Frequency Estimation of Virtual Call Targets for Object-Oriented Programs

Cheng Zhang[1], Hao Xu[2,⋆], Sai Zhang[3], Jianjun Zhao[1,2], and Yuting Chen[2]

[1] Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2] School of Software, Shanghai Jiao Tong University
{cheng.zhang.stap,steven_xu,zhao-jj,chenyt}@sjtu.edu.cn
[3] Computer Science & Engineering Department, University of Washington
szhang@cs.washington.edu

Abstract. The information of execution frequencies of virtual call targets is valuable for program analyses and optimizations of object-oriented programs. However, to obtain this information, most of the existing approaches rely on dynamic profiling. They usually require running the programs with representative workloads, which are often absent in practice. Additionally, some kinds of programs are very sensitive to run-time disturbance, thus are generally not suitable for dynamic profiling. Therefore, a technique which can statically estimate the execution frequencies of virtual call targets will be very useful.

In this paper we propose an evidence-based approach to frequency estimation of virtual call targets. By applying machine learning algorithms on the data collected from a group of selected programs, our approach builds an estimation model to capture the relations between static features and run-time program behaviors. Then, for a new program, the approach estimates the relative frequency for each virtual call target by applying the model to the static features of the program. Once the model has been built, the estimation step is purely static, thus does not suffer the shortcomings of existing dynamic techniques. We have performed a number of experiments on real-world large-scale programs to evaluate our approach. The results show that our approach can estimate frequency distributions which are much more informative than the commonly used uniform distribution.

1 Introduction

Most of the object-oriented programming languages provide the virtual call mechanism to support polymorphism. While enhancing the modularity and extensibility in both design and implementation, virtual calls also complicate the static call graphs by adding extra branches at the call sites. As a result, call graph-based program analyses and optimizations may become less effective. In addition, virtual calls may cause significant performance overhead, because the

⋆ He is currently a graduate student at Department of Computer Science, University of Southern California.

exact callees must be determined at run-time by selecting them from all the candidates based on the receiving objects (this process is often called *dynamic binding*). Therefore, it is important to resolve virtual calls at compile time or to obtain information about the execution frequencies of the targets (i.e., callees) for the unresolved virtual calls.

An empirical study [16] has shown that the distribution of execution frequencies of virtual call targets is highly peaked, that is, a small number of methods are frequently called in most of the run-time virtual calls. Thus it is rewarding to find the most frequently executed targets. Various dynamic profiling techniques are developed to explore the frequency distribution or relative frequencies of virtual call targets. Some of them [12] [16] [19] [14] [21] achieve high accuracy with relatively low overhead. However, in order to generate useful profiles, most of the dynamic techniques require driving programs with representative workloads, which are often absent, especially for newly developed programs. Moreover, dynamic techniques are usually intrusive in that they have to instrument programs to collect information, whereas some kinds of programs (e.g., multi-thread programs) may be extremely sensitive to run-time disturbance. In these cases, a static technique with acceptable accuracy could be a preferable alternative.

In this paper, we propose *Festival*, an evidence-based approach to **f**requency **esti**mation of **v**irtual c**al**l targets. The underlying assumption of Festival is that developers' design intentions, which cause the imbalance of usage of different virtual call targets, can be revealed by examining a group of static program-based features. Festival consists of two phases: 1) model building and 2) estimation. In the model building phase, Festival selects a set of existing programs with representative workloads, extracts some static features, and runs the programs to get the dynamic profiles for their virtual call targets. Based on the collected data, Festival uses machine learning algorithms to discover the relationship between the execution frequencies and the static features. The relationship is represented as an artificial neural network. As a prerequisite for model building, we assume that the programs and their representative workloads are available. This assumption is reasonable, because there exist numerous object-oriented programs, which have been used in practice for years. The accumulated workloads for such programs are probably representative. In the estimation phase, for a new program, Festival extracts the same set of features from it and uses the model to estimate the relative frequencies of the virtual call targets in the program. It is worth noting that, once the model has been built, the estimation phase is purely static. We have implemented a prototype of Festival and performed a set of experiments on the DaCapo benchmark suite [7]. The experimental results show that the estimated frequency distributions are significantly more informative than the uniform distribution which is commonly used in static analyses.

The main contributions of this work can be summarized as:

1. Festival, the first evidence-based approach we are aware of to estimate frequencies of virtual call targets for object-oriented programs. It can be a good complement to existing dynamic techniques. As will be discussed in Section 3, a variety of client applications may benefit from our approach.

2. An evaluation conducted to validate the effectiveness of our Festival approach. It consists of a comprehensive group of experiments, which show the estimation performance of Festival from various aspects.

The rest of this paper is organized as follows. Section 2 uses an example to give a first impression of the static features. Section 3 discusses a number of potential applications of our Festival approach. Section 4 describes the technical details of the approach. Section 5 shows the experimental results. Section 6 compares Festival with related work and Section 7 concludes the paper and describes our future work.

2 Motivating Example

In this section we use a real-world example to illustrate some of the features used in our approach. The features are program-based and easy to extract using static analysis. Nevertheless, we believe that they are related to the run-time execution frequency of virtual call targets.

The code segments shown in Figure 1 are excerpted from ANTLR (version 2.7.2) [1], a parser generator written in Java. From the code, we can see that class `BlockContext` contains three fields and three methods, while its subclass `TreeBlockContext` has only one field and one method, `addAlternativeElement`, which overrides the implementation provided by `BlockContext`. The virtual call of interest is at line 6. The method `context` (whose definition is omitted for brevity) has a return type `BlockContext`. Thus the virtual call has two possible targets: one is the method `addAlternativeElement` defined in `BlockContext` and the other is the one defined in `TreeBlockContext`. By running ANTLR using the workload provided in DaCapo benchmark (version 2006-10-MR), we obtained the dynamic profiles of these two targets and found that the execution frequency of the method defined in `BlockContext` is about ten times higher than that of the method defined in `TreeBlockContext`. But what if we cannot run the program, say, because the workload is unavailable? Can we make a good guess at the relative frequencies of these two targets?

