Correcting the Dynamic Call Graph Using Control-Flow Constraints[⋆]

Byeongcheol Lee, Kevin Resnick, Michael D. Bond, and Kathryn S. McKinley

The University of Texas at Austin

Abstract. To reason about programs, dynamic optimizers and analysis tools use sampling to collect a *dynamic call graph* (DCG). However, sampling has not achieved high accuracy with low runtime overhead. As object-oriented programmers compose increasingly complex programs, inaccurate call graphs will inhibit analysis and optimizations. This paper demonstrates how to use static and dynamic *control flow graph* (CFG) constraints to improve the accuracy of the DCG. We introduce the *frequency dominator* (FDOM), a novel CFG relation that extends the dominator relation to expose static relative execution frequencies of basic blocks. We combine conservation of flow and dynamic CFG basic block profiles to further improve the accuracy of the DCG. Together these approaches add minimal overhead (1%) and achieve 85% accuracy compared to a perfect call graph for SPEC JVM98 and DaCapo benchmarks. Compared to sampling alone, accuracy improves by 12 to 36%. These results demonstrate that static and dynamic control-flow information offer accurate information for efficiently improving the DCG.

1 Introduction

Well-designed object-oriented programs use language features such as encapsulation, inheritance, and polymorphism to achieve reusability, reliability, and maintainability. Programs often decompose functionality into small methods, and virtual dispatch often obscures call targets at compile time. The dynamic call graph (DCG) records execution frequencies of call site-callee pairs, and dynamic optimizers use it to analyze and optimize whole-program behavior [2,3,4,5,11,21]. Prior approaches sample to collect the DCG, trading accuracy for low overhead. Software sampling periodically examines the call stack to construct the DCG [4,12,17,20,24]. Hardware sampling lowers overhead by examining hardware performance counters instead of the call stack, but gives up portability. All DCG sampling approaches suffer from sampling error, and timer-based sampling suffers from timing bias. Arnold and Grove first measured and noted these inaccuracies [4].

[⋆] This work is supported by NSF CCF-0429859, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, Samsung, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Figure 5(a) (appears in Section 6) shows DCG accuracy for the SPEC JVM98 benchmark *raytrace* using Jikes RVM default sampling. Each bar represents the *true* relative frequency of a DCG edge (call site and callee) from a fully instrumented execution. Each dot is the frequency according to sampling. Vertical dashed lines separate the calling method. Notice that many methods make calls with the same frequency (i.e., bars have the same magnitude between dashed lines), but sampling tells a different story (i.e., dots are not horizontally aligned). These errors are due to timing bias.

This paper presents new *DCG correction* algorithms to improve DCG accuracy with low overhead (1% on average). Our insight is that a program's static and dynamic *control flow graph* (CFG) constrains possible DCG frequency values. We introduce the static *frequency dominator* (FDOM): given statements x and y, x FDOM y if and only if x executes at least as many times as y. FDOM extends the dominator and strong region relations on CFGs. For example, FDOM tells us when two calls must execute the same number of times because the static control flow dictates that their basic blocks execute the same number of times.

We also exploit dynamic *basic block profiles* to improve DCG accuracy. Most dynamic optimizers collect accurate control-flow profiles such as basic block and edge profiles to make better optimization decisions [1,3,12,16,17]. We show how to combine these constraints to further improve the accuracy of the DCG. Our intraprocedural and interprocedural correction algorithms require a single pass over the basic block profile, which we perform periodically.

We evaluate DCG correction in Jikes RVM [3] on the SPEC JVM98 and DaCapo [8] benchmarks. Our approach improves DCG accuracy over the default sampling configuration in Jikes RVM, as well as over the *counter-based sampling* (CBS) configuration recommended by Arnold and Grove [4]. Compared to a perfect call graph, default sampling attains 52% accuracy and DCG correction algorithms boost accuracy to 71%; CBS by itself attains 76% accuracy and DCG correction boosts its accuracy to 85%, while adding 1% overhead.

