Call Sequence Prediction through Probabilistic Calling Automata

Zhijia Zhao, Bo Wu[◇], Mingzhou Zhou, Yufei Ding[†], Jianhua Sun, Xipeng Shen[†], Youfeng Wu[*]
College of William and Mary [◇] Colorado School of Mines [†] North Carolina State University [*]Intel Labs
{zzhao,mzhou,jianhua}@cs.wm.edu [◇]bwu@mines.edu [†]{yding8,xshen5}@ncsu.edu [*]youfeng.wu@intel.com

Abstract

Predicting a sequence of upcoming function calls is important for optimizing programs written in modern managed languages (e.g., Java, Javascript, C#.) Existing function call predictions are mainly built on statistical patterns, suitable for predicting a single call but not a sequence of calls. This paper presents a new way to enable call sequence prediction, which exploits program structures through Probabilistic Calling Automata (PCA), a new program representation that captures both the inherent ensuing relations among function calls, and the probabilistic nature of execution paths. It shows that PCA-based prediction outperforms existing predictions, yielding substantial speedup when being applied to guide Just-In-Time compilation. By enabling accurate, efficient call sequence prediction for the first time, PCA-based predictors open up many new opportunities for dynamic program optimizations.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords Function call, Call sequence prediction, Probabilistic calling automata, Dynamic optimizations, Just-in-time compilation, Parallel compilation

1. Introduction

Languages with a managed environment—such as JAVA, Javascript, C#—become increasingly popular. Programs in these languages often have a large number of functions, and feature many dynamic properties. For them, knowing the upcoming sequence of function calls in a run can be helpful. For example, a feature in these languages is dynamic function loading: Some classes or functions are loaded from local disks or remote servers during an execution [25]. The loading takes time. With the upcoming call sequence known, the delay can be largely hidden through prefetching. As another example, the runtime system supporting those languages, especially on embedded systems, often uses a small chunk of memory (called code cache) to store the generated native code for reuse. Knowing the upcoming call sequence can enhance the code cache usage substantially [16]. It can also help the runtime system decide when to invoke JIT to compile which function and at which optimization levels [15], and so on. The benefits may go beyond the runtime of managed languages. Co-design virtual machines [18], for instance, use runtime Binary Code Translation to reconcile disparity between conventional ISA and native ISA. Its runtime translation also uses JIT, sharing similar opportunities.

Call sequence prediction is to provide such knowledge through prediction. It is challenging for the large scope of prediction. The state of the art is yet preliminary. Most of them have concentrated on exploiting statistical patterns in call history [4, 23, 27], and predicting the next one call rather than a sequence of calls. This limited prediction scope does not well suit the many needs of runtime systems. Even worse, as the scope enlarges, the regularity diminishes, forming a main barrier for existing prediction techniques.

In this paper, we present a new way to enable call sequence prediction. It centers on an effective exploitation of program structures. The rationale is that *program structures inherently define some constraints on function calling relations, which often cast some deciding effects on function call sequences*. Conceptually, the key of this approach is in developing an expressive model of the relations among function calls to effectively capture those constraints. To facilitate runtime call sequence prediction, the model must distinguish call sites, capture calling contexts, incorporate the influence of branches and loops, and finally accommodate the various complexities in programs and language implementations (e.g., function dynamic dispatch, function inlining, code coverage variations across inputs). Existing models—such as call graphs, call trees, and calling context trees [3]—meet some but not all these requirements.

We present *Probabilistic Calling Automata* (PCA), a new program representation that uses extended Deterministic Finite Automata (DFA) to capture both the inherent ensuing

OOPSLA '14, October 19 - 21 2014, Portland, OR, USA

ACM 978-1-4503-2585-1/14/10...$15.00.
http://dx.doi.org/10.1145/2660193.2660221

relations among functions, and the probabilistic nature of execution paths caused by branches, loops, and dynamic dispatch. A PCA is composed of a number of augmented state machines, with each encoding the control flows related function calls in a function. The PCA features a return stack and a shadow stack for efficiently maintain calling contexts, an α-stack to handle complexities brought by exceptions and unknown function calls, and the concept of v-nodes and candidate tables for addressing calling ambiguities caused by polymorphism, function pointers, and dynamic dispatch. Serving as a unified representation for function calls, PCA incorporates static program structures with profiling information, supports easy runtime state tracking, and tolerates various complexities in practical deployment.

After presenting the definition, properties, construction and usage of PCA in Section 2 and Section 3, we discuss the insufficiencies of existing program representations in Section 4, introduce some metrics for call sequence prediction in Section 5, and then describe an empirical comparison between PCA-based predictors and the extensions of three alternative methods, respectively based on Calling Context Trees and statistical patterns. Experiments show that PCA-based predictor achieves 89% on average in a basic accuracy metric, 20–50% higher than that of the other predictors. Through parallel JIT compilation, we demonstrate that a simple usage of the PCA-based prediction can lead to performance improvement by up to 32% (15% on average).

Overall, this work makes the following contributions:

- It introduces PCA, a novel representation of function ensuing relations in a program that captures the influence cast by control flows and calling contexts.
- It shows how PCA can be used to enable effective call sequence prediction with design choices and usage study, as well as a systematic comparison with alternatives.
- It provides a set of metrics for measuring the quality of a call sequence prediction at various levels. They may meet the needs of different uses of the prediction.
- Finally, this work, for the first time, demonstrates the feasibility and benefit of accurate call sequence prediction, which opens up new opportunities for dynamic optimizations in various layers of the execution stack.

2. Problem Definition and Design Considerations

As the problem has not been systematically explored before, we first provide a formal definition as follows.

2.1 Definition of Call Sequence Prediction

Definition 1. *A* **function call sequence** *of a program is a sequence of the IDs of the functions in the order of their invocations in an execution time window.*

Definition 2. Call Sequence Prediction: *For a given execution of program P, function call sequence prediction at a time point t is to predict the function call sequence of P in the time window that immediately follows t.*

The time window is called *prediction window*. Its length is usually in logical time (e.g., the number of function calls), and may be fixed or vary. Depending on the windows' length, the prediction may happen many times during an execution of a program. For a multithreading execution, the prediction can be at the whole program level including all threads, or at the level of one or several specific threads.

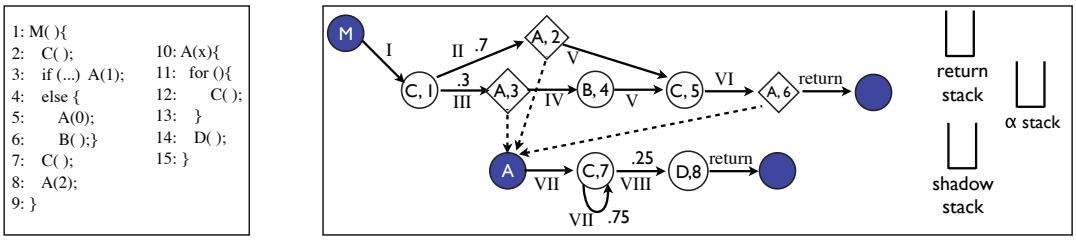
Function call sequences are largely dictated by program structures. A primary goal of this work is to examine how to leverage program structures for call sequence prediction. Conceptually, the problem is to develop a representation that is effective in capturing the relevant constraints on call sequences coded in a program.

2.2 Design Considerations

Ensuing Relations vs. Calling Relations There are some commonly used inter-procedural program representations, such as call graphs, call trees and calling context trees. They primarily represent *calling relations* among functions. But a call sequence is about which call follows which—and hence an embodiment of a series of *ensuing relations*.

It is important to note the differences between these two kinds of relations. Calling relations are about what functions would be called **by** what functions, while ensuing relations are about what function would be called **right after** what function calls. Calling relations affect ensuing relations. Knowing Y as one of the callees of X, for instance, suggests that Y will be, with some uncertainty in the presence of branches, called after a call to X. But when that call will happen is not coded in the calling relation: It could be immediate, several or hundreds of calls later. An example is the call of "D()" by "A()" in the *PCAExample* code in Figure 1 (a). For the loop before "D", there could be zero or thousands of "C" being called before "D" is called. Conversely, two adjacent function calls in a call sequence, say "V W", does not entail that W must be called by V: That call to W could be made by V, its caller, or any of its calling ancestors— respectively illustrated by "A C" in lines 10 and 12, "C D" in lines 12 and 14, and "D B" in lines 14 and 6 in the PCAExample.

Four Basic Properties to Consider So the first consideration in our design of program representation is that it must capture ensuing relations of function calls. Ensuing relations naturally relate with program control flows (e.g., branches, loops), and often differ from one call site to another and from one calling context to another. So the representation should also consider these factors. In addition, to be used in runtime call sequence prediction, the representation should be reasonable in size, resilient to program complexities, and applicable to most executions of a program. We put these considerations together as follows, and call them the *four basic properties*:



(a) "PCAExample" program (b) PCA

Figure 1. An example program named "PCAExample" and its PCA.

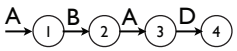


Figure 2. A DFA for code "`A(); B(); A(); D();`".

- *Ensuing relations*: capturing ensuing relations among function calls;
- *Discriminating:* discerning different control flows, call sites, and calling contexts[1];
- *Generality:* being resilient to program complexities, such as ambiguous calling targets in the presence of virtual functions, function dynamic dispatch, and so on;
- *Scalability:* having a bounded and acceptable size, regardless of the program execution length.

Existing representations were designed mainly for program analysis rather than call sequence prediction. They center around calling relations, and fall short in some of the four basic properties (detailed in Section 4). Because of their insufficiency, we propose the following design of PCA.