If we analyze the program source code, some informative evidences can be discovered. First, `TreeBlockContext` is a subclass of `BlockContext`. As a general rule of object-oriented design, the subclass (i.e., `TreeBlockContext`) is a specialized version of the superclass (i.e., `BlockContext`). Second, because the method `addAlternativeElement` has a concrete implementation in `BlockContext` rather than being abstract, it is probably designed to provide common functionalities, while the method in `TreeBlockContext` is designed for special cases. Third, if we explore the calling relations between relevant methods, we will find that the method defined in `TreeBlockContext` calls its super implementation (the call is at line 36 in Figure 1). It indicates that `TreeBlockContext` delegates a part of its responsibility to its superclass. At last, there are three fields and three methods defined in class `BlockContext`, while class `TreeBlockContext` has only one field and one method. The fact that `BlockContext` has higher complexity may also show its relative importance. Based on these evidences we are likely to consider

```
public class MakeGrammar
  extends DefineGrammarSymbols {
  protected void addElementToCurrentAlt(
    AlternativeElement e) {
    ...
    context().addAlternativeElement(e);}
}

class BlockContext {
  AlternativeBlock block;
  int altNum;
  BlockEndElement blockEnd;

  public void addAlternativeElement(
    AlternativeElement e) {
    currentAlt().addElement(e);
  }

  public Alternative currentAlt() { ... }
  public AlternativeElement
                currentElement() { ... }
}

class TreeBlockContext
  extends BlockContext {
  protected boolean
               nextElementIsRoot = true;

  public void addAlternativeElement(
    AlternativeElement e) {
    TreeElement tree=(TreeElement)block;
    if (nextElementIsRoot) {
      tree.root=(GrammarAtom)e;
      nextElementIsRoot = false;
    }else {
      super.addAlternativeElement(e);
    }
  }
}
```

Fig. 1. Code segments from ANTLR 2.7.2

the method in `BlockContext` as the major one, and thus correctly predict a higher frequency for it.

The example presents the intuition that design intentions can be revealed by analyzing static features. This strongly motivates us to use such kind of features to estimate relative frequencies of virtual call targets. However, it is still challenging to tell how a specific feature may indicate the frequencies. Moreover, when there is a large amount of feature data extracted (especially for large programs), different features may lead to contradictory judgements in some cases. Consequently, the problem of how to make optimized estimations based on such kind of features motivates us to leverage the power of machine learning techniques, which are devised to discover useful knowledge from data.

3 Potential Applications

Various techniques dependent on frequency information of virtual call targets may benefit from Festival. On one hand, traditional profile-guided techniques can use the estimated profiles when dynamic profiles are unavailable. It makes them applicable in more situations. On the other hand, static techniques may achieve better performance by using more accurate information.

Program optimizations usually use dynamic profiles to help make economic optimization decisions. Nevertheless, when dynamic profiling is inappropriate, Festival can be a good substitution. Sometimes it may be integrated into the optimization process more conveniently than dynamic profilers. For instance, as shown in Figure 2, when performing the class test-based optimization [16], the compiler can insert a test for the dominant class (i.e., the class which defines the most frequently executed target method) and statically determine the target method in the successful branch. Since the test is mostly successful at run-time, the overhead of dynamic binding can be reduced. If Festival is used to identify the dominant class, this optimization can be performed without running the program.

```
Before optimization:
TypeA a = ...;
a.method();

After optimization:
TypeA a = ...;
if( a instanceof DominantSubtypeOfA ){
    ((DominantSubtypeOfA)a).method();
}else{
    a.method();
}
```

Fig. 2. An example of code optimization

Another application of Festival may be the probabilistic program analyses. Besides computing the *must* or *may* behaviors of programs, probabilistic program analyses [5][18] also show the likelihood of each *may* behavior's occurrence. For example, the probabilistic points-to analysis [18] assigns a frequency distribution to each points-to set indicating which memory locations are more likely to be the target of the pointer. Since the existing work focuses on C language, it performs analysis based on control-flow graphs which do not involve virtual calls. If the work is extended to handle object-oriented languages (e.g., Java), it can use Festival, at the beginning of its analysis, to allocate probabilities to virtual call targets instead of assuming a uniform frequency distribution.

For static bug finding tools (e.g., Findbugs [17]), high false positive rate is a major obstacle to their applications. Thus alert ranking methods are introduced to reduce the effort for finding real warnings. Some methods (e.g., [9]) rank alerts in terms of the execution likelihood of program elements. They calculate the likelihood by propagating probabilities along the edges of control-flow graphs. Using the frequency information of virtual call targets, the propagation may be more accurate when it has to branch at virtual call sites. Then better results of alert ranking may be obtained.

In summary Festival can be useful to call graph-based analyses and optimizations that deal with object-oriented programs. While the frequency estimation only needs static features of programs, a model must be built beforehand. In the next section, we will describe the technical details.

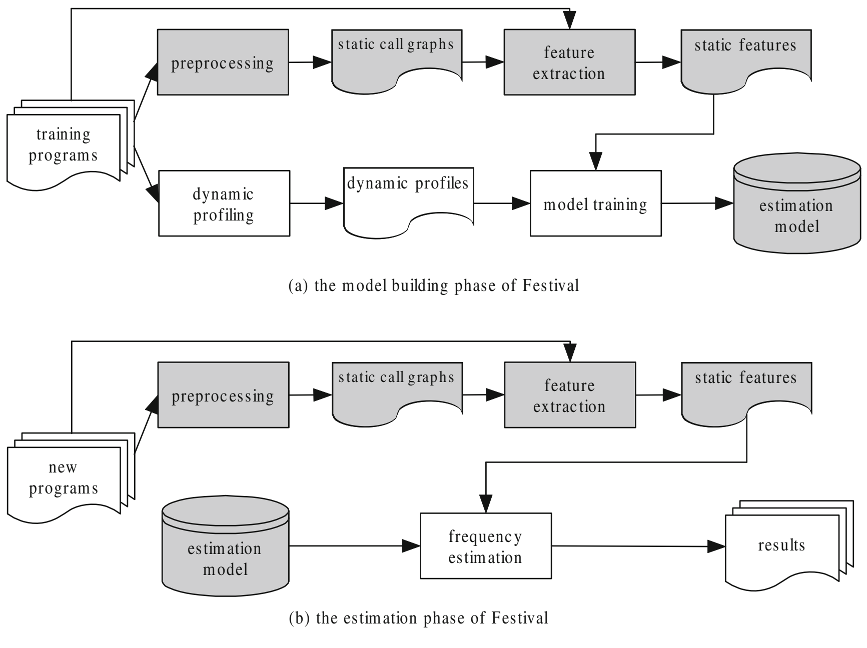

Fig. 3. The architecture of Festival which consists of two phases. In the first phase, static features and dynamic profiles are used to build the estimation model. In the second phase, only static features are extracted from new programs to perform frequency estimation. The shaded elements stand for the steps or entities involved in both phases.