Clients of the DCG include alias analysis, escape analysis, recompilation analysis, and inlining. We use inlining to evaluate accurate DCGs. The adaptive compiler in Jikes RVM periodically recompiles and inlines hot methods. We modify Jikes RVM to apply DCG correction immediately before recompilation. We measure the potential of inlining with a perfect call graph, which provides a modest 2% average improvement in application time, but significantly improves *raytrace* by 13% and *ipsixql* by 12%. DCG correction achieves a similarly high speedup on *raytrace* and improves average program performance by 1%. We speculate that these modest improvements are the result of tuning the inlining heuristics with inaccurate call graphs and that further improvements are possible.

2 Background and Related Work

This section first discusses how dynamic optimizers sample to collect a DCG with low overhead. It then compare the new frequency dominator relation to

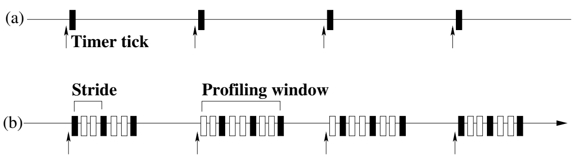


Fig. 1. Sampling. Filled boxes are taken samples and unfilled boxes are skipped. Arrows are timer ticks. (a) Timer-based sampling: one sample per timer tick. (b) CBS: multiple samples per tick, randomly skips initial samples, and strides between samples.

dominators and strong regions. Finally, it compares DCG correction to previous static call graph construction approaches for ahead-of-time compilers.

2.1 Collecting Dynamic Call Graphs

Dynamic optimizers could collect a *perfect* DCG by profiling every call, but the overhead is too high [4]. Some optimizers profile calls fully for some period of time and then turn off profiling to reduce overhead [17,20]. For example, HotSpot adds call graph instrumentation only in unoptimized code [17]. Suganama et al. insert call instrumentation, collect call samples, and then remove the instrumentation [20]. This *one-time profiling* keeps overhead down but loses accuracy when behavior changes.

Many dynamic optimizers use software sampling to profile calls and identify hot methods [4,6,12]. Software-based approaches examine the call stack periodically and update the DCG with the call(s) on the top of the stack. For example, Jikes RVM and J9 use a periodic timer that sets a flag that triggers the system to examine the call stack at the next *yield point* and update the DCG [6,12]. These systems insert yield points on method entry and exit, and on back edges.

Figure 1(a) illustrates timer-based sampling. Arnold and Grove show that this approach suffers from insufficient samples and *timing bias*: some yield points are more likely to be sampled than others, which skews DCG accuracy [4]. To eliminate bias, they present *counter-based sampling* (CBS), which takes multiple samples per timer tick by first randomly skipping zero to *stride-1* samples and then striding between samples. Figure 1(b) shows CBS configured to take three samples for each timer tick and to stride by three. By widening the profiling windows, CBS improves DCG accuracy but increases profiling overhead. They report a few percent overhead to attain an average accuracy of 69%, but to attain 85% accuracy, they hit some pathological case with 1000% overhead. With our benchmarks, their recommended configuration attains 76% accuracy compared to a perfect call graph, whereas our approach combined with the recommended configuration reaches 86% accuracy by adding 1% overhead.

Other dynamic optimizers periodically examine hardware performance counters such as those in Itanium. All sampling approaches suffer from sampling error, and timer-based sampling approaches suffer from timing bias as well. DCG correction can improve the accuracy of any DCG collected by sampling and Section 6 demonstrates improvements over two sampling configurations.

Zhuang et al. [24] present a method for efficiently collecting the calling context tree (CCT), which represents the calling context of nodes in the DCG. Their work is orthogonal to ours since they add another dimension to the DCG (context sensitivity), while we improve DCG accuracy. We believe that our correction approach could improve CCT accuracy as well.