3. Probabilistic Calling Automaton (PCA)

Intuition PCA is in an augmented form of finite state automata. Before defining it formally (in Section 3.6), we first offer some intuitive explanations. We choose automata as the basic form for their natural fit for expressing ensuing relations. For instance, Figure 2 shows an automaton for code "A( ); B( ); A( ); D( );". The four nodes represent four stages of an execution of the code. The DFA can easily track the execution through state transitions: Upon the first call of the function "A", it moves to state 1, and then to state 2 after "B" is invoked, and so on. The structure of the DFA reflects the ensuing relations of function calls imposed by the control flow in the function—in Figure 2, a constraint is that "A D" but not any other sequences immediately follows the call of "B". With this DFA, call sequence prediction becomes a simple walk over the DFA. For instance, suppose the DFA is now at node 1. To predict the remaining call sequence, we can simply walk along the DFA from node 1 and output the functions on the edges we bypass ("B A D" in this example.)

[1] Here, calling context refers to the sequence of functions on the current call stack. A more precise context also includes parameter values, which further complicates the problem. It is out of the scope of this work.

This example is simple, but conveys the basic idea of PCA: Incorporating constraints defined by program code into a finite automaton and converting call sequence prediction into a walk over the automaton. For the idea to work, there are many challenges, some from program structures (e.g., branches, loops), some from language implementations (e.g., function dynamic dispatching), some from compiler transformations (e.g., function inlining and outlining). PCA addresses these challenges through a careful design.

To make the explanation easy to follow, we first draw on an example (*PCAExample*) rather than formalism to explain our PCA design and how it addresses various complexities for runtime call sequence prediction. After that, we provide a formal, rigorous definition of PCA, along with the algorithm to construct it automatically.

3.1 Structure of PCA

A PCA consists of a number of finite state automata, and three types of stacks. There is one automaton for each non-leaf function in the program. (A function is a *leaf* function if its code contains no function calls; it is *non-leaf* otherwise.)

Nodes Each node in a PCA automaton corresponds to one call site in the function. If the invoked function is non-leaf, we call the node a *diamond*, otherwise, a *circle*. Each node carries a label, written as ⟨FunctionID, CallSiteID⟩, where "FunctionID" and "CallSiteID" are the ID of the function invoked at that call site and the ID of the call site itself. (A unique ID is assigned to every function and every call site.) A diamond carries an extra field, recording the address of the entry point of the automaton of the function called at the call site represented by the diamond. This field allows smooth transitions among automata of different functions.

When the "FunctionID" at a call site is either non-unique or unknown at the PCA construction time, "*" is used for that field of the node. Such a node is called a *v-node* (v stands for virtual function). A *v-node* can be either a diamond or circle. The abstraction of *v-nodes* is important for treating ambiguous function calls as Section 3.4 will show.

An automaton has a single entry node, and a single terminal node. They correspond to no call site, just indicating the entry and exit points of the automaton respectively.

Edges Edges in a PCA represent the ensuing relations among the function call sites contained in a function. There

is a directed edge from node A to node B in an automaton if after A's call site (i.e., the call site represented by node A) is reached in an execution, node B's call site could be reached before any other call site in that automaton is reached. Note that some call sites in other automata could be reached between them. An example is the call of "A" on line 5 and the call of "B" on line 6 in Figure 1 (a). The latter immediately follows the former and hence there should be an edge between their nodes, despite that the callees of function "A" are reached between the calls to "A" and "B".

Each edge carries a label and a weight. The label is the ID of the sink node's call site. It gives conveniences to tracking program state transitions in a call site discriminative manner as we will see later. The weight is the probability for the sink to follow the source in the program's executions. An edge flowing into a terminal node can have only "return" or "exit" as the label, indicating the exit of the function.

Stacks There are three stacks, associated with the entire PCA of a program. They are the *return stack*, *shadow stack*, and α*-stack*. The first two are designed to provide discrimination of calling contexts, explained in Section 3.2. The third helps handle unexpected function calls for practical deployment of PCA, explained in Section 3.4.

Example Figure 1 (b) shows the PCA of the PCAExample code in Figure 1 (a). The top part shows the automaton of function "M". It contains three diamonds, all representing calls of the non-leaf function "A" at the bottom. The three dotted lines are not PCA edges, but illustrations of the three diamonds' references to the entry of A's automaton. Entry and terminal nodes are shown as disks. Each node has its ID labeled. For instance, the diamond on top has a label "A,2", meaning that this call site ID is "2" and the call is to function "A". The edge from node "C,1" to "A,2" has label II as it is the ID number of the call site represented by the sink node. Its weight ".7" indicates that 70% instances of the call site 1 are immediately followed by a call made at call site 2 in function "M". Weights equaling 1 are not shown for readability. Theoretically speaking, the call site ID needs to be labeled only on either the edge or in the sink node. Having the label at both places is for conveniences.

3.2 Basic Usage for Tracking and Prediction

The design of PCA makes it handy for efficient tracking the state of a program execution and predicting its upcoming function calls.

Tracking Execution State To track the execution state of a program through PCA, we just need to let PCA transit to its next state upon every function call in an execution. An exit or return prompts a transit to its terminal state. When reaching a diamond node, the transition immediately moves to the entry node of the corresponding automaton. For instance, a call at line 3 of PCAExample makes the PCA move from state "C,1" to "A,2", and then immediately to the entry node of the automaton of function "A". State transitions when a PCA reaches a terminal state are facilitated by the *return stack*. When the PCA transits from a diamond node to the entry of another PCA, the address of the diamond node is pushed into the *return stack*. When the PCA reaches the terminal state, it pops the diamond node address out of the *return stack* and jumps to that address immediately.

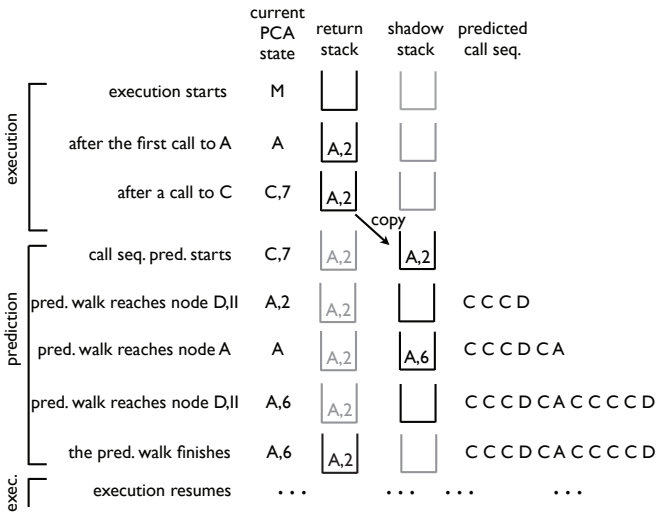


Figure 3. Illustration of how the PCA in Figure 1 (b) supports call sequence prediction. It assumes that the prediction starts after execution sees the call sequence "M C A C" and the goal is to predict all remaining function calls.

The runtime tracking requires some code to be instrumented at function call sites. To minimize the overhead, the inserted code only puts the ID of the function and call site (or a predefined numerical ID for "return") into a buffer. At the beginning of a prediction, the buffer is consumed to bring the PCA status to date.

Predicting Call Sequences Call sequence prediction by PCA is a quick walk over the PCA while outputting the IDs of the functions in the passed nodes. The *return* and *shadow* stacks make it possible for the prediction to discriminate call sites and calling contexts. When a sequence prediction starts, the shadow stack gets a copy of the content of the return stack to attain the current program state to work with. When a sequence prediction finishes, the shadow stack is emptied.

Figure 3 illustrates how the PCA in Figure 1 (b) supports the prediction process of PCAExample. The gray color indicates the time when the stacks are inactive. When the program starts, both stacks are empty and the PCA is at the starting state, state M. After line 3 (C and A are called), the PCA moves to state "A" and the diamond node "A,2" is pushed into the return stack. After a call to C on line 12, the PCA gets to state "C,7". It is assumed that the runtime now starts a call sequence prediction. The shadow stack attains a copy of the content of the return stack and becomes active, while the return stack pauses its operations. The predictor starts walking on the PCA from the current state, state "C,7", which has two outgoing edges. Suppose that the predictor takes the backedge (which carries a call to "C") three times before

taking the edge (carrying a call to "D") towards node "D,8". That walk yields the predicted sequence "C C C D". As node "D,8" leads to a terminal node, the shadow stack pops out node "A,2" and the prediction walk immediately jumps to that node. It is assumed that the walk then takes the edge to node "C,5" and then to node "A,6" and outputs "C A" as the prediction. It then gets to node "A" again and continues the prediction. When the prediction finishes, the shadow stack is emptied and the return stack becomes active again.

The example touches one type of ambiguity in PCA: A node has more than one outgoing edge, as exemplified by nodes "C,1" and "C,7". We call this *edge ambiguity*. Edge weights provide probabilistic clues on resolving the ambiguity. We experiment with two policies for exploiting the hints. The first is the *maximum likelihood (ML) approach*, which always selects the edge with the largest weight. The second is *random walk*, which chooses an edge with a probability equaling the weight of that edge. For a node with k outgoing edges, the approach works like throwing a k-sided biased dice, the biases of which equal the edge weights.

The ML approach seems to be subject to loops: A backedge with a high probability may trap the predictor into the loop[2]. However, when using PCA for call sequence prediction, the runtime queries the PCA occasionally. Hence, even though PCA might predict a seemingly-infinite loop, continued execution of the real program generally results in escaping the loop. A subsequent PCA query would then ask about execution following the loop. In practice, it outperforms random walk in most cases as Section 6 will show.

3.3 Challenges for Practical Deployment

The aforementioned basic usage of PCA for prediction has two implicit assumptions:

1. *Known-ID condition:* The PCA construction can completely determine the ID of the function to be invoked at every call site.
2. *Completeness condition:* The PCA captures all possible and correct ensuing relations among function calls of a program.

The two conditions ensure that all call sequences occurring in an execution would be expected (and hence processable) by the PCA. However, in many practical cases, the two conditions do not hold due to the complexities in language implementation, compiler optimizations, and PCA construction process. We will base our discussion mainly on a managed programming language (e.g., Java). Other types of languages (e.g., C/C++) share some of those complexities.