4 Approach

Figure 3 shows the architecture of Festival, which consists of two phases: a) model building and b) estimation. At the beginning of the model building phase, a set of selected programs are preprocessed to construct their static call graphs. Then a group of static features are extracted based on the call graphs and the programs' source code. Meanwhile dynamic profiling is performed to get dynamic profiles for these programs. In the end of this phase, the estimation model is built based on the static features and dynamic profiles using machine learning algorithms. Once the model has been built, it will be repeatedly used afterwards. In the estimation phase, static features are extracted, in the same way, from the new programs which are not used in model building. Then the relative frequencies for virtual call targets are estimated by applying the estimation model to the features.

4.1 Preprocessing

The purpose of the preprocessing step is to construct precise static call graphs for feature extraction. In this step, we use the points-to analysis provided by the Spark framework [20] to compute the types of objects that may be referenced by each variable. Based on the type information of the receiver variable, the set of possible targets for each virtual call can be computed more precisely than traditional algorithms used for call graph construction (e.g., CHA [13]). As a result, a number of single-target virtual calls are resolved before the subsequent steps. The preprocessing step enables our approach to focus on real multi-target virtual calls[1] in order to handle large-scale programs. Hereafter the term "virtual call" means multi-target virtual calls, unless we explicitly state that a virtual call is single-target.

4.2 Static Feature Set

A static feature can be viewed as a specific measure used to capture one characteristic of a virtual call target. Thus different targets may have different values for the same feature. We use 14 static features (as shown in Table 1) to characterize virtual call targets from various aspects, including type hierarchy, calling relation, naming style, program complexity, etc.

Type hierarchy features. Features 1 to 5 are designed to represent information about the type hierarchy of the classes in which the target methods are defined. Hereafter we call such a class as a *target class* and all the target classes of a virtual call comprise the *target class set* (of that virtual call). For a specific virtual call site, the receiver variable has its explicit type (we call it the *called type*) and each target class must be either the called type itself or a subtype of the called type. Figure 4 shows the class diagram of an example for illustrating the five features related to type hierarchy. As shown in the figure, `A` is an interface which has three implementers `B`, `C`, and `D`. Class `D` extends class `C` and class `C` implements interface `F` which has no inheritance relationship with interface `A`. In addition, classes `B`, `C`, and `D` all have their own implementations of method `m`. Suppose that at a call site the method `m` is called on a variable of type `A`. In this case, the called type is `A` and the target class set is $\mathrm{TC}(\texttt{A}) = \{\texttt{B}, \texttt{C}, \texttt{D}\}$.

[1] Since the problem of points-to analysis is undecidable in general, some real single-target virtual calls may still be regarded as multi-target.

Table 1. Static features used in Festival

Number	Feature Name	Feature Description
1	Type Distance	the number of levels of subtyping from the called type to the target class
2	Subclass in TC	the number of subclasses of the target class in the target class set
3	Superclass in TC	the number of superclasses of the target class in the target class set
4	Subtree Size	the number of subclasses of the target class in the whole program
5	Number of Ancestors	the number of supertypes of the target class in the whole program (except for library types)
6	Does Call Super	whether the target method calls its super implementation (yes or no)
7	Number of Callers	the number of methods which call the target method
8	Package Depth	the depth of the package of the target class
9	Name Similarity	the number of classes whose names are similar to that of the target class
10	Number of Methods	the number of methods defined in the target class
11	Number of Fields	the number of fields defined in the target class
12	Is Abstract	whether the target class is abstract (yes or no)
13	Is Anonymous	whether the target class is anonymous (yes or no)
14	Access Modifier	the access modifier of the target class (public, protected, private, default)

Feature 1 (type distance) measures the distance between a target class and the called type, that is, it records the number of edges on the path between the corresponding nodes on the class diagram. If there is more than one path, the shortest path will be used for this feature. In the example, the feature value of both `B` and `C` is 1, while that of `D` is 2. A possible heuristic may be that the target method whose class has shorter distance to the called type will have higher execution frequency, since the target class is more general and likely to be designed for handling common cases. Features 2 (subclass in TC) and 3 (superclass in TC) encode the inheritance relations among the target classes **in the same target class set**. For example, `C` has the values 1 and 0 for feature 2 and feature 3, respectively, because `C` is the superclass of `D` which also belongs to TC(`A`) and `C` has no superclass in TC(`A`). Meanwhile, as there is no subclass or superclass of `B` in TC(`A`), `B` has the value 0 for both of the features. Sometimes target classes are "parallel" to each other, that is, there are no inheritance relations between them, such as `B` and `C`. Thus, for each target class, we use features 4 (subtree size) and 5 (number of ancestors) to count the numbers of its subclasses and supertypes in the whole program so that the relative importance of the "parallel" target classes can be characterized. In the example, the values of features 4 and 5 for `B` are 0 and 1, while they are 1 and 2 for `C`. Note that library types (e.g., `java.lang.Object`) are not counted in these two features.

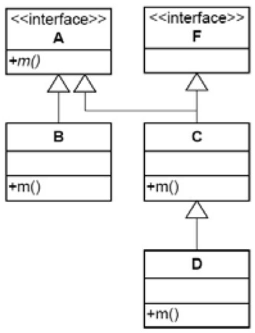

Fig. 4. An example class diagram

Call graph features. Similar to type hierarchy, calling relations may also provide hints to design intentions. Thus two of the features are based on call graph. Feature 6 (does call super) shows whether the target method calls the implementation of the superclass. If the target method just overrides its super implementation, it is not so perceivable which of the implementations is designed to take more responsibility. On the contrary, as discussed in Section 2, if the target method calls its super implementation, it is very likely that it delegates a part or all of its job to the callee. As a result, the super implementation may have much higher execution frequency. The other call graph feature, feature 7 (number of callers), tries to indicate the target method's popularity in the scope of the whole program. The intuition is that the more callers a target method has, the more popular it is. High popularity means high probability for the method to be a central part of the program, which may result in high execution frequency of the method in run-time virtual calls. Note that feature 7 counts in all the methods that explicitly call the virtual call target. In other words, its value is equal to the number of incoming edges of the virtual call target in the static call graph. Among these call graph edges, some represent resolved (single-target) virtual calls, while others represent unresolved (multi-target) virtual calls. In general, we focus on the latter, that is, we record (during dynamic profiling) and estimate execution frequencies only at the unresolved call sites. However, in feature 7, we take into account both resolved and unresolved virtual calls to a target method in order to characterize its popularity.