2.2 Constructing the DCG Using Control-Flow Information

Static compilers have traditionally used control-flow information to construct a call graph [13,23]. Hashemi et al. use static heuristics to construct an estimated call frequency profile [13], and Wu and Larus construct an estimated edge profile for estimating call frequency profile [23] for C programs. These approaches use static *heuristics* to estimate frequencies, while DCG correction uses static *constraints* and combines them with dynamic profile information.

2.3 The Dominator Relation and Strong Regions

This paper introduces the *frequency dominator* (FDOM) relation, which extends *dominators* and *strong regions* [7,10,22]. The set of dominators and post-dominators of x is the set of y that will execute at least once if x does. The set that frequency dominates x, is the subset that executes at least as many times as x. While strong regions find vertices x and y that must execute the same number of times, FDOM also identifies vertices x and y where y must execute at least as many times as x.

3 Dynamic Call Graph Correction Algorithms

This section describes DCG correction algorithms. We first present formal definitions for a control flow graph (CFG) and the dynamic call graph (DCG). We introduce the *frequency dominator* (FDOM) and show how to apply its static constraints to improve the accuracy of the DCG, and how to combine them with dynamic CFG frequencies to further improve the DCG.

3.1 Terminology

A *control flow graph* represents *static* intraprocedural control flow in a method, and consists of basic blocks (V) and edges (E). Figure 2 shows an example control flow graph CFG_p that consists of basic blocks *ENTRY*, a, b, c, d, e, and *EXIT*. The dashed lines show edges between basic blocks. The dark edges show calls between methods (other CFGs). A *call edge* represents a method call, and consists of a *call site* and a *callee*. An example call edge in Figure 2 is e_5, the call from cs_c to CFG_t. The DCG of a program includes the *dynamic* frequency of each call edge, from some execution. For a call site cs, $OutEdges(cs)$ is the set of call edges that start at call site cs. $OutEdges(cs_a) = \{e_3, e_4\}$ in Figure 2. For a method m, $InEdges(m)$ is the set of call edges that end at m. $InEdges(CFG_t) = \{e_4, e_5\}$ in Figure 2.

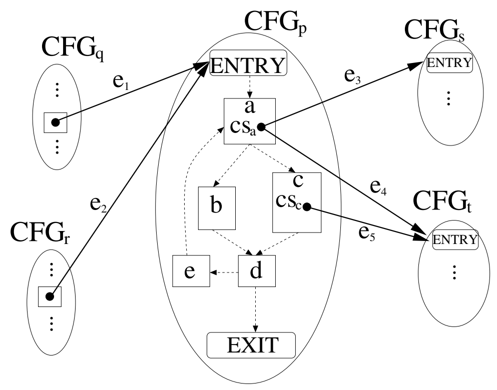


Fig. 2. Example dynamic call graph (DCG) and its control flow graphs (CFGs)

Definition 1. *The* INFLOW *of a method m is the total* flow *into m:*

$$\mathrm{INFLOW}(m) \equiv \sum_{e \in \mathrm{InEdges}(m)} f(e)$$

where $f(e)$ is the execution frequency of call edge e. INFLOW(m) *in an accurate DCG is the number of times m executes.*

Definition 2. *The* OUTFLOW *of a call site* cs *is:*

$$\mathrm{OUTFLOW}(\mathrm{cs}) \equiv \sum_{e \in \mathrm{OutEdges}(\mathrm{cs})} f(e)$$

OUTFLOW(cs) *in an accurate DCG is the number of times* cs *executes.*

Because a sampled DCG has timing bias and sampling errors, the DCG yields *inaccurate OUTFLOW* and *INFLOW* values. DCG correction corrects *OUTFLOW* using constraints provided by static and dynamic control-flow information (doing so indirectly corrects method *INFLOW* as well).

DCG correction maintains the relative frequencies between edges coming out of the same call site (which occur because of virtual dispatch), and does not correct their relative execution frequencies. For example, DCG correction maintains the sampled ratio between $f(e_3)$ and $f(e_4)$ in Figure 2.