Function Dynamic Dispatch The *known-ID condition* does not always hold in the presence of function dynamic dispatch, with which feature, what function is called at a call site may remain unknown until the call actually happens. It often relates with polymorphism. For instance, suppose Cat, Dog, and Sheep are all subclasses of Animal, and they all have their own implementation of the virtual function "voice()" in Animal. The call to "a[i].voice()" at line 3 in Figure 4 may actually invoke the "voice()" function of any of the three classes, depending on which subclass $a[i]$ is. Another common cause of dynamic dispatch is function pointers, whose values may not be precisely determined at compile time in a C program. No matter what the implementation is, a common property of dynamic dispatch is that the exact function to be invoked at a call site sometimes cannot be determined until the call happens. As an analogy to the *edge ambiguity* mentioned earlier, this issue can be regarded as *node ambiguity*. It is embodied by v-nodes in a PCA, the labels of which have "*" as the FunctionID.

```
/* a is an array of Animal that has a virtual function "voice()"; class
Animal has subclasses Cat, Dog, and Sheep.*/

1: for (i=0;i<N;i++){
2:     F();
3:     a[i].voice();
4:     G();
5: }
```

Figure 4. An example of dynamic dispatch for polymorphism

Compilation Complexities As Section 3.6 will show, PCA construction usually happens through some training runs with the help of compilers. In a managed environment, the compilation is through a JIT compiler, and typically happens in every run. The compilation may differ in different runs, causing different ensuing relations among function calls, and hence the violation of the *completeness condition*. For instance, function inlining replaces a call site with the code of the callee, while function outlining forms new functions in the binary code. So different inlining and outlining decisions in different runs could lead to different sets of call sites and ensuing relations.

Furthermore, training runs and production runs may have a different coverage of the code. Some functions invoked in a production run may have never been encountered by the JIT compiler in training runs, and hence may not appear in the constructed PCA. The training process could aggressively apply JIT to all possibly invoked functions, no matter whether they are invoked in the training runs. However, due to ambiguity in calling targets, it could end up including too many irrelevant functions (e.g., an entire library excluding library calls to JNI, which are not JITed).

Exception Handling Exception handling causes violations to both conditions. In Java, it is usually implemented with static exception tables, which, similar to function pointers, cause fuzziness in function calling targets. At the same time, some exceptions (e.g., division by zero) are not checked. Similar to signal handlers in C code, there may be no explicit calls to those handlers in the code, forming violations to the *completeness condition*.

[2] If the edge weights get appropriately updated across iterations, ML may not face such a problem.

Moreover, sometimes users may not be concerned of all functions. They, for instance, may not be interested in the invocations of functions in the Java Runtime but only those in the application. The bottom line is that some kind of resilience to the incompleteness of PCA and node ambiguity would be necessary for a practical deployment of PCA.

3.4 Solutions through v-Node and α-Stack

Features of v-nodes help address the issues related to the *known-ID condition*. Each v-node is equipped with a *candidates table*. Every entry in the table indicates the possibility for that call site to be an invocation of a particular function. A threshold K is used to control the size of the table. Only the top K most likely candidates appear in the table. Figure 5 shows the PCA for a variant of the "M" function in our PCAExample, in which, the call to "A" at line 3 is replaced with a function pointer whose most likely calling targets are functions "A" and "B". Besides them, there is another 15% chance for the target to be some other functions. The probabilities of candidate targets are obtained through offline profiling, but adjustable at runtime as explained later.

During sequence prediction, the candidate table is used for speculating on the ID of the function to be invoked at the corresponding call site. The speculation employs the same methods as in resolving edge ambiguity (i.e., the ML or random walk method). The speculation happens every time when the prediction-oriented PCA walk reaches an v-node.

The issues on *completeness condition* are addressed through a combination of dynamic PCA evolvement and α-stack. The dynamic evolvement is done at JIT time. Our examination shows that function inlining and outlining are the major reasons for violations of the *completeness condition*. The dynamic adjustment for inlining and outlining is straightforward. Upon a function inlining, the JIT replaces the node of that call site with the automaton of the inlined function; upon a function outlining, the JIT creates an automaton for the newly formed PCA, assigns an ID to the new call site, and updates the automaton of the parent PCA accordingly. As outlining happens rarely, negligible overhead was seen on the runtime PCA construction. The edge weights of the newly created PCA are initiated with some values determined by the compiler (e.g., a policy common in compiler construction is to put 0.9 for backedges and 0.5 for normal two-way branches [36]).

As an option, during runtime, edge weights can be refined with the runtime observations through weighted average (i.e., new weight = old weight*r + new observations*(1-r), $0.5 > r > 0$) with the decay rate r set by the user. Such an adjustment can be applied to other existing edges as well. (Our experiments did not use this runtime refinement.)

The α-stack addresses the issue of incomplete PCA (i.e. some functions do not have automata built). Initially the α-stack is empty and inactive. At an invocation of a function that has no automata built, the ID of the function is pushed into the α-stack, and the α-stack becomes active. While the α-stack is active, the ID of an invoked function is automatically pushed into the stack, regardless of whether the function has PCA; the top of the stack pops out at each function return. The PCA stalls while the α-stack is active. It resumes state transitions as soon as the α-stack becomes inactive when it turns empty. For example, suppose that the PCA is now in state "C,1" of Figure 1 (b) and some unexpected function "X" is then invoked. Assume that "X" calls "Y" and "Y" calls "A". Neither "X" nor "Y" has automaton built. The PCA would stay at node "C,1" until "X" returns. It then resumes state transition according to the PCA.

Essentially, the α-stack makes operations on PCA skip functions that do not have automata, as well as the functions directly or indirectly invoked by them. Such a design offers a simple way to deal with unexpected calls. A more sophisticated design is to skip only functions that have no automata (e.g., "X" and "Y" but not "A" in our example). It is potentially doable, but adds much complexity: It has to deal with broken chains of states. For instance, when "A" returns, it is unclear which state the PCA should return to.

Additional complexities include native function calls and tail call optimizations. Native code is ignored. Optimized tail calls become jump instructions and hence are not tracked or predicted.

3.5 Properties

We now examine how PCA embodies the four basic properties listed in Section 2.

(1) *Ensuing relations.* PCA is centered on ensuing relations. A transition edge represents what function call follows (rather than invokes) another call. For example, in Figure 1 (b), "C,5"→"A,6" represents that after the finish of "C" on line 7 in Figure 1 (a) (represented by node "C,5"), the next function call must be a call to "A" at line 8 (represented by node "A,6"), despite that "C" never calls "A" in the program.

(2) *Discriminating.* The structure of PCA encodes both branches and loops. Its edge weights facilitate the resolution of ambiguities caused by control flows. With node and edge labels carrying call site IDs, PCA naturally distinguishes different call sites. The *return stack* and *shadow stack* add calling contexts to PCA. For example, suppose the PCA is now at state "C,7" in Figure 1 (b). The two stacks help the prediction automatically tell whether the call of "A" was from node "A,2", "A,3", or "A,6" when the PCA walk returns from node "D,8", and hence produce different prediction results. In addition, the PCA structure allows an even deeper level of discrimination: Instead of defining an edge weight as a probability given the source node, one could employ conditional probabilities as edge weights, with the top k levels of the shadow stack as the conditions. In this way, they could further discriminate call sites and calling contexts. Such a model may increase the size of the PCA; we leave it to future study.

(3) *Generality.* The design of v-node and α-stack, along with runtime PCA evolvement, make PCA resilient to various complexities in the language implementation, compilation, and other aspects.

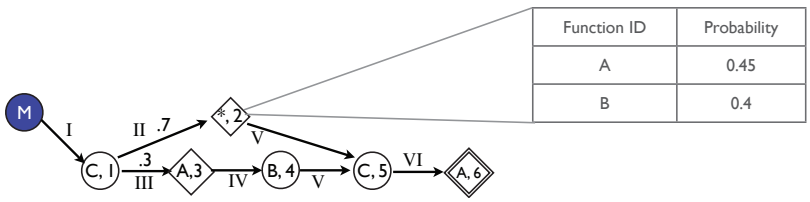


Figure 5. Example PCA with v-nodes Used.

(4) *Scalability.* Unlike some other representations (e.g., dynamic call tree), the size of a PCA is bounded by the number of call sites in a program, independent of the length of an execution. Section 6 reports the size on some programs.

Besides those, the modular structure of PCA gives good compatibility. If the body of a function is changed, except edge weights and that function's automaton, the structure of the program's PCA needs no change.

3.6 Formal Definition of PCA and Its Construction

We now give a formal definition of PCA as follows:

Definition A PCA $\mathcal{P}$ is a tuple $\langle A_1, A_2, \cdots, A_k, \Sigma, \Gamma, \Lambda, r, s, \alpha \rangle$, where:

A_i	: a finite state machine	Σ	: input alphabet
Γ	: stack alphabet	Λ	: exit alphabet
r	: the return stack	s	: the shadow stack
α	: the α-stack		

State machine A_i in a PCA is a tuple $\langle N_i, s_i, f_i, e_i, \delta_i, P, D_i, M_i \rangle$, where:

N_i : the set of nodes.
s_i : a unique entry node. $s_i \in N_i$.
f_i : a unique terminal node. $f_i \in N_i$.
e_i : a unique error node. $e_i \in N_i$.
δ_i : transition relation over N_i.
P : transition probability over δ_i.
D_i : the set of diamonds, $D_i \subset N_i$.
M_i : a mapping function from D_i to s_j, $j \neq i$.

Transition relation δ_i is a finite set of rules such that: ① for every state $S \in (N_i - f_i)$ and an input symbol $a \in \Sigma$, there is a unique rule of the form $\delta_i(S, a) \rightarrow T$, where $T \in (N_i - f_i - s_i)$; ② for any input symbol $a \in \Lambda$, there is at least one state $S \in (N_i - f_i)$ such that $\delta_i(S, a) \rightarrow f_i$. Recursion is allowed; a self-recursive call corresponds to a diamond that carries a reference to the entry node of its own automaton. Transition probability P is a function which to every rule $\delta_i(S, a) \rightarrow T$ assigns its probability $P(\delta_i(S, a) \rightarrow T) \in (0, 1]$ so that for any given $S \in N_i$, we have

$$\sum_{T \in N_i : \exists a, \delta_i(S,a) \rightarrow T} P(\delta_i(S, a) \rightarrow T) = 1.$$

There is a special transition: $d \rightarrow M_i(d), \forall d \in D_i$, which happens every time when d is reached; with the transition, d is pushed into the return stack, r. Another special transition is: $f_i \rightarrow \tau(r)$ (where $\tau(r)$ is the top of the stack r), which happens every time when f_i is reached; meanwhile, r pops its top off the stack. The shadow stack s gets a copy of r when call sequence prediction starts. During and only during the prediction, s plays the role of r in dictating the state transitions. Like many conventional automata, there is an implicit error state e associated with a PCA; $\forall x \in N_i, x \rightarrow e$ on any unexpected input. Along with such a transition, x is pushed into the α-stack. While the PCA is at e, every input belonging to the exit alphabet Λ makes the α-stack pop, while all other inputs are pushed into the α-stack. When the length of the stack becomes two, an encounter of input $a \in \Lambda$ prompts the transition $e \rightarrow \tau(\alpha)$ besides making the α-stack pops. After the transition, the α-stack pops again to turn empty.

Construction The construction of PCA involves two main steps: The first builds up the PCA structure during compilation; the second trains the PCA by adding weights through profiling. Algorithm 1 outlines the procedures.

Algorithm 1 PCA Construction

```
1: /* Building PCA Structure */
2: G[F] = control flow graph of function F;
3: P={};
4: for each function f do
5:     if G[f] contains function calls then
6:         R = buildRegExp(G[f]);
7:         R' = cleanUp(R);
8:         A = regExp2DFA(R');
9:         createCandidateTables(A);
10:        P.add(A);
11:    end if
12: end for
13: connectAutomata(P);
14:
15: addWeights(P); /* Profiling for Weights */
```

For the modularity of PCA, the first step can happen on each function individually. Not all instructions in a function are relevant to function calls. A compiler goes through the code, ignores irrelevant parts, and converts the rest into an automaton. Conceptually, in this step, the compiler derives a skeleton of the control flow graph, where, all statements but function calls and branches are removed, (leaving some empty basic blocks), while the edges remain. The compiler derives a regular expression from the skeleton graph. The

$$
\begin{array}{lcl}
\text{M} & \rightarrow & \text{m}\ C_1\ (A_2|\ (A_3\ B_4))\ C_5\ A_6 \\
\text{A} & \rightarrow & \text{a}\ {C_7}^+\ D_8 \\
\text{B} & \rightarrow & \text{b} \\
\text{C} & \rightarrow & \text{c} \\
\text{D} & \rightarrow & \text{d}
\end{array}
$$

Figure 6. The FCG of the program in Figure 1 (a). Every letter represents a function. An upper-case letter is a non-terminal variable, and a lower-case letter is a terminal variable, representing the prologue of a function represented with the corresponding upper-case letter, and a subscript represents a call site ID.

vocabulary of the regular expression consists of β, representing an empty block in the skeleton graph, a terminal variable and a non-terminal variable for each function in the program. The β helps encode the logic of empty blocks into regular expression. The non-terminal variable represents a call to the function. The terminal variable represents the entry of the function, which is always the first symbol in the regular expression of the function. Branches are represented with the "|" operator, while loops (or backedges) are represented with the "*" or "+" operator.

Next, the compiler simplifies the regular expression in a standard way, which removes all βs as well. Figure 6 shows the regular expressions of our PCAExample. We refer to such a set of regular expressions as the function calling grammar (FCG) of the program. As a whole, an FCG is a Context Free Grammar (CFG). The simple form directly leads to PCA through standard algorithms of regular expression-to-automaton conversion. Another advantage of using FCG as the intermediate form is that the conversion algorithms, by default, minimize the generated automaton and hence the overall size of the PCA. The candidate tables are then built for each v-node in the DFA.

The final step adds weights to the edges in the PCA. It uses profiling executions of the program to do so. During a profiling run, the PCA runs along with it by updating its state upon each function call. A profiler records the number of times an edge is visited if the out-degree of the source is greater than one. The weight of an edge is then used to calculate the weights on those edges. It puts in the probabilities for the entries in candidate tables in the same manner. An edge that has not been encountered in the profiling runs is assigned with an extremely small weight for the completeness of the PCA.

What profiling mechanism to use is orthogonal to the proposal of PCA. Besides offline profiling, there are many other techniques for efficient online sampling [9, 13, 20] or cross-run accumulation of samples [24]. They could all be used for PCA construction, depending on the usage scenario.

4. Comparisons to Existing Representations

Before this work, there are a variety of program representations relevant to program function calls. In this section, we examine four most commonly studied ones, qualitatively showing that they are ill fit for call sequence prediction for not meeting some of the *four basic properties*. Section 6 will complement the comparison with some quantitative evidences.

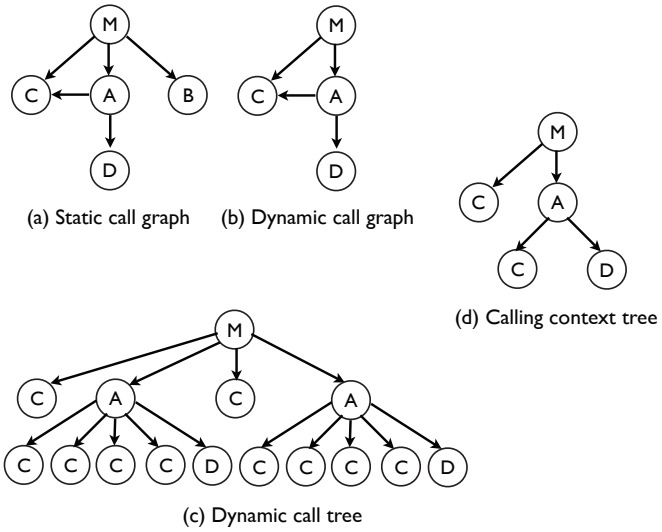


Figure 7. Four other representations of function calls in executions of "PCAExample" in Figure 1 (a). In the program, no functions except "M" and "A" contain function calls. In the considered execution, the "if" branch is taken.

Among existing models of program function calls, the most influential are static and dynamic call graphs, dynamic call trees, and calling context trees (CCTs). We use Figure 7 to review them briefly. In a static call graph (Figure 7 (a)), each function has a unique node no matter at how many call sites it is invoked, and there is an edge directed from function "M" to function "A" if it is possible for "M" to call "A". A dynamic call graph (Figure 7 (b)) has the same structure, except that it is built through a profiling run and there is an edge between two nodes only if that invocation actually takes place in that run. A dynamic call tree (Figure 7 (c)) also comes from a profiling run. It adds calling context information, with each node representing a function invocation, and the path to it from the root representing its calling context. A CCT [3] is similar to a dynamic call tree except that it uses a single node to represent all calls to a function that have the same calling contexts. In Figure 7 (d) for instance, all the "C" nodes under "A" in Figure 7 (c) are folded into one.

All four representations are designed for program analysis rather than call sequence prediction. They have some variations. We analyze their properties with their basic forms first, and discuss their extensions later. Specifically, we examine them against the *four basic properties*, which qualitatively reveals their limitations for call sequence prediction.

- *Ensuing relations*. The four representations are all centered on calling relations rather than ensuing relations. For example, Figures 7 (a) (b) (c) and (d) all indicate that both "C" and "D" are possible callees of "A", but none encodes the relation that a call to "D" must follow calls

to "C" if those calls are made by "A"[3]. The lack of ensuing relations makes them fundamentally ill fit for call sequence prediction.

- *Discriminating*. **Control flows**: None of the four representations encodes branches or loops. The static call graph in Figure 7 (a), for instance, fails to show that function "A" is invoked at both branches and "B" is not. The other three representations, on the other hand, completely miss the branch that contains "B". Moreover, none of the representations expresses that "C" is called inside a loop (the dynamic call tree in Figures 7 (c) shows four consecutive calls to "C" in "A" but leaves it unclear whether they are caused by a loop or four different call sites of "C".) Missing control flows hinders these representations for call sequence prediction. For example, the control flows tell us that if and only if the second call sites of "A" is reached, "B" will be called immediately after "A" finishes. None of the four representations captures that constraint. **Calling contexts**: Dynamic call tree and CCT both maintain calling contexts. But static and dynamic call graphs do not. In Figures 7 (a) and (b), for instance, all calls to "C" are aggregated into a single node, despite that they differ in their calling contexts. **Call sites**: None of the representations except dynamic call trees offers a full discrimination of call sites. For instance, the two sites of calls to "C" in "M" are folded into a single node in Figures 7 (a) (b) (d). They hence fail to encode that different call sequences could follow the two calls.
- *Generality*. Dynamic call graphs, call trees, and CCT all contain only the invocations made in some training execution(s) rather than the complete calling relations in the program. Some functions (e.g., "B") absent from them may be called in other runs. It is possible to append these newly encountered calls to these graphs or trees at runtime. But there are no machinery in these representations to overcome the incompleteness and the ambiguity (e.g., by dynamic dispatch) for call sequence prediction.
- *Scalability*. Static and dynamic call graphs are bounded by the number of unique functions in the program. CCT is bounded by the number of distinct calling contexts. They all have reasonable scalability, although sometimes a CCT could be orders of magnitude larger than the program itself. A dynamic call tree, on the other hand, may contain as many nodes as the number of function invocations in a run, often too large for practical usage.

Overall, in their basic forms, the four representations all miss some of the basic properties. They have some variations, the extra features of which may alleviate some issues, but cannot address their inherent limitations. For example, in a call graph with labeled edges, a caller may have multiple calling edges connecting to a callee, with each edge corresponding to a distinct call site. Similarly, CCT can be made call site-aware as well if different call sites of a function are represented with different nodes, even if they have the same calling context [32]. However, these variations do not change the inherent nature of these representations of centering around calling rather than ensuing relations. Neither do they address the issues on control flows or generality.