Naming style features. In practice several naming conventions are used to organize program elements in terms of their functionalities. A typical example is the name space mechanism provided in various programming languages. In Java, where name spaces are specified by package names, the depth of package may indicate the specialty of functionality of the classes in the package. Therefore, feature 8 (package depth) is designed to represent this characteristic. From another viewpoint on naming style, feature 9 (name similarity) counts the number of classes whose names are similar to that of the target class. If some classes have similar names, they may collaborate with each other to accomplish the same task. The more classes involved, the more important the task may be. Again the relative importance can be used to estimate execution frequencies. When computing the value of feature 9 for a given target class, we find similar class names in the following steps:

1. Find the longest common suffix of the names of the classes in the target class set.
2. Identify the special prefix for the given target class by removing the longest common suffix from its name.
3. Within the whole program, the class names that begin with the special prefix are considered as similar to the name of the target class.

Suppose, for a specific virtual call, there are two target classes `WalkingAction` and `DrivingAction`. Then the longest common suffix is "Action" and the special prefix is "Walking" for `WalkingAction`. Therefore, the number of classes (all over the program) whose names begin with "Walking" is recorded as the value of feature 9 for the target method defined in `WalkingAction`. Note that library classes are not taken into consideration during the computation, because their names usually do not indicate the design of application classes.

Complexity features. In some cases, program complexity metrics can be used to represent the importance of a program element. Currently we use two simple metrics, features 10 (number of methods) and 11 (number of fields), to measure the complexity of the target classes, because these two features are easy to extract and have satisfactory predictive power. It is worth noting that we do not add complexity of inner classes or anonymous classes to their outer classes, because we believe they are less coherent to the outer classes than the member methods and fields. We have also tried other common metrics (e.g., line of code), but found them less indicative. In our future work, we are planning to investigate some more complex metrics, such as the depth of nesting loops and the number of program paths.

Other features. Features 12 (is abstract), 13 (is anonymous), and 14 (access modifier) are mainly about surface properties of the target class. These features are used to capture design intentions from aspects other than the aforementioned ones. For example, a class is defined as abstract (instead of an interface) probably means that it provides some method implementations that will be reused by its subclasses. Thus feature 12 may be useful when we investigate the frequencies of the implemented methods.

In general the static features are selected to represent evidences which are supposed to be indicative of execution frequencies. A variety of heuristics may be proposed based on these features. Nevertheless, counterexamples may be found against each heuristic by checking the dynamic profiles, and there may also be contradictions between indications of different features. Thus we use machine learning techniques to analyze the feature data and discover relatively consistent knowledge in order to make efficient frequency estimation.

4.3 Estimation Model

We formulate the frequency estimation problem as a supervised classification problem [24] in machine learning. The task of classification is to determine which category (usually called *class*) an instance belongs to, based on some observable features of the instance and a model representing the existing knowledge. A classification problem is said to be supervised when the model is trained (i.e., built) using instances (called *training instances*) whose classes have already been specified. A typical example of supervised classification is to predict the weather condition (sunny, cloudy, or rainy) of a specific day based on some measures (temperature, humidity, etc.) of that day and a forecast model derived from historical weather record. In Festival, we build a model to classify each virtual call target as *frequent* or *infrequent* and use the predicted probability of being frequent as the estimated frequency.

In essence a model is a parameterized function, which represents the relations between its input and output. In Festival, for a specific virtual call target, the input of the estimation model is the values of the target's static features, and the output is the estimated frequency for the target. Therefore, the estimation model of Festival correlates static features with actual frequencies, providing a way to estimate unknown frequencies of new targets on the basis of the targets' static features. A key assumption of Festival is that the relations between static features and dynamic behaviors of virtual calls are stable across different programs. In other words, we can build a model based on some programs and use it to estimate frequencies for others.

In order to build the estimation model, we first select a set of programs (called *training programs*) and extract their values for the static features described in Section 4.2. Then we instrument the training programs at each virtual call site and run them with their representative workloads. The run-time execution count of each target is recorded during the execution.[2] Since the workloads are representative, the execution counts can be used as the real execution frequencies of the targets. When both static features and dynamic profiles have been obtained for training programs, we are ready to build the model.

During model building, each virtual call target corresponds to an instance which is represented as a vector $< f_1, f_2, f_3, ..., f_{14}, c >$, where f_i is the value of the ith feature and c is the recorded execution count. We have to process the instance data to make them fit for our approach. Because targets from different virtual calls are used together for model training, their feature values should be measured relatively within each virtual call. Thus we normalize the feature values within all the targets of the same virtual call. For example, if the values of feature 1 for three targets (of a specific virtual call) are f_1^x, f_1^y, and f_1^z, then they will be normalized as $\frac{f_1^x}{max\{f_1^x, f_1^y, f_1^z\}}$, $\frac{f_1^y}{max\{f_1^x, f_1^y, f_1^z\}}$, and $\frac{f_1^z}{max\{f_1^x, f_1^y, f_1^z\}}$, respectively. Moreover, to specify the class of each instance, we order the targets of each

[2] More specifically, what we record is the number of times a method becomes the actual target of its corresponding virtual call, rather than the total number of times a method is called.

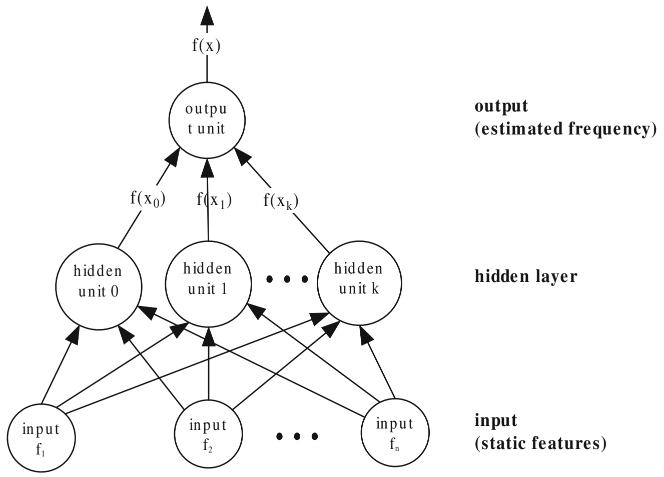

Fig. 5. The multilayer perceptron used in frequency estimation. Each input f_i corresponds to a static feature of a virtual call target and the output $f(x)$ is used as the estimated frequency of the target.

virtual call with respect to their execution counts in a descending order. Then we assign class **1** (means "frequent") to the top 20% targets and class **0** (means "infrequent") to the others. When a virtual call has less than 10 targets, we assign class **1** to the top one target and class **0** to the others. After the data processing, the instances are transformed into the form $< f_1^{'}, f_2^{'}, f_3^{'}, ..., f_{14}^{'}, L >$, where $L \in \{1, 0\}$ and $f_i^{'}$ is the normalized value of f_i. Then we use the processed instances to train a multilayer perceptron [24], which will be used as our estimation model.