3.2 The Frequency Dominator (FDOM) Relation

This section introduces the *frequency dominator* relation, a static property of CFGs that represents constraints (theorems) on program statements' relative execution frequencies. Due to space constraints, we omit the algorithms for computing FDOM and only sketch the constraint propagation algorithms based on FDOM. The detailed algorithms may be found in an extended technical report [15]. The FDOM algorithm is closely related to dominator algorithms [10]. Like the dominator relation, FDOM is reflexive and transitive.

Definition 3. *Frequency Dominator (FDOM). Given statements x and y in the same method, x FDOM y if and only if for every possible path through the method, x must execute at least as many times as y. We also define FDOM(y) $\equiv \{x \mid x \text{ FDOM } y\}$.*

3.3 Static FDOM Constraints

We first show how to propagate the FDOM constraint to DCG frequencies.

Theorem 1. ***FDOM OUTFLOW Constraint:*** *Given method m and two call sites cs_1 and cs_2 in m, if cs_1 FDOM cs_2,* OUTFLOW(cs_1) $\geq$ OUTFLOW(cs_2)

Intuitively, the *OUTFLOW* constraint tells us that flow on two call edges is related if they are related by FDOM. For example, in Figure 2, cs_a *FDOM* cs_c and thus $OUTFLOW(cs_a) \geq OUTFLOW(cs_c)$.

Theorem 2. ***FDOM INFLOW Constraint:*** *Given method m, if cs FDOM ENTRY (m's entry basic block),* INFLOW(m) $\leq$ OUTFLOW(cs)

Intuitively, the *INFLOW* constraint specifies that a call site must execute at least as many times as a method that always executes the call site.

3.4 Static FDOM Correction

We use an algorithm called *FDOMOutflowCorrection* to apply the *FDOM OUTFLOW* constraint to a sampled DCG. We give pseudocode in the technical report [15]. The algorithm compares the sampled *OUTFLOW* of pairs of call sites that satisfy the FDOM relation. If their *OUTFLOW*s violate the *FDOM OUTFLOW* constraint, *FDOMOutflowConstraint* sets both *OUTFLOW*s to the maximum of their two *OUTFLOW*s. After processing a method, *FDOMOutflowConstraint* scales the *OUTFLOW*s of all the method's call sites to preserve the sum of the frequencies out of the method. For instance, consider a call site cs_1 and a call site cs_2 in the same method, and suppose that cs_1 executes at least as many times as cs_2 due to the *FDOMOutflowConstraint*. However, suppose also that the call graph profiler samples cs_1 10 times and cs_2 30 times since the program spends a lot of time executing the callee of cs_2. The *FDOMOutflowCorrection* algorithm corrects this anomaly and assigns 30 to $OUTFLOW(cs_1)$, and scales the two *OUTFLOW*s by $(10+30)/(30+30) = 2/3$ so the *OUTFLOW* sum is preserved. Then both call sites have 20 as their corrected execution frequency.

We also implemented correction algorithms using the *INFLOW* constraint, but they degraded DCG accuracy in some cases. This class of correction algorithms requires high accuracy in the initial *INFLOW* for a method to subsequently correct its *OUTFLOW*. In practice, we found that errors in *INFLOW* information propagated to the *OUTFLOW*s, degrading accuracy.

3.5 Dynamic Basic Block Profile Constraints

This section describes how to incorporate constraints on DCG frequencies provided by basic block profiles, and the following section shows how to correct the DCG with them. The *Dynamic OUTFLOW* constraint computes execution ratios from the basic block execution frequency profiles, and then applies these ratios to the *OUTFLOW* of call sites in the basic blocks.

Theorem 3. ***Dynamic OUTFLOW Constraint:*** *Given two call sites cs_1 and cs_2, and execution frequencies $f_{\text{bprof}}(\text{cs}_1)$ and $f_{\text{bprof}}(\text{cs}_2)$ provided by a basic block profile,*

$$\frac{\text{OUTFLOW}(\text{cs}_1)}{\text{OUTFLOW}(\text{cs}_2)} = \frac{f_{\text{bprof}}(\text{cs}_1)}{f_{\text{bprof}}(\text{cs}_2)}$$

We apply the *Dynamic OUTFLOW* constraint within the same method, i.e., *intraprocedurally.* Edge profiles alone do not compute accurate relative basic block profiles between methods, i.e., basic block profiles with *interprocedural accuracy.* To attain interprocedural accuracy, we experiment with using low-overhead method invocation counters to provide basic block profiles interprocedural accuracy. In this case, *Dynamic OUTFLOW* can correct call sites in different methods (see Section 4).

Theorem 4. ***Dynamic INFLOW Constraint:*** *Given a method m with a single basic block and a call site cs in m,* $\text{INFLOW}(m) = \text{OUTFLOW}(cs)$

The *Dynamic INFLOW* constraint uses the total flow (frequency) coming into the method to constrain the flow leaving any call sites in the method (*OUTFLOW*). When basic block profiles do not have interprocedural accuracy, the *Dynamic INFLOW* constraint is useful for methods with a single basic block because the *Dynamic OUTFLOW* constraint cannot constrain the *OUTFLOW* of call sites in a single basic block.

3.6 Dynamic Basic Block Profile Correction

We use an algorithm called *DynamicOutflowCorrection* to apply the *Dynamic OUTFLOW* constraint [15]. This algorithm sets the *OUTFLOW* of each call site cs to $f_{bprof}(cs)$, its frequency from the basic block profile. The algorithm then scales all the *OUTFLOW* values so that the method's total *OUTFLOW* is the same as before (as illustrated in Section 3.4). This scaling helps to maintain the frequencies due to sampling across disparate parts of the DCG.

Since *DynamicOutFlowCorrection* determines corrected DCG frequencies using a basic block profile, accuracy may suffer if the basic block profile is inaccurate. Jikes RVM collects an edge profile (which determines the basic block profile) in baseline-compiled code only, so phased behavior may affect accuracy, although we find that the edge profile is accurate enough to improve DCG accuracy in our experiments. To avoid the effects of phased behavior, DCG correction could use edge profiles collected continuously [1,9].

We also use an algorithm called *DynamicInflowCorrection* that applies the *Dynamic INFLOW* constraints to the DCG [15]. For each method with a single basic block, *DynamicInflowCorrection* sets the *OUTFLOW* of each call site in the method to the *INFLOW* of the method. As in the case of the FDOM *INFLOW* constraint, we do not use the *Dynamic INFLOW* constraint together with an intraprocedural edge profile. However, with an interprocedural edge profile, *INFLOW* is accurate enough to improve overall DCG accuracy.

4 Implementing DCG Correction

Dynamic compilation systems perform profiling while they execute and optimize the application, and therefore DCG correction needs to occur at the same time with minimal overhead.

We minimize DCG correction overhead by limiting its frequency and scope. We limit correction's frequency by delaying it until the optimizing compiler requests DCG information. The correction overhead is thus proportional to the number of times the compiler selects optimization candidates during an execution. Correction overhead is thus naturally minimized when the dynamic optimizer is selective about how often and which methods to recompile.

We limit the scope of DCG correction by localizing the range of correction. When the compiler optimizes a method m, it does not require the entire DCG, but instead considers a localized portion of the DCG relative to m. Because we preserve the call edge frequency sum in the *OUTFLOW* correction algorithm, we can correct m and all the methods it invokes without compromising the correctness of the other portions of the DCG. Because we preserve the DCG frequency sum, the normalized frequency of a call site in a method remains the same, independent of whether call edge frequencies in other methods are corrected or not.

For better interaction with method inlining, one of the DCG clients, we limit correction to *nontrivial* call edges in the DCG. Trivial call edges by definition are inlined regardless of their measured frequencies because their target methods are so small that inlining them always reduces the code size.

Table 1 summarizes the correction algorithms and their scope. The algorithms take as input the set of call sites to be corrected. Clearly, for FDOM correction, the basic unit of correction is the call sites within a procedure boundary. For dynamic basic block profile correction, there are two options. The first limits the call site set to be within a procedural boundary, and the second corrects all the reachable methods. Since many dynamic compilation systems support only high precision intraprocedural basic profiles, the first configuration represents how much DCG correction would benefit these systems.

Because our system does not collect interprocedural basic block profiles, we implement interprocedural correction by adding method counters. DCG correction multiplies the counter value by the normalized intraprocedural basic block frequency. We find this mechanism is a good approximation to interprocedural basic block profiles.

Table 1. Call Graph Correction Implementations

Correction algorithm	**Input**	**Correction unit**	**Algorithms**
Static FDOM CF Correction	Sampled DCG	Call sites in method to be optimized	*FDOMOutflowCorrection*
Dynamic Intraprocedural CF Correction	Sampled DCG block profile	Call sites in method to be optimized	*DynamicOutflowCorrection*
Dynamic Interprocedural CF Correction	Sampled DCG block profile	Call sites in DCG	*DynamicOutflowCorrection* & *DynamicInflowCorrection*

5 Methodology

This section describes our benchmarks, platform, implementation, and VM compiler configurations. We describe our methodologies for accuracy measurements against the perfect dynamic call graph (DCG), overhead measurements, and performance measurements.

We implement and evaluate DCG correction algorithms in Jikes RVM 2.4.5, a Java-in-Java VM, in its production configuration [3]. This configuration precompiles the VM methods (e.g., compiler and garbage collector) and the libraries the VM uses into a boot image. Jikes RVM contains two compilers: the *baseline compiler* and *optimizing compiler* with three optimization levels. (There is no interpreter.) When a method first executes, the baseline compiler generates assembly code (x86 in our experiments). A call-stack sampling mechanism identifies frequently executed (*hot*) methods. Based on these method sample counts, the *adaptive compilation system* then recompiles methods at progressively higher levels of optimization. Because it is sample-based, the adaptive compiler is non-deterministic.

Jikes RVM runs by default using *adaptive* methodology, which dynamically identifies frequently executed methods and recompiles them at higher optimization levels. Because it uses timer-based sampling to detect hot methods, the adaptive compiler is non-deterministic. For our performance measurements, we use *replay compilation* methodology, which forces the compiler to behave in deterministically. We use *advice files* to specify which methods to compile and at what level. For each method in the file, Jikes RVM compiles it to the specified level when it is first invoked. We use advice files that include all methods and represent the best performance of 25 adaptive runs. The advice files specify (1) the optimization level for compiling every method, (2) the dynamic call graph profile, and (3) the edge profile. Fixing these inputs, we execute two consecutive iterations of the application. During the first iteration, Jikes RVM optimizes code using the advice files. The second iteration executes only the application at a realistic mix of optimization levels. Both iterations eliminate non-determinism due to the adaptive compiler.

We use the SPEC JVM98 benchmarks [18], the DaCapo benchmarks (beta-2006-08) [8], and *ipsixql* [14]. We omit the DaCapo benchmarks *lusearch*, *pmd* and *xalan* because we could not get them to run correctly. We also include *pseudojbb* (labeled as *jbb*), a fixed-workload version of SPEC JBB2000 [19].

We perform our experiments on a 3.2 GHz Pentium 4 with hyper-threading enabled. It has a 64-byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 2GB main memory, and runs Linux 2.6.0.

Accuracy Methodology. To measure the accuracy of our technique against the perfect DCG for each application, we first generate a perfect DCG by modifying Jikes RVM call graph sampling to sample every method call (instead of skipping). We also turn off the adaptive optimizing system to eliminate non-determinism due to sampling and since call graph accuracy is not influenced by code quality. We modify the system to optimize (at level 1) every method and to inline only trivial calls. Trivial inlining in Jikes RVM inlines a callee if its size is smaller than the calling sequence. The inliner therefore never needs the frequency information for these call sites. We restrict the call graph to the application methods by excluding all call edges with both the source and target in the boot image, and calls from the boot image to the application. We include call edges into the boot image, since these represent calls to libraries that the compiler may want to inline into the application.

To measure and compare call graph accuracy, we compare the perfect DCG to the final corrected DCG generated by our approach. Because DCG clients use incomplete graphs to make optimization decisions, it would be interesting, although challenging, to compare the accuracy of the instantaneous perfect and corrected DCGs as a function of time. We follow prior work in comparing the final graphs [4] rather than a time series, and believe these results are representative of the instantaneous DCGs.

Overhead Methodology. To measure the overhead of DCG correction without including its influence on optimization decisions, we configure the call graph correction algorithms to perform correction without actually modifying DCG frequencies. We report the first iteration time because the call graph correction is triggered only at compilation time. We report the execution time as the median of 25 trials to obtain a representative result not swayed by outliers.

Performance Methodology. We use the following configuration to measure the performance of using corrected DCGs to drive inlining. We correct the DCG as the VM optimizes the application, providing a realistic measure of DCG correction's ability to affect inlining decisions. We measure application-only performance by using the second iteration time. We report the median of 25 trials.

6 Results

This section evaluates the accuracy, overhead and performance effects of the DCG correction algorithms.

We use the notation *CBS(SAMPLES, STRIDE)* to refer to an Arnold-Grove sampling configuration [4]. To compare the effect of the sampling configuration on call graph correction, we use two sampling configurations: *CBS(1,1)* and *CBS(16,3)*, 16 samples with a stride of 3. The default sampling configuration in Jikes RVM is *CBS(1,1)*, which is equivalent to the default timer-based sampling in Figure 1(a). Arnold and Grove recommend *CBS(16,3)*, which takes more samples to increase accuracy while keeping average overhead down to 1-2%.

6.1 Accuracy

We use the overlap accuracy metric from prior work to compare the accuracy of DCGs [4].

$$\begin{aligned}&\mathit{overlap}(DCG_1,\ DCG_2) = \\ &\qquad \textstyle\sum_{e \in \mathit{CallEdges}} \mathit{min}(\mathit{weight}(e, DCG_1),\ \mathit{weight}(e,\ DCG_2))\end{aligned}$$

where *CallEdges* is the intersection of the two call edge sets in DCG_1 and DCG_2 respectively, and $\mathit{weight}(e,\ DCG_i)$ is the normalized frequency for a call edge e in DCG_i. We use this function to compare the perfect DCG to other DCGs.

Figures 3 and 4 show how DCG correction boosts accuracy over the *CBS(1,1)* and *CBS(16,3)* sampling configurations. The perfect DCG is 100% (not shown). The graphs compare the perfect DCG to the base system (*No Correction*), *Static FDOM CF Correction*, *Dynamic Intraprocedural CF Correction* and *Dynamic Interprocedural CF Correction.* Arnold and Grove report an average accuracy of 50% on their benchmarks for *CBS(1,1)*, and 69% for *CBS(16,3)* for 1 to 2% overhead [4]. We show better base results here with an average accuracy of 52% for *CBS(1,1)* and 76% for *CBS(16,3).*

These results show that our correction algorithms improve over both of the sampled configurations, and that each of the algorithm components contributes to the increase in accuracy (for example, *raytrace* in Figure 3 and *jack* in Figure 4), but their importance varies with the program. FDOM and intraprocedural correction are most effective when the base graph is less accurate as in *CBS(1,1)* because they improve relative frequencies within a method. Interprocedural correction is relatively more effective using a more accurate base graph such as *CBS(16,3).* This result is intuitive; a global scheme for improving accuracy works best when its constituent components are accurate.

Figure 5 shows how the correction algorithms change the shape of the DCG for *raytrace* for *CBS(1,1)*, our best result. The vertical bar presents normalