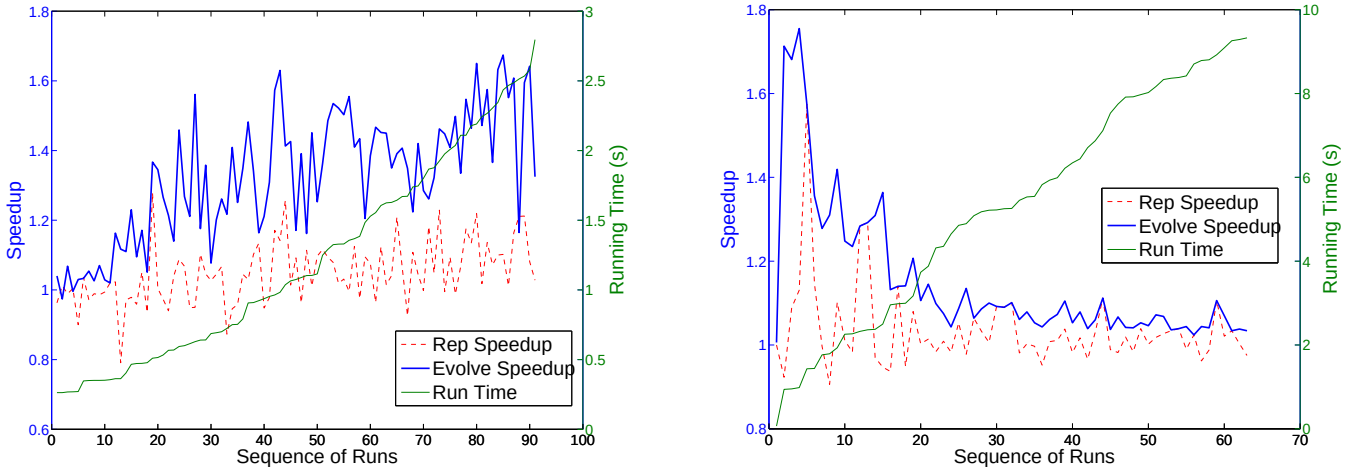
become negligible compared to the long program execution. This phenomenon is clearly exposed in the results of Compress as shown in Figure 9 (b). But we note that this phenomenon should not lead to the conclusion that the idea of *Evolve* cannot apply to long-running server programs. Typically, those programs receive different requests continuously, and because different requests often trigger different behaviors of the program and result in different resource demands and influence on computing resources, the concept of *Evolve* may yield proactive, request-specific optimizations and resource management. The details are out of the scope of this paper.

b) Results on All Benchmarks: The boxplots in Figure 10 shows the speedups of all the 11 programs in *Evolve* and *Rep*, normalized by the performance in the default RVM. Each box presents the distribution of the speedups in all the runs of a benchmark. The benchmarks form two groups: Mtrt, Compress, Euler, MolDyn, RayTracer are more input-sensitive than the other benchmarks. On *Evolve*, they show 14% average median-speedup and 37% average max-speedup, outperforming the performance on *Rep* by 10% and 18% respectively. Overall, *Evolve* improves the performance of all the 11 programs by 7–21% on average. With the discriminative prediction, *Evolve* successfully prevents some unprofitable predictions, reflected by the better minimum speedups in 9 programs than the results in *Rep*. On Mtrt and RayTracer, the minimum speedups in *Evolve* are slightly worse than in *Rep*. However, both the boxes in Figure 10 and the curves in Figure 8 show that most of the runs of those two programs have better performance in *Evolve* than in *Rep*. The rightmost columns in Table I reports the average confidence and prediction accuracies of all the 11 benchmarks.

2) Overhead Analysis: The evolvable virtual machine includes 3 parts of extra overhead compared to the default virtual machine. The first part contains the time spent on feature extraction by the XICL translator. The second part includes the time spent in the prediction of optimization levels. The third part consists of the construction of the predictive models. (Runtime profiling does not increase the overhead as it already exists in the default RVM.) The third part occurs only after the execution of the application, so it does not increase the runtime of the application. The other two parts are included in the performance shown earlier in this section. For most runs, the overhead weights less than 0.4% of the program running time. The largest weight is 1.38% on program Bloat when the input is small. Although the overhead is negligible in this experiment, we note that there are risks that the overhead may be too much if the programmer-defined feature extraction is inefficient. This problem can be solved if the XICL translator sets an upperbound and throttles the extraction when the time exceeds the upperbound and falls back to the default optimizations.

3) Sensitivity to Thresholds and Execution Order: We conduct a series of studies to measure the effects of the accuracy threshold and confident threshold. We briefly summarize the results as follows. Higher threshold values make *Evolve* more conservative, reflected by smaller ranges of speedup. For instance, the maximum speedup of Mtrt decreases from 1.8 to 1.4 when the confidence threshold increases from 0.7 to 0.9; on the other hand, the worst performance improves from 10% to zero slowdown. We also changed the order in input sequences. The results show that *Rep* is more sensitive to the changes than *Evolve* does. For example, the worst-case performance of RayTracer in *Rep* decreases 15% for a different input order; whereas, it has no noticeable changes in *Evolve*. The reason is that *Rep* conducts prediction even when there are only few runs in the history. Different input order changes the starting set of history runs significantly. Whereas, the discriminative prediction in *Evolve* prevents most immature predictions.

As a final note, we observe that there are some differences between the results of *Rep* obtained in this experiment and the results in the previous paper [2]. Besides the difference in applications and input sets, the other reason is the differences between JIT compilers (We thank Matthew Arnold and David Grove for providing the insights.) The previous experiments



(a) Mtrt (b) Compress

Fig. 9. The correlation between the speedup brought by *Evolve* and the execution times of Mtrt and Compress. The data are collected from a sequence of runs of the applications with inputs arriving randomly; but before plotting the curves, we sort the runs in an ascending order of their default running times.

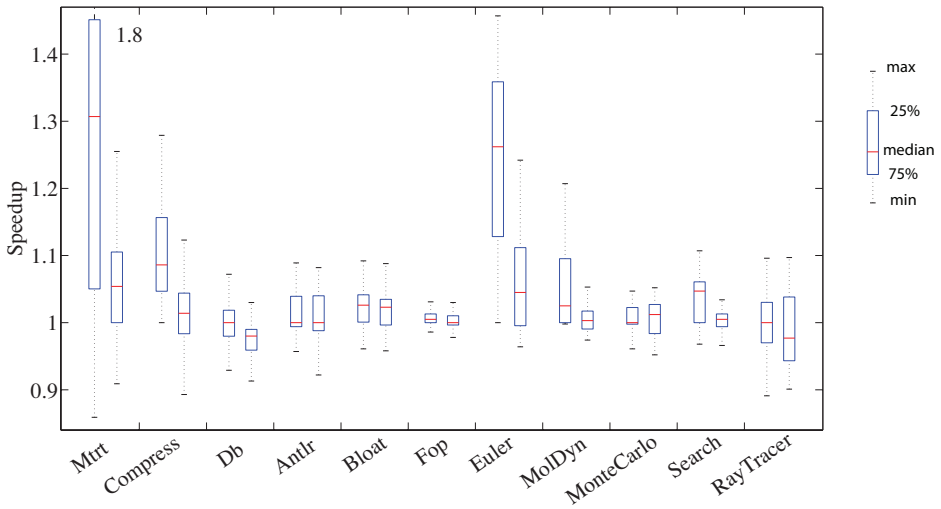


Fig. 10. Boxplot of the speedup by the evolvable RVM (*Evolve*, left in each benchmark) and the repository-based optimization (*Rep*, right in each benchmark).

use IBM J9 virtual machine, which uses an interpreter rather than a (preliminary) compiler as the baseline; so, the performance baseline is lower in J9 than in Jikes RVM. Meanwhile, the optimizing compiler in J9 is more sophisticated than the one in RVM; so, the benefit from an appropriate recompilation is more significant in J9. (For the same reasons, we expect that *Evolve* should show more significant benefits on J9 than what have been observed on Jikes RVM.)

VI. Discussion

This section discusses some issues related to the XICL specifications. First, because the specifications come from programmers, they could have mistakes, such as missing some important features. The evolvable virtual machine can well tolerate such mistakes, because the discriminative prediction scheme would prevent the uses of the resulted inferior models. A possible extension is to let the virtual machine offer feedback to the programmers for the refinement of the specifications. Second, we emphasize that the benefits on the selection of optimization levels are only part of what the XICL specifications can bring. By facilitating the resolution of input complexities, the specifications can help the virtual machine produce predictive models between program inputs and various runtime behaviors, and thus enable a wide range of proactive input-specific optimizations. Example uses, besides the selection of optimization levels, include input-specific selection of garbage collectors [12], method inlining, and loop transformations. The compound benefits from various optimizations could be significant enough to well justify the extra efforts needed for developing the XICL specifications.

VII. Related Work

Departing from static compilations, adaptive runtime optimizations [1], [4], [8], [10], [15], [17], [19] learn about program dynamic behavior patterns during production runs. Evolvable virtual machines move one step forward to enable *cross-run* learning and evolvement, making dynamic optimization proactive and input-specific.

The work closest to our study is the repository-based cross-run learning system from Arnold and his colleagues [2]. Our work differs from theirs in three major aspects. First, our technique tailors the optimization strategy for every input rather than producing a single strategy that maximizes the average performance of all past runs. Second, our technique uses self-evaluation to selectively predict optimal strategies with confidence; their technique applies the learned strategy to new inputs with no guarding. Finally, we use XICL to tackle input complexity, which is not addressed in their study.

Gu and Verbrugge [6] have recently proposed phase-based adaptive recompilation in a JVM. In their approach, the runtime system detects phases and varies the aggressiveness of recompilation in different phases based on some heuristics. Our approach is complementary to theirs in that, the cross-run learning makes it possible to predict the optimization levels that best suit the entire execution. On the other hand, phase-based adaptation offers fine-grained control in optimizations.

Our work shares the same theme as the concept of continuous compilation, such as the design of CoCo [5] and the CPO framework [18]. Their focuses are on the design of high-level architectures and modeling the influence of loop transformations or multiple levels of the software stack.

There have been many explorations that attempt to build optimization models by applying machine learning techniques to many offline training runs and conduct run-time prediction by plugging in the input characteristics of a real run (e.g., [3], [9], [11], [14]–[16].) Such techniques combine the advantages of both offline profiling and runtime adaptation. They typically require a large number of offline training runs.

The XICL framework described in this work is more general than our previous study on feature extraction [13] in that this framework exploits the runtime values in the original program, is compatible to interactive programs, and is suitable for incremental learning in virtual machines.

VIII. Conclusions

In this paper, we describe a set of techniques to make virtual machines more intelligent. We present a specification language and its translator for addressing input complexity. We propose a learner that is able to learn the relation between program inputs and good optimization decisions. With self-evaluation-based discriminative prediction, the learner enables a virtual machine to tailor its optimization decisions to each input, proactively and confidently. Experiments show the promise of the techniques in overcoming the inherent limitations of reactive optimizations in current virtual machines, opening many opportunities for proactive optimizations and cross-run evolvement of virtual machines.

Acknowledgments

We are especially grateful to David Grove for his insightful suggestions and great help to the final version of this paper. We thank Matthew Arnold, Mark Stoodley, Clark Verbrugge, and the anonymous reviewers for their valuable comments. This material is based upon work supported by the National Science Foundation under Grant No. CSR-0720499 and CCF-0811791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] 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*, 2002.

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

[3] John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM conference on programming language design and implementation*, pages 183–194, 2004.

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

[5] 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.

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

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

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

[9] C. Krintz. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

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

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

[12] F. Mao and X. Shen. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2009.

[13] 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.

[14] 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*, 2007.

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

[16] 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 PPoPP*, 2005.

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

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

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