A multilayer perceptron is a kind of artificial neural network which is illustrated in Figure 5. In the multilayer perceptron, the output of a node i in the hidden layer is described by

$$f(x_i) = \frac{1}{1 + e^{-x_i}}$$

where x_i is the weighted sum of its inputs, that is, $x_i = w_{i0} + w_{i1} f_1 + w_{i2} f_2 + ... + w_{in} f_n$. Using the same function, the final output $f(x)$ is computed by taking the outputs of the hidden nodes as its inputs (i.e., $x = w_0 + w_1 f(x_1) + w_2 f(x_2) + ... + w_k f(x_k)$). The final output represents the probability for an instance to be of a certain class (e.g., class **1**). During model training the weights are so computed that the total squared error of the output is minimized. The squared error of a single instance i is described by

$$error_i = \frac{1}{2}(L_i - f(x)_i)^2$$

where L_i is the class of i and $f(x)_i$ is the model's output for i. If there are totally N instances used in model training, then the total squared error is $\Sigma_{i=1}^{N} error_i$. When the weights have been established, the model training is finished.

During frequency estimation, a new target, whose frequency is to be estimated, is viewed as a new instance whose class is unknown. Thus we encode it as a feature vector $< f_1^{'}, f_2^{'}, f_3^{'}, ..., f_{14}^{'} >$ and feed the vector as the input to the multilayer perceptron. In the end, the outputted probability is considered as the estimated frequency of the target.

5 Evaluation

To evaluate our Festival approach, we have implemented a prototype on the basis of the Soot framework [4] and the Weka toolkit [24]. Soot is used to construct static call graphs[3] and extract static features, while Weka is used for machine learning. We have performed a set of experiments on the implementation prototype. Through the experiments we try to answer the following research questions:

– RQ1: What is the estimation performance of Festival?
– RQ2: Is Festival applicable to various programs?
– RQ3: What is the predictive power of each feature?

5.1 Experimental Design

Subject programs. In the experiments we use 11 programs from the DaCapo benchmark suite (version 9.12-bach) as the subject programs, because DaCapo provides comprehensive workloads for each program. Moreover, as the benchmark suite is originally designed for Java runtime and compiler research (especially for performance research), we believe the workloads are representative. Table 2 shows the basic characteristics of the subject programs. The column #M shows the number of methods that are included in the static call graph of each program, and the column #VC shows the number of virtual calls that have multiple targets as identified by the points-to analysis. From the columns LOC and Description, we can see that the subject programs are medium-to-large real-world programs used in various application domains. Thus they are quite suitable for our study on the research questions, especially *RQ*2. Note that we have not selected three programs (namely jython, tradebeans, and tradesoap) from DaCapo. Because tradebeans and tradesoap involve too much multi-threading, it is difficult for us to obtain their representative dynamic profiles. As for jython, we failed to finish the instrumentation (for dynamic profiling) within tens of hours. Although these three programs have been left out in the experiments, we believe that the selected 11 subject programs are sufficient to validate our Festival approach. Moreover, the lack of dynamic profiles does not indicate that these programs cannot be estimated by Festival. It just prevents us from using the programs for model training and evaluating Festival's performance on them.

Model training scheme. In the experiments, multilayer perceptron is used as the machine learning model in a way which is described in Section 4.3. During

[3] The aforementioned Spark framework is a building block of Soot. We use Soot 2.4.0 together with TamiFlex [8] to build call graphs that include method calls via reflection.

Table 2. Subject programs from DaCapo-9.12-bach (LOC is measured using cloc [2] version 1.51)

Name	Description	LOC	#M	#VC
avrora	simulation and analysis tool	68864	2439	34
batik	SVG toolkit	171484	5365	321
eclipse	non-gui part of Eclipse IDE	887336	18632	3007
fop	PDF file generator	96087	5503	682
h2	in-memory database	78124	3902	250
luindex	text indexing	36099	1546	107
lusearch	text searching	41153	1201	70
pmd	code analyzer	49610	4741	110
sunflow	rendering system	21960	1096	27
tomcat	web application server	158658	11554	4417
xalan	XSTL processor	172300	4366	547

Table 3. Numbers of virtual calls categorized by target number

Name	2	3	4	5	$6 \sim 10$	$\leqslant$10	>10
avrora	10	3	3	0	3	19	4
batik	52	40	3	5	7	107	12
eclipse	626	128	121	73	119	1067	380
fop	60	10	3	3	4	80	26
h2	10	4	3	15	13	45	42
luindex	31	5	5	0	1	42	0
lusearch	16	6	0	1	0	23	0
pmd	46	1	1	1	3	52	2
sunflow	11	5	4	0	0	20	0
tomcat	2309	59	17	32	62	2479	75
xalan	25	16	0	2	15	58	20
total	3196	277	160	132	227	**3992**	**561**

model training, we take the leave-one-out strategy. That is, while evaluating the performance of Festival on one specific subject program, we use the other ten subject programs as training programs. In this way, the virtual call targets, whose frequencies are estimated by the model, are never used to train that model.

Another special strategy we take for model training is to use the data only from the virtual calls whose numbers of targets are less than or equal to **10**. It is mainly due to the fact that virtual calls with too many targets may affect the training data drastically, whereas the current static features can hardly represent the information embedded in such kind of virtual calls (This limitation of Festival will be discussed in Section 5.6). Nevertheless, as shown in Table 3, most virtual calls in the subject programs have relatively small numbers of targets[4].

[4] Table 3 shows only the virtual calls that are executed during dynamic profiling, thus the total number of virtual calls is smaller than that shown in Table 2.

Therefore, our model training scheme takes into account the vast majority of the cases. Note that we obtained similar experimental results when we limited the number of targets to 5 and 15.

Platform and runtime. The experiments have been conducted on a Linux server, which has a 2.33GHz quad-core CPU and 16GB main memory. We use IBM J9 VM for feature extraction and Sun HotSpot VM for dynamic profiling. Although DaCapo provides workloads of different sizes, including small, default, large, and huge, we only use the large workloads for dynamic profiling. Because huge workloads are not available for most benchmarks, and small and default workloads are generally less representative than large ones[5]. Table 4 shows the runtime of each step in Festival, including preprocessing (PRE), feature extraction (FE), instrumentation (INS), execution (EXE), model training (MT), and frequency estimation (EST). We can see that the time cost is reasonable even for large-scale programs.

Table 4. Runtime of each step in Festival (EST is measured by second and others use the format of h:mm:ss)

Name	PRE	FE	INS	EXE	MT	EST
avrora	1:27	1:27	0:35	8:00	0:36	0.18
batik	8:21	8:22	6:46	0:09	0:34	0.16
eclipse	14:41	12:05	3:50:47	5:33	0:24	0.56
fop	14:12	14:16	12:07	0:04	0:36	0.11
h2	2:21	2:11	8:21	1:47	0:35	0.15
luindex	1:31	1:31	0:28	0:08	0:34	0.09
lusearch	1:20	1:20	0:21	0:54	0:35	0.06
pmd	2:18	2:15	2:54	0:21	0:34	0.11
sunflow	4:19	4:22	0:27	29:17	0:36	0.08
tomcat	15:35	15:32	1:19:25	0:24	0:17	0.71
xalan	2:13	2:15	3:16	4:23	0:33	0.09

5.2 Rank Correlation Analysis

In this experiment, we evaluate the agreement between the estimated and real frequency distributions. Specifically, for each virtual call, we first rank its targets according to their estimated frequencies and dynamic profiles, respectively. Then we measure the correlation between these two ranks by computing their Kendall tau distance [3]. Conceptually Kendall tau distance represents the similarity between two ordered lists by counting the number of swaps needed to reorder one list into the same order with the other. The normalized value of Kendall tau distance lies in the interval [0, 1], where low distance value indicates high agreement. The average normalized value of Kendall tau distance between a list and its random permutation is 0.5. Because in this experiment random permutation corresponds to the uniform frequency distribution, we use 0.5 as the baseline.

[5] Since fop and luindex do not have large workloads, we use default ones instead.

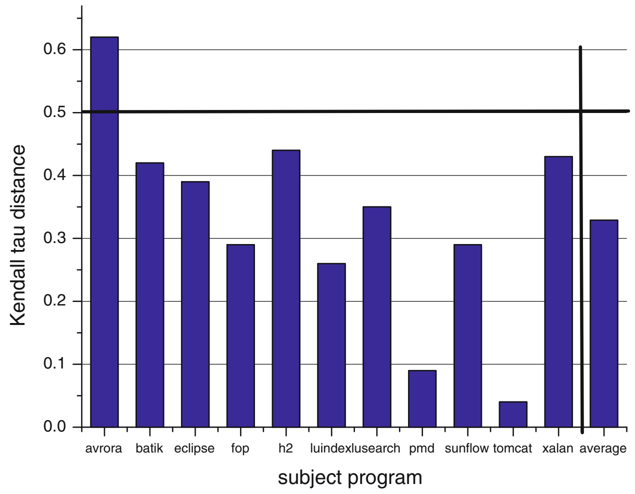

Fig. 6. Kendall tau distance between estimated and real distributions

Figure 6 shows the average normalized Kendall tau distance for virtual calls in each subject program as well as the overall average distance. Except for avrora, all the subject programs have a distance less than 0.5 and the overall average distance is 0.33. It indicates that the estimated frequency distributions can reflect the real distributions much better than the uniform distribution. By inspecting the relevant data of avrora, we found that some static features that can characterize avrora well (i.e., package depth and name similarity) have relatively low predictive power in the estimation model trained using the other subject programs. It might be due to the different design styles between avrora and other subject programs.

5.3 Top Target Prediction

According to the study by Grove et al. [16], one or two "hottest" targets usually take up most of the execution count of a virtual call. Therefore, it is meaningful to evaluate Festival's ability to predict the top targets. As the virtual calls that we study have at most 10 targets, we focus on the top **one** target of each virtual call. The measure is straightforward: for a specific virtual call, if the estimated top target actually has the largest execution count in the dynamic profile, we score the prediction as 1; otherwise the score is 0. Figures 7 and 8 show the average scores of top target prediction based on Festival and the uniform distribution, where the scores are categorized by the number of targets and the subject program, respectively.

As shown in Figure 7, we get a mixed result in top target prediction: for some target numbers, Festival significantly outperforms uniform estimation, whereas it has much worse performance for others. It is difficult for Festival to constantly

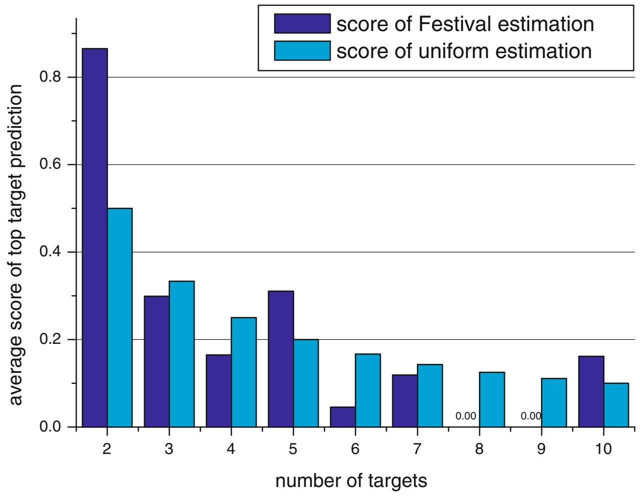

Fig. 7. Average scores of top target prediction categorized by number of targets

predict the top one target, especially when the number of targets is relatively large. However, as shown in Table 3, over 80% of the virtual calls (used for evaluation) have two targets. Therefore, the average score of 0.87 for two-target virtual calls can be viewed as more important than the other scores.

Figure 8 shows Festival's performance for top target prediction from another point of view. When the scores are averaged within each subject program, Festival mostly outperforms uniform estimation. To be more detailed, for each subject program, the average scores of Festival and uniform estimation are calculated by $\frac{S_{total}}{N_{vc}}$ and $\frac{1}{A_{tgt}}$, respectively, where S_{total} stands for the total score of all virtual calls, N_{vc} stands for the number of virtual calls, and A_{tgt} stands for the average number of targets for each virtual call. Conceptually, $\frac{S_{total}}{N_{vc}}$ represents the likelihood for Festival to score 1 for each virtual call, and $\frac{1}{A_{tgt}}$ represents the likelihood to randomly predict the top target of a virtual call. Therefore, it is reasonable to compare them with each other. As for avrora, Festival does not perform well, which is probably due to the same reason as discussed in Section 5.2.

5.4 Weight Matching Analysis

Besides the order of virtual call targets, the quantity of the estimated frequency may also be useful in some quantitative analyses. In this experiment, we use weight matching score [23] to measure the estimation performance of Festival in this aspect. For example, Table 5 shows five virtual call targets, along with their (normalized) estimated and real profiles. The two target lists are ordered by the estimated and real profiles, respectively.

In computing the weight matching score, a cut-off n is specified at first. Then we calculate the sum of **real profile values** for the top n targets in the

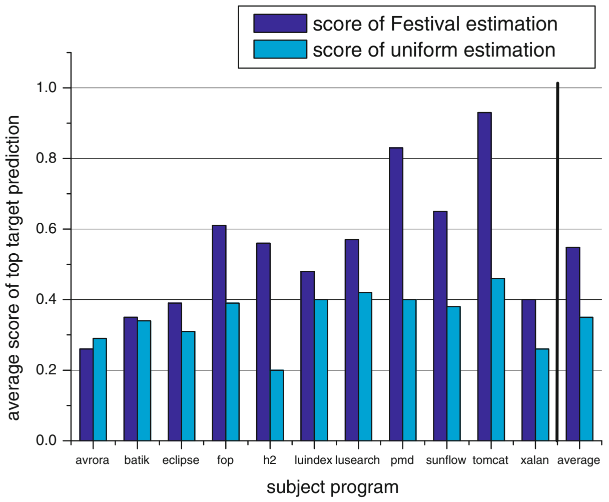

Fig. 8. Average scores of top target prediction categorized by subject program

estimated list as well as the sum for the top n in the **real** list (noted as Sum_e and Sum_r, respectively). The weight matching score is the ratio Sum_e/Sum_r. The perfect estimation has a score of 1, and the closer to 1, the better the estimation is. In the example, for $n = 2$, the weight matching score is 0.85/0.95.

Table 5. An example for computing weight matching score

Estimated	Target	Real	Target
0.50	A	0.80	A
0.20	B	0.15	C
0.10	C	0.05	B
0.10	D	0.00	D
0.10	E	0.00	E

In this experiment, we calculate the scores with the cut-off $n = 1$. As shown in Figure 9, the average weight matching score of Festival is about 59%. For the subject programs, the top target averagely takes up 91% of the execution count in terms of the dynamic profiles. Thus Festival assigns more than a half (i.e., 54%) of the execution count to the estimated top target in average.

5.5 Predictive Power Analysis

This experiment is designed for investigating the relative predictive power of each static feature. To this end, we first build the estimation model based on each single feature instead of the whole feature set. Then we compute the Kendall tau

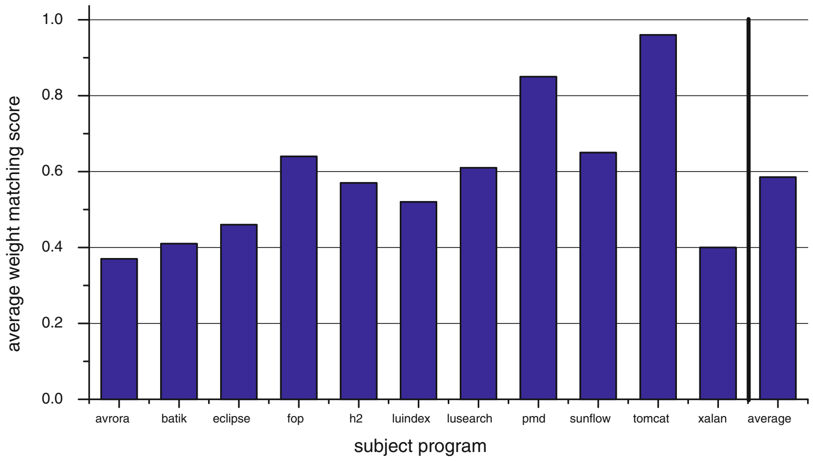

Fig. 9. Average weight matching scores

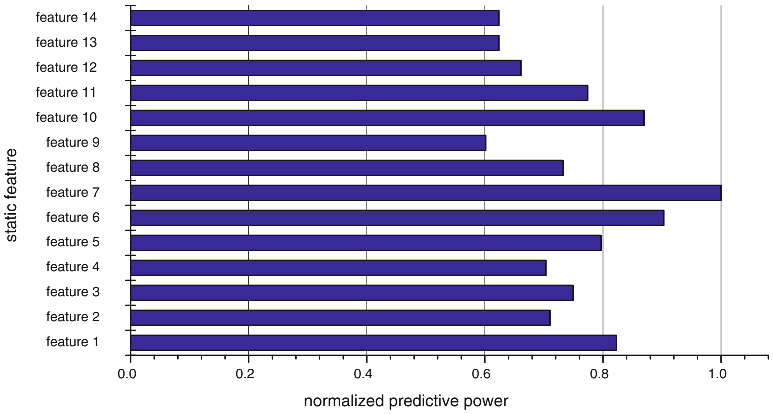

Fig. 10. Predictive power of static features

distance, in the same way as the first experiment, to measure the performance of the model, which in turn represents the predictive power of the feature.

Figure 10 shows the results, in which we use the normalized reciprocal of the Kendall tau distance to represent the predictive power. Thus the features that have larger values are more predictive than the others. In the experiment, the most predictive two features are both call graph-based, while the five features based on type hierarchy are seemingly less useful. However, as discussed in Section 2, feature 6 is closely related to features 2 and 3. Thus the correlations between call graph and type hierarchy may have large impact on the estimation. Similar to those based on type hierarchy, the complexity-based features have moderate predictive power. In contrast, the surface features of target class generally have the lowest power. They might be too simple to capture sufficient design intentions. For naming style features, the depth of package is more informative than the name similarity. However, by analyzing the source code, we have found that the latter may become more predictive, if we have a better algorithm for computing name similarity. We are planning to improve this feature in our future work.

5.6 Discussion

The first three experiments have evaluated the estimation performance of Festival with respect to different measures. The experimental results give satisfactory answers to the research questions $RQ1$ and $RQ2$. In addition, the analysis based on the fourth experiment presents an answer to $RQ3$. In summary, Festival can indeed provide useful information of execution frequency for virtual call targets.

Currently Festival has a limitation that it cannot provide useful estimation for virtual calls that have too many targets. A typical case is the *visitor* design pattern [15], which usually involves complex type hierarchies. For example, the eclipse JDT compiler API uses visitor pattern to process AST. Consequently, the class `ASTNode` has more than 90 subclasses, most of which have implemented the `accept` method. It is really difficult to estimate frequencies for so many implementations of the `accept` method. Another difficulty may stem from the fact that such kind of type hierarchies usually indicate complicated design intentions that can hardly be captured by our current static features.

5.7 Threats to Validity

One threat to the validity of the experiments is that we evaluate the accuracy of our approach by comparing the estimated frequencies with dynamic profiles. If the workloads are not representative, the comparison may lead to skewed results. To alleviate this problem, we choose subject programs from the DaCapo benchmark suite. Another threat is overfitting which means the machine learning model fits too well to the training data and has poor predictive performance on unseen test data. In the experiments we always leave out the program to be estimated and train the model using all the other programs. We believe this kind of cross-validation can avoid the threat of overfitting. Finally, the static features we use may not comprehensive enough to capture all the valuable information. In fact we have studied over 20 features and selected the most informative 14 of them to build the estimation model.

6 Related Work

Machine learning is a powerful tool for predicting program behaviors. Calder et al. [11] proposed a branch prediction technique using decision trees and neural nets and coined the term *evidence-based static prediction* or *ESP*. Our approach has a similar architecture to their work. However, while their technique tackles the problem of branch prediction for C and Fortran programs, our work investigates the frequency estimation of virtual call targets which are specific to object-oriented programs. Furthermore, the static features used in their approach are mostly based on characteristics of instructions and control flows. In contrast, Festival focuses on features at higher levels, since it tries to reveal design intentions. Buse and Weimer [10] recently introduced a machine learning-based approach to estimation of execution frequency for program paths. Inspired by this work, we choose our static features to capture design intentions. Different from Festival, their approach is focused on program paths rather than virtual calls. Moreover, their main idea is to perform estimation based on state change patterns. It is different from our idea, which is mainly about the specialty and popularity of target methods and classes.

Dynamic profile-guided techniques are widely used to predict program behaviors to support code optimizations. Grove et al. [16] developed the call chain profile model to describe profile information at various granularities. In their work, they have performed a detailed study on the predictability of receiver class distributions which shows that the distributions are strongly peaked and stable across both inputs and program versions. Although taking a different prediction approach, our work is largely motivated by the results of the study. Virtual method calls and switch statements are usually implemented by indirect jumps at the instruction level. Li and John [21] explored the control flow transfer behaviors of Java runtime systems. Besides other observations, they found that most of the dynamic indirect branches are multi-target virtual calls and a few target addresses have very high frequencies, which confirms the results of the study by Grove et al. Other dynamic techniques [14] [19] have been proposed to improve the prediction accuracy and reduce the misprediction penalty for indirect branches. Compared with these dynamic techniques, Festival is relatively lightweight in that it only requires surface level instrumentation and program-based features. No instruction level manipulation or hardware extension is needed. In addition, the estimation phase of Festival is purely static and does not rely on representative workloads.

Besides dynamic profiling, static techniques have also been proposed to estimate frequencies of various program elements. Wall [23] conducted a comprehensive study on how well real (dynamic) and estimated (static) profiles can predict program behaviors. Based on the study, Wall argued that real profiles are usually better than estimated profiles. However, he also warned about the representativeness of real profiles. Focusing on non-loop two-way branches, Ball and Larus [6] proposed several heuristics to perform program-based branch prediction for programs written in C and Fortran. Based on these heuristics, Wu and Larus [25] designed a group of algorithms to statically calculate the relative frequencies of program elements. They use Dempster-Shafer technique to combine basic heuristics into stronger predictors. To address the similar issue, Wagner et al. [22] independently developed a static estimation technique. They used Markov model to perform inter-procedural estimations. Similar to our approach, these static approaches aim at the problems which are not amenable to dynamic techniques. Therefore, their motivations also greatly motivate our work. However, we focus on virtual calls in object-oriented programs that have not been studied by the existing work.

7 Conclusions and Future Work

In this paper we have described the Festival approach to frequency estimation for virtual call targets in object-oriented programs. Using static feature data and dynamic profiles of selected programs, we train a multilayer perceptron model and use it to perform estimation for new programs. The evaluation shows that Festival can provide estimations which are much more accurate than estimations based on the uniform frequency distribution. It means that the approach can be useful to a number of applications.

In our future work, we are planning to investigate more static features from other aspects (e.g., control-flow graph structure) and figure out how to combine probabilistic points-to analysis with Festival in order to make them benefit from each other.

Acknowledgments. We are grateful to Qingzhou Luo for his discussion and to Eric Bodden for his suggestions on using Soot and TamiFlex. The presented work was supported in part by National Natural Science Foundation of China (NSFC) (Grants No.60673120 and No. 60970009).

References

1. ANTLR Parser Generator, `http://www.antlr.org/`
2. CLOC – Count Lines of Code, `http://cloc.sourceforge.net/`
3. Kendall tau distance, `http://en.wikipedia.org/wiki/Kendall_tau_distance`
4. Soot: a Java Optimization Framework, `http://www.sable.mcgill.ca/soot/`
5. Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. In: ISSTA 2008: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 189–200. ACM, New York (2008)
6. Ball, T., Larus, J.R.: Branch prediction for free. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 300–313. ACM, New York (1993)
7. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 169–190. ACM, New York (2006)
8. Bodden, E., Sewe, A., Sinschek, J., Mezini, M.: Taming Reflection (Extended version). Technical Report TUD-CS-2010-0066, CASED (March 2010), `http://cased.de/`

9. Boogerd, C., Moonen, L.: Prioritizing software inspection results using static profiling. In: Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 149–160 (2006)
10. Buse, R.P.L., Weimer, W.: The road not taken: Estimating path execution frequency statically. In: ICSE 2009: Proceedings of the 31st International Conference on Software Engineering, pp. 144–154. IEEE Computer Society, Washington, DC, USA (2009)
11. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based static branch prediction using machine learning. ACM Trans. Program. Lang. Syst. 19(1), 188–222 (1997)
12. Chambers, C., Dean, J., Grove, D.: Whole-program optimization of object-oriented languages. Technical report, Department of Computer Science and Engineering, University of Washington (1996)
13. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
14. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, pp. 278–288. ACM, New York (2003)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
16. Grove, D., Dean, J., Garrett, C., Chambers, C.: Profile-guided receiver class prediction. SIGPLAN Not 30(10), 108–123 (1995)
17. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not 39(12), 92–106 (2004)
18. Hwang, Y.-S., Chen, P.-S., Lee, J.K., Ju, R.D.-C.: Probabilistic points-to analysis. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 290–305. Springer, Heidelberg (2004)
19. Joao, J.A., Mutlu, O., Kim, H., Agarwal, R., Patt, Y.N.: Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 80–90. ACM, New York (2008)
20. Lhoták, O.: Spark: A flexible points-to analysis framework for Java. Master's Thesis, McGill University (2002)
21. Li, T., John, L.K.: Understanding control flow transfer and its predictability in Java processing. In: IEEE International Symposium on Performance Analysis of Systems and Software, pp. 65–76 (2001)
22. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate static estimators for program optimization. In: PLDI 1994: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 85–96. ACM, New York (1994)
23. Wall, D.W.: Predicting program behavior using real or estimated profiles. SIGPLAN Not 26(6), 59–70 (1991)
24. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)
25. Wu, Y., Larus, J.R.: Static branch frequency and program profile analysis. In: MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 1–11. ACM, New York (1994)