Consequently, these representations cannot well capture the relevant constraints defined by the program. In Figure 7, for example, none of them reflects the constraint that either "B" or "C" but not any other functions will follow the first invocation of "D". Neither do they reflect that if "A" has been invoked twice by "M" and the current execution point is inside "D", there will be definitely no other function calls by the end of the execution.

The qualitative analysis reveals the high-level limitations of these representations for call sequence prediction; Section 6 confirms them through some quantitative comparisons with PCA.

5. Metrics for Call Sequence Prediction

We find no prior definition of metrics for assessing a call sequence prediction. We introduce three levels of metrics, which are of different strictness, suitable for different uses of the prediction results.

Let Q and $\hat{Q}$ be the true and predicted call sequences, and U and $\hat{U}$ be the set of unique functions in Q and $\hat{Q}$. The three levels of metrics are as follows.

- *Set-level:* It quantifies the closeness between U and $\hat{U}$. We introduce the following notations: $TP = |U \cap \hat{U}|$, $TN = |\overline{U} \cap \overline{\hat{U}}|$, $FP = |\hat{U} - U|$, $FN = |\overline{\hat{U}} - \overline{U}|$; $\overline{U}$ and $\overline{\hat{U}}$ are the set of functions in the entire program that do not appear in U or $\hat{U}$ respectively. ("T" for true, "F" for false, "N" for negative, "P" for positive.) Following information retrieval theory [17], we use two common metrics: **recall**=$TP/|U|$; **precision**=$TP/|\hat{U}|$. They respectively measure how much the true set is uncovered and how precise the prediction set is. To integrate them into a single metric, we borrow the concept of Matthews correlation coefficient (MCC) [31], which takes into account true and false positives and negatives and is generally regarded as a balanced measure. It is defined as MCC=$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$.
  MCC has a value range [-1, 1]. We normalize it to [0, 1] as follows: **Set accuracy=(MCC + 1)/2**.
- *Frequency-level:* Let n_f and $\hat{n_f}$ be the numbers of times the function f appears in Q and $\hat{Q}$ respectively. The **frequency accuracy** of $\hat{Q}$ is $1 - average_{f \in U \cup \hat{U}}(|n_f - \hat{n_f}|/max(n_f, \hat{n_f}))$.
- *Sequence-level:* Let e be the minimum number of atomic editing operations (insertion, deletion, or replacement of a single token in $\hat{Q}$) needed to change $\hat{Q}$ into Q. The **sequence accuracy** of $\hat{Q}$ is $1 - e/max(|Q|, |\hat{Q}|)$. Let Q^* be the sequence of the functions in U ordered in their first

[3] Nodes in a dynamic call tree by default have no specific orders. If extended with a time order, the tree may capture some ensuing relations.

occurrences in Q, and $\hat{Q}^*$ be the counterpart for $\hat{Q}$. The **first-occ sequence accuracy** is the *sequence accuracy* of $\hat{Q}^*$ regarding Q^*.

The usage of "max" in the frequency accuracy and sequence accuracy ensures that the accuracy is in the range of 0 and 100%. For instance, the e in sequence accuracy must be no greater than $max(|Q|, |\hat{Q}|)$ since a naive way to generate Q from $\hat{Q}$ is to replace every token in $\hat{Q}$ with the corresponding one in Q and its number of operations is $max(|Q|, |\hat{Q}|)$.

As an example, assume that the true sequence is "A A B C B D", while the predicted sequence is "A A A B E F", and there are 10 unique functions in the whole program. The measures are as follows: TP=2, TN=4, FP=2, FN=2, recall=0.5, precision=0.5, set accuracy=0.58, frequency accuracy=0.19, sequence accuracy=0.33, first-occ sequence accuracy=0.5.

Set-level measures are the most relaxed among all. They ignore the order and frequency of function calls in the sequences. First-occ sequence accuracy is slightly stronger by considering the order of the first-time occurrences of the functions in Q. They are useful when the prediction is for guiding early compilation or prefetching.

Frequency accuracy reflects how well the prediction captures the hotness of the functions in Q. It is useful for hotness-based optimizations.

Sequence accuracy is the most strict on the difference between two sequences. The usage of atomic editing operations in the definition avoids some misleading effects of alternative definitions. For instance, Hamming distance—which does pair-wise comparison at token level—is sensitive to local differences and cannot precisely measure the similarity of two sequences. For example, Q is "A B C D" while $\hat{Q}$ is "E A B C", accuracy based on Hamming distance is 0, even though the two sequences share a large subsequence. The definition on atomic operations is not subject to the problem. Computing the needed minimum number of operations can be challenging, but some existing tools (e.g., the Linux utility "diff") can be used as the ruler.

6. Evaluation

For evaluation, we concentrate on the following questions:

- Can PCA enable accurate call sequence predictions? What is the time and space cost?
- Is the enabled prediction useful?

For the first question, we design a set of experiments to measure the call sequence prediction accuracies and overhead; for the second question, we apply the prediction results to help JIT decide when to compile which methods for reducing response time. It would be ideal to assess these results in the context of existing techniques. But it is difficult as there are no existing work directly on call sequence prediction. To circumvent the difficulty, we implement three other call sequence predictors by extending most relevant existing techniques.

6.1 Three Alternatives

In Machine Learning, there is a problem called discrete sequence prediction [21], but its prediction target is still just the next symbol in a sequence. To put our results into a context, we implement two representatives of such methods and extend them for call sequence prediction.

Alternative-1: The first is called *Pattern method*, an extension from the single-call predictor by Lee and others [12, 23]. It is based on Markov model. Through a Machine Learning engine, it derives statistical patterns by examining all the $K + 1$-long subsequences of a training sequence, based on which, its predictor looks at the K most recent function calls to predict which function will be called next. The authors showed the usage of the prediction for detecting OS security issues.

Alternative-2: The second is called *TDAG* method, which also exploits frequent subsequences but in a more sophisticated manner through a classical Machine Learning method called Markov Tree [34]. It uses a tree to store frequent subsequences of various lengths and maintains confidence for each tree node. With the tree, it intelligently picks the best subsequence (frequent enough with strong predictive capability) for each prediction. To avoid tree size explosion, it adds some constraints on the nodes and height of the tree [21]. In our implementation, we adopt the same parameter values as in the previous publication.

Both methods were originally designed for predicting only the next symbol. We expand the prediction target naturally to a sequence of calls. The training process remains the same as in the previous work. At a prediction time, the extended methods gives prediction of the next symbol, $s_{\hat{t+1}}$, based on the previous k-symbol sequence (s_{t-k+1}, s_{t-k+2}, $\cdots$, s_t) in their default manner, and then in the same manner, gives prediction $s_{\hat{t+2}}$ by regarding the sequence (s_{t-k+2}, s_{t-k+3}, $\cdots$, $s_{\hat{t+1}}$) as the most recent k-symbol sequence. Other symbols in the time window are predicted likewise. A comparison to these methods helps reveal the benefits of PCA's capitalization of program inherent constraints.

Alternative-3: Although we are not aware of previous usage of the other representations listed in Section 4 for call sequence prediction, they can be adapted to do so in a manner similar to our PCA. We implement such a predictor on CCT, the most sophisticated representation of all of them. It is called *CCT-based predictor*. There are two extensions. First, we add an edge from every node to each of its immediate parents (callers), representing the transition happening when the current function returns. Second, we use profiling to add probabilities to all the edges in the extended CCT in the same way as in PCA construction. As a program executes, each function call triggers one move on the CCT. At prediction time, the predictor walks on the CCT based on the directions of its edges, and outputs as the predicted sequence the functions corresponding to the nodes it encounters. For a

node with multiple outgoing edges, we also experiment with both the ML and random walk approaches.

A comparison to CCT-based predictor helps quantitatively assess the benefits of PCA for its better treatment to control flows, calling contexts and call sites.

6.2 Methodology

All experiments happen on a machine equipped with dual-socket quad-core Intel Xeon E5310 processors that run Linux 2.6.22; the heap size ("-Xmx") is 512MB for all. We use Jikes RVM [1] (v3.1.2), an open-source Java Virtual Machine, as our basic framework. We modify its JIT to derive the FCG from a function's bytecode, and to collect calling sequence for training the edge weights on a PCA and a CCT. The Jikes RVM runs with the default JIT (including both baseline and optimizing compilation and inlining) unless noted otherwise.

We use the Dacapo (2006) benchmark suite [7]. (The latest version of Dacapo does not work well with Jikes RVM [19].) Two programs, *chart* and *jython*, were left out because they fail to run on the Jikes RVM-based profiler. Table 1 shows the benchmarks, their lines of code, the numbers of unique calling contexts, and sequence lengths(i.e., the total numbers of calls a program makes in a run) on the *small* and *default* inputs coming with the benchmark suite. In our experiment, we use *small* runs for training and *default* runs for testing. On most programs, the two runs differ substantially in both the length of call sequences, as shown in Table 1, and the distribution of function calling frequencies included in Appendix A. All executions involve a few JNI calls. As Java uses dynamic dispatch, Table 2 reports the size distribution of the candidate sets of function calls. For all programs except for *bloat*, the call sites with larger than 4 candidate set are less than 5%. We use ten as the upper bound of the candidate table size.

Our evaluation concentrates on the startup phase of program executions. Here, the startup phase refers to the beginning part of a program execution, by the end of which, a major portion of the methods that the whole execution needs have been compiled. Quantitatively, we determine the startup phase by finding the *knee point* on the cumulative compilation curve of an execution. Figure 8 illustrates the concept by depicting the curve of an execution of benchmark *bloat*. Formally, a knee point on a smooth ascending convex curve is defined as the point where the radius of curvature is a local minimum. The cumulative compilation curve of a program execution are often not smooth, but its trend (i.e., when local noises are smoothed out) is typically so. In our experiments, we draw all the cumulative compilation curves of all executions and manually find the knee points through visual examination of the trend of the curves. We observe that the knee points of all of the program executions appear before the 700,000th function call in their executions. For simplicity, we take the first 700,000 function calls as the approximated startup phases of all the programs in our evaluation[4]. In all those executions, a majority of method compilations happen in those startup phases.

Table 1. Benchmark Information

Program	# code lines	# unique call. contexts ($\times 10^3$)	Seq. length ($\times 10^6$) *small*	*default*
antlr	32263	1006	7	490
bloat	73563	1980	9	6276
eclipse	1903219	4816	18	1267
fop	88846	175	3	44
luindex	8570	374	10	740
lusearch	12709	6	9	1439
pmd	49331	8043	6	2727
xalan	243516	163	33	10084

Table 2. Size of Candidate Sets

Program	size distribution 1	2	3	4	≥5
antlr	73%	10%	9%	4%	4%
bloat	49%	15%	8%	3%	25%
eclipse	65%	22%	8%	2%	3%
fop	65%	18%	8%	8%	0%
luindex	57%	26%	15%	1%	1%
lusearch	51%	31%	6%	9%	3%
pmd	70%	18%	5%	3%	4%
xalan	72%	15%	5%	3%	4%

For many applications, the end of the startup phase roughly corresponds to the time when the application finishes initialization and becomes ready to interact with users. The length of the phase, therefore, critically determines the responsiveness of the launches of such applications. It is especially so for utility programs, which, unlike server programs, are often utility tools that do not have a long-running execution, but whose responsiveness is important for user experience. For them, compilation could take a substantial portion of its execution time, especially during the startup stage of their executions. In our experiments of the replay runs of the Dacapo benchmarks, we observe that method compilations take 7∼96% (65% on average) of their startup times.

All reported timing results are average of ten repetitive measurements. Each reported accuracy number of a benchmark is computed by averaging the prediction accuracy of all its prediction windows. In all experiments, a prediction window is in the unit of the number of function calls. If the prediction window size is 20, after a program starts, the predictor is triggered after every 20 function calls to output the prediction of what the next 20 function calls will be.

6.3 Accuracy

Table 3 shows the comparison among the four predictors on all six metrics. In the setting, the prediction window length is 20, and the maximum likelihood is employed for both the PCA and CCT predictors. (Other settings are shown later.) The rightmost column shows the geometrical mean.

PCA results are consistently better than the other predictors, with about 20% higher set accuracy, 40-50% higher

[4] Because the optimizations based on our prediction, as shown in Section 6.4, save more time in the startup phase than in the stable-running phase due to the more compilations in the startup phase, the true speedups for the startup phase could be higher than the reported due to the over approximation of startup phases.

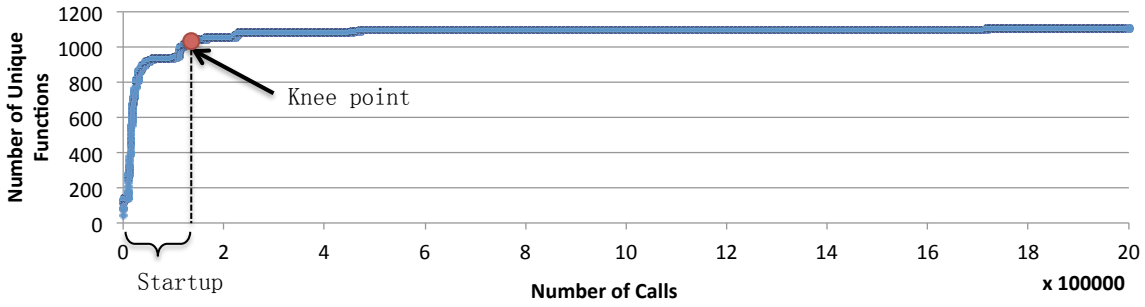


Figure 8. The cumulative compilation curve of benchmark *bloat* and its knee point.

Table 3. Prediction Accuracy (window size=20)

		antlr	bloat	eclipse	fop	luin.	luse.	pmd	xalan	mean
Set accuracy	PCA	0.94	0.96	0.79	0.89	0.90	0.86	0.91	0.92	0.89
	CCT	0.67	0.77	0.58	0.65	0.79	0.72	0.62	0.65	0.68
	TDAG	0.65	0.79	0.62	0.65	0.78	0.69	0.60	0.65	0.68
	Pattern	0.79	0.87	0.51	0.82	0.51	0.86	0.69	0.79	0.72
Set recall	PCA	0.92	0.96	0.65	0.84	0.95	0.96	0.85	0.92	0.87
	CCT	0.69	0.91	0.31	0.66	0.89	0.89	0.52	0.64	0.66
	TDAG	0.65	0.90	0.50	0.77	0.94	0.84	0.50	0.74	0.71
	Pattern	0.63	0.79	0.02	0.66	0.04	0.78	0.40	0.59	0.29
Set prec	PCA	0.87	0.91	0.56	0.75	0.71	0.58	0.81	0.81	0.74
	CCT	0.18	0.34	0.09	0.15	0.40	0.23	0.12	0.17	0.19
	TDAG	0.15	0.42	0.14	0.13	0.35	0.19	0.10	0.15	0.18
	Pattern	0.53	0.70	0.01	0.63	0.04	0.68	0.37	0.57	0.25
Frequency accuracy	PCA	0.78	0.87	0.36	0.66	0.52	0.45	0.74	0.77	0.62
	CCT	0.07	0.13	0.05	0.08	0.26	0.12	0.06	0.12	0.10
	TDAG	0.05	0.11	0.05	0.05	0.22	0.08	0.05	0.05	0.07
	Pattern	0.44	0.62	0.01	0.49	0.03	0.55	0.27	0.43	0.20
1st occ. Sequence accuracy	PCA	0.81	0.88	0.37	0.66	0.62	0.55	0.73	0.77	0.65
	CCT	0.17	0.34	0.08	0.15	0.38	0.22	0.11	0.17	0.18
	TDAG	0.15	0.40	0.13	0.09	0.35	0.18	0.10	0.11	0.16
	Pattern	0.75	0.88	0.00	0.80	0.28	0.92	0.51	0.69	0.20
Full Sequence accuracy	PCA	0.77	0.85	0.26	0.63	0.56	0.45	0.70	0.74	0.59
	CCT	0.01	0.04	0.00	0.05	0.36	0.11	0.02	0.10	0.03
	TDAG	0.00	0.00	0.00	0.00	0.35	0.04	0.00	0.00	0.01
	Pattern	0.73	0.86	0.00	0.79	0.38	0.89	0.51	0.68	0.28

Table 4. Size and Training Time

Program	Size (MB)				Training Time (sec)			
	PCA	CCT	TDAG	Pattern	PCA	CCT	TDAG	Pattern
antlr	0.55	0.96	0.02	1.7	21	15	1019	325
bloat	0.56	1.77	0.03	83	21	21	1316	6735
eclipse	1.33	1.95	0.11	15	68	39	2710	4385
fop	0.43	0.69	0.05	9.7	19	7	359	1884
luindex	0.23	0.07	0.01	0.92	23	24	1510	145
lusearch	0.20	0.02	0.01	1.1	24	22	1335	175
pmd	0.49	0.73	0.03	51	6	3	78	3724
xalan	0.55	0.24	0.02	46	11	6	477	3146

frequency accuracy, about 40% higher first occurrence sequence accuracy, and 30-56% higher whole sequence accuracy. As the metrics become stricter, the accuracies of all methods except PCA drop sharply to no greater than 30% on average. The PCA results also show some considerable drop, but it still keeps the accuracy on half of the benchmarks higher than 70% on all the metrics. There are some quite challenging programs. For example, the program *eclipse*, for its large number of functions and complex control flows, causes the CCT, TDAG and Pattern methods to get near zero frequency and sequence accuracies and less than 62% set accuracy. The PCA does not get very high frequency and sequence accuracies either, but it manages to still achieve a 79% set accuracy.

It is important to note the connections between prediction errors and the usefulness of the prediction. It is generally true that a more accurate prediction may give a larger benefit for program optimizations. However, many program optimizations have a certain degree of tolerance of prediction errors. For instance, when predicted call sequences are used to trigger function prefetching from remote servers, a 80% means that 20% of the prefetched functions may not be useful. The prefetching of them may waste some bandwidth and energy. But the prefetching of the 80% useful functions may still shorten the execution time of the program substantially and considerably outweigh the loss by the 20%. In the next subsection, we will see that the 79% set accuracy on *eclipse*, for example, yields up to a 10% speedup when the prediction is applied to code cache management.

Another observation is that the CCT-based approach is overall no better than the Pattern-based approach in terms of prediction accuracies. It indicates that although capitalization of program structure can be beneficial for call sequence prediction, how to capitalize it and using what representation to encode the structure are critical: The lack of support in CCT for various levels of contexts leaves its capitalization of program structures ineffective.

Figure 9 gives a more detailed report. (TDAG performs the worst and is hence omitted for lack of space.) As the prediction scope increases, the difficulty for prediction increases. All three methods show a certain degree of reduction in accuracy. On two programs with some frequently occurring call sequence patterns (*fop* and *lusearch*), the Pattern method performs well, yielding set accuracies close to those from the PCA method. But across all window sizes, PCA maintains an average accuracy higher than 80%, about a 20% edge over the other methods.

Another dimension of comparison is between the Random Walk and Maximum Likelihood. From Figure 10, we can see their influence on the PCA method in terms of three types of accuracies. For most programs, Maximum Likelihood gives higher accuracies. An exception is *lusearch*, which has a number of loops with a low loop trip-count. Being able to get out of the loop early, Random Walk helps

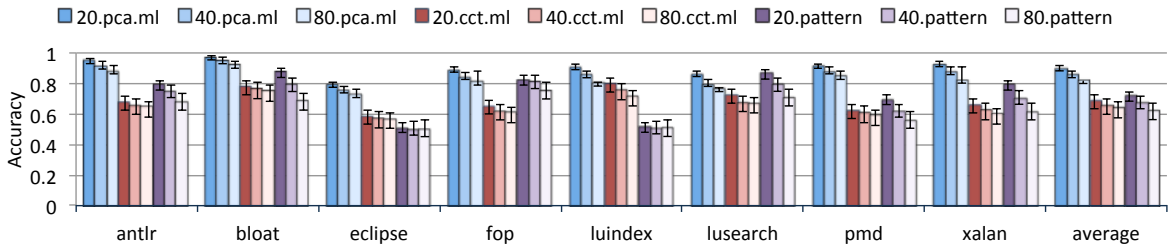


Figure 9. Comparison of Set Accuracy among different prediction window sizes.

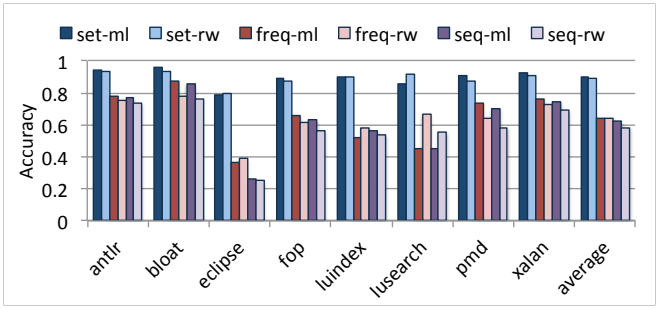


Figure 10. Comparison between maximum likelihood (ml) and random walk (rw). (window size=20)

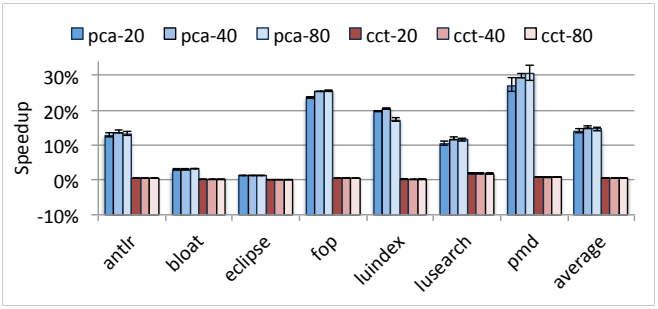


Figure 11. Speedup when call sequence prediction is used for parallel JIT compilation (Two compilation threads are used).

the prediction. But overall, the average accuracies show that their influence on PCA and CCT does not differ much.

Besides accuracy, we have examined the size, training time, and prediction time of the four methods. As Table 4 shows, the pattern-based predictor can be much larger than the other three predictors, when there are many different subsequences (e.g., *bloat*, *pmd*, and *xalan*.) The TDAG method successfully reduces the size of the predictor through its constrained tree structure (but fails in enhancing the prediction accuracy.) The training time of both Pattern and TDAG are several orders of magnitude longer than the other two predictors. PCA predictors are slightly larger than CCT predictors; both are quick to train. The time taken to perform a prediction is independent of benchmarks. The PCA and CCT predictors take $32\mu s$ and $8\mu s$ to predict a 40-call sequence respectively, negligible compared to the time needed to compile the functions by JIT. By contrast, the TDAG predictors take $827\mu s$ on average, caused by the Markov tree searching at each call prediction. The time overhead of state tracking is marginal, no more than 2% for the programs.

6.4 Uses

The PCA-based call sequence prediction may benefit many uses, such as guiding the replacement policy in code cache to reduce cache misses [16], enabling better prefetching to enhance instruction cache performance [27], and helping preload remote classes in mobile computing.

In this work, we experiment with parallel JIT compilation. Parallel JIT creates multiple threads to compile functions. By default, it compiles a method only after the method gets called. With the prediction of upcoming method calls, the compilation of a method could happen earlier, enabling better overlap between execution of the program and compilations of to-be-invoked methods. The overlap can help prevent some (part of) compilations from appearing on the critical path of the program execution. It is especially beneficial for speeding up the startup phase of a program.

In our experiment, we implement a prototype of parallel JIT on JikesRVM. For parallel JIT to work well, there are two aspects. The first is to determine the appropriate optimization level to use for the target function, the other is to decide the good time to compile the function. There are many studies on predicting the optimization levels [5]. The focus of our experiment is on the compilation timing aspect. So to avoid the distractions of the other factor, we use the advice files produced by JikesRVM for all experiments. The files record the appropriate optimization level for each method based on its importance.

In our experiments, after each prediction window, the JIT invokes the predictor to get the predicted call sequence in the next time window. It then creates compilation events for the methods in the predicted sequence that have not been compiled before, and puts those events into the compilation event queue in JikesRVM. Compilation threads automatically dequeue the events and conduct the compilation.

The number of compilation threads we tested ranges from two to seven. We see diminishing gains from parallel JIT when the number is greater than two. As two is the most cost efficient, Figure 11 reports the speedup in that setting. We chose CCT method as the representative of alternatives to PCA for its relative ease to use and having a similar or higher prediction accuracy and prediction speed than others.

The baseline in Figure 11 is the performance when the default replay mode is enabled, which uses the same compilation levels as in the advice files but uses no prediction of call sequences. Given that most studied programs are utility programs, their responsiveness rather than steady-state performance is what often matters. The performance is based on the end-to-end wall-clock time of the startup phase of an execution.

Call sequence prediction not only increases compilation parallelism, but also enables better overlapping between execution and compilation. The PCA-based predictions, in all three window sizes, lead to more than 20% speedups on three programs, and an average around 15% on all seven programs ("xalan" fails working in the default replay mode). In contrast, the CCT-based prediction gives only slight speedup on lusearch and pmd. It is due to its low prediction precision: an average of 19% versus the 74% of PCA-based approach as Table 3 shows. In consequence, many useless functions are compiled, which delays the compilation of those useful ones.

Two other observations are worth mentioning. First, a larger prediction window does not always deliver higher speedup. It is because on large windows, the prediction, although finding more useful methods to compile, could enqueue more methods that won't be used in the near future. Second, some programs (e.g., bloat) that have high prediction precision and accuracy do not show large speedups. It is because the speedup also depends on how much the compilation time weight in the overall running time. If it is small, the entire potential of parallel JIT is small.

7. Related Work

7.1 Program Representations

Besides the work mentioned in Section 4, some other studies also relate with calling contexts. Program summary graphs by Callahan [11], for instance, use nodes for formal and true parameters of functions and edges for their bindings. By showing the flow of values across procedures, the graphs facilitate inter-procedural data flow analysis. The *probabilistic calling context* by Bond and McKinley [8] offers an efficient way to collect and represent calling contexts. Later work proposes other ways to encode calling contexts precisely [35]. Alur and others [2] analyzed Recursive State Machines for representing recursive procedural calls in the context of system verification. As Section 2 discusses, calling context is only one of the necessary conditions for call sequence prediction. Without capturing control flows, call sites discrimination, and *ensuing* relations among calls, calling contexts alone do not suffice for call sequence prediction. Moreover, these representations provide no machinery—such as the v-nodes, α-stack in PCA—to overcome the various ambiguities (e.g., by dynamic dispatch) for call sequence prediction.

Some previous studies aim at finding hot code or data streams [14, 22]. Similar to the pattern method implemented in Section 6, these methods centered on statistical patterns of sequences, and hence suffer from the diminishing regularity as prediction scope increases. Moreover, predicting cold call sequences and dealing with local variations (e.g., caused by branches) are essential for our call sequence prediction and its usage for startup time reduction.

A previous study uses DFA to record traces found in a binary translation process [30]. It starts from traces of function calls and builds automata based on their patterns. Another study that uses DFA is to construct object usage models [33]. For each abstract object, it builds an automaton with some places in the code as nodes and function calls related to that object as edge labels. Neither of the two studies is for predicting function call sequences; the first is for compressing traces and the second is for detecting anomalies in object usage. Consequently, their designs are not suitable for call sequence predictions. First, they are at the level of either traces or objects, rather than the whole program. When the scope goes to the whole program level with potentially infinite recursions, it becomes more complex than the pure automata can model. Second, they do not have any of the three stacks in PCA. The stacks adds more expressiveness to automata. More importantly, the stacks, along with unique call site IDs, inject into PCA the capability to discriminate different call sites and calling contexts in the prediction. In addition, their designs give no systematic treatment to ambiguous or unexpected function calls.

The probabilities associated with the v-nodes were inspired by some prior work on virtual function target prediction [6]. There are many other works trying to predict program behaviors beyond function calls, such as function returning values [28], load value prediction [29]. They center on leveraging statistical patterns rather than constraints through program representations.

7.2 Stochastic Models

In time series related domains, lots of data analysis has been based on probabilistic state machines (e.g., weighted automata [26], probabilistic pushdown automata [10]), or other stochastic models (e.g., Markov Model, Markov Tree). PCA can be regarded as an augmented form of probabilistic state machines that is specially customized for leveraging constraints coded in programs and for accommodating their various complexities, reflected by its design of the three types of stacks, diamond and v-nodes, and the edge and node labels. These features make PCA more effective in predicting function call sequences, as exemplified by the comparison with Markov Trees in the evaluation.

8. Conclusion

In this paper, we have presented the first systematic study in exploiting program defined constraints to enable function call sequence prediction. We have introduced PCA, a new program representation that captures both the inherent calling relations among functions, and the probabilistic nature

of execution paths determined by conditional branches and loops. Experiments show that the new approach can produce more accurate call sequence predictions than alternatives. As a fundamental representation of function calling relations, PCA may open up many new opportunities for optimizing the performance of modern virtual machines and beyond.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. Martin White suggested MCC for metric. This material is based upon work supported by DOE Early Career Award, IBM CAS Fellowship, and the National Science Foundation under Grant No. 1320796 and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, DOE, or IBM.

References

[1] Jikes rvm. `http://jikesrvm.org`.

[2] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, July 2005.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, 1997.

[4] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems*, 21(4), 2003.

[5] M. Arnold, A. Welc, and V.T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA*, pages 297–311, 2005.

[6] D. F. Bacon and P. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, 1996.

[7] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[8] M. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, 2007.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *PLDI*, 2010.

[10] Toms Brzdil, Javier Esparza, Stefan Kiefer, and Antonn Kucera. Analyzing probabilistic pushdown automata. *Formal Methods in System Design*, 43(2):124–163, 2013.

[11] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI*, 1988.

[12] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

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

[14] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI*, Berlin, Germany, June 2002.

[15] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. Finding the limit: Examining the potential and complexity of compilation scheduling for jit-based runtime systems. In *ASPLOS* , pages 607–622, 2014.

[16] A. Guha, K. Hazelwood, and M. L. Soffa. Balancing memory and performance through selective flushing of software code caches. In *CASES*, 2010.

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

[18] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. In *ISCA* , 2006.

[19] Jikes rvm project and status. http://http://jikesrvm.org/Project+Status.

[20] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, 2010.

[21] P. Laird and R. Saul. Discrete sequence prediction and its applications. *Machine Learning*, 15:43–68, 1994.

[22] J. R. Larus. Whole program paths. In *PLDI*, Atlanta, Georgia, May 1999.

[23] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[24] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[25] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In *Symp. on the Foundations of Software Engineering*, 2008.

[26] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23:269–311, 1997.

[27] P. Nagpurkar, H. Cain, M. Serrano, J. Choi, and C. Krintz. Call-chain software instruction prefetching in j2ee server applications. In *PACT*, 2007.

[28] C. J. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report, McGill University, 2009.

[29] M. Burtscher and B. G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. Journal of Instruction-Level Parallelism. 1999.

[30] J. Porto, G. Araujo, E. Borin, and Y. Wu. Trace execution automata in dynamic binary translation. In *Proceedings of 3rd Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2010.

[31] D. Powers. Evaluation: From precision, recall and f-factor to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, pages 37–63, 2007.

[32] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *PPPJ*, 2011.

[33] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Symp. on the Foundations of Software Engineering*, 2007.

[34] R. William. Dynamic history predictive compression. *Information Systems*, 13(1):129–140, 1988.

[35] W.M.Summer, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE*, 2010.

[36] Y. Wu and J. Larus. Static branch frequency and program profile analysis. In *Proceedings of the International Symposium on Microarchitecture*, 1994.

A. Function Call Frequency Distributions

Figures 12 and 13 show the distributions of the function call frequencies in the training (small) and testing (default) runs. The X-axis is the method ID, and each point in the graphs show the percentage of the number of calls of a method over the total number of calls in a run. For better legibility, the points are connected into curves. All programs except lusearch and xalan show substantial differences in the distributions.

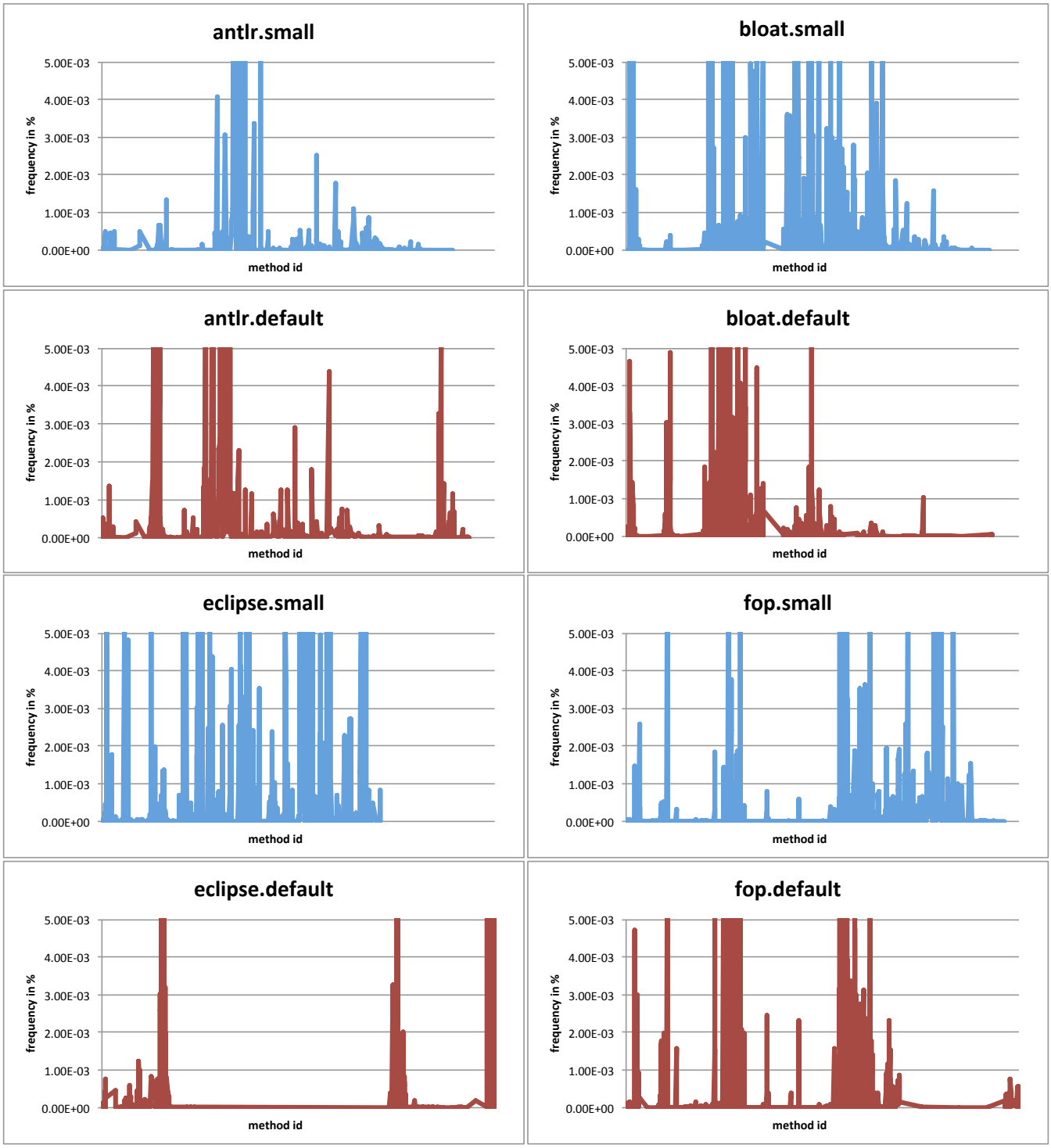


Figure 12. Function Call Frequency Distributions (Part I)

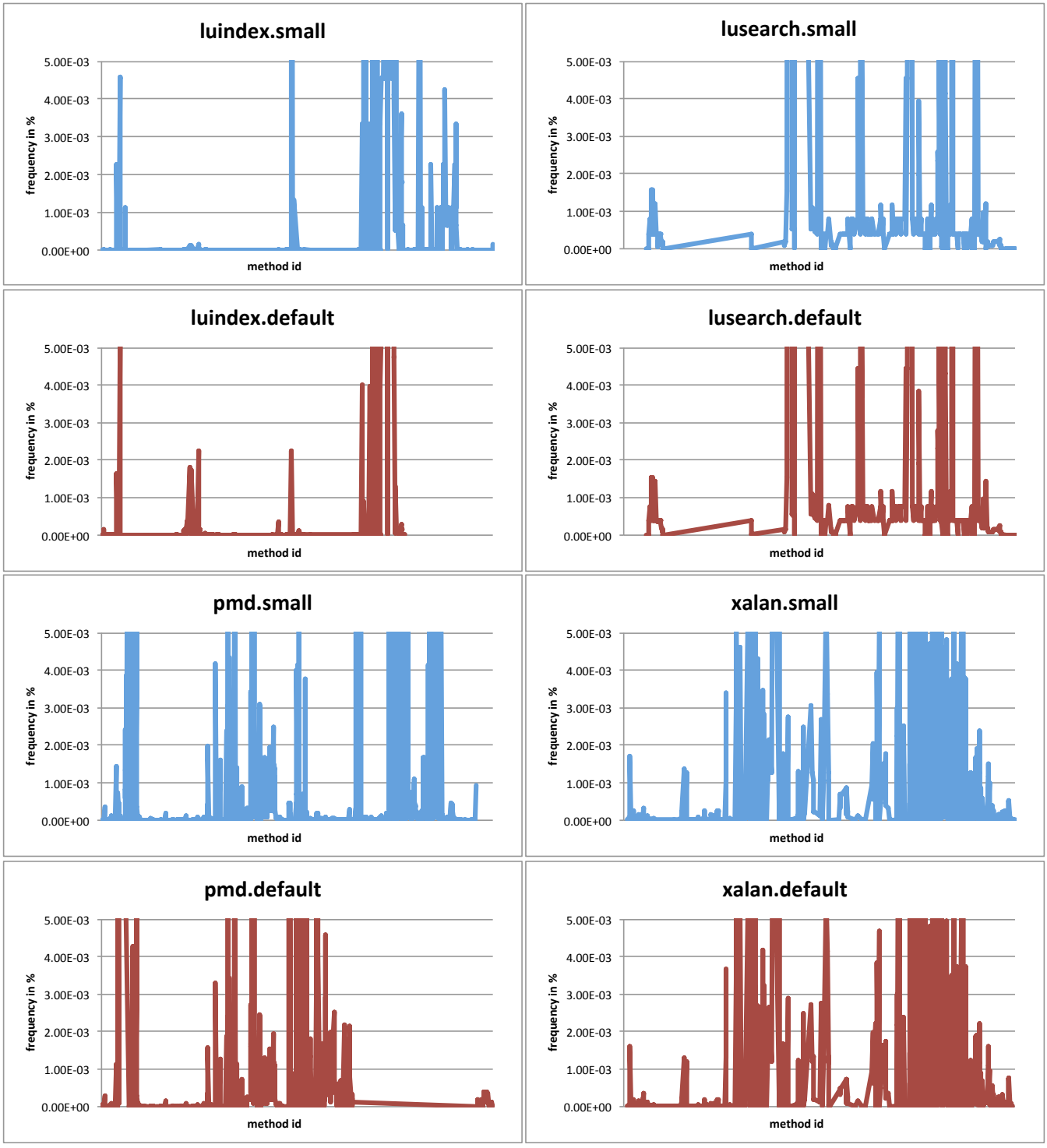


Figure 13. Function Call Frequency Distributions (Part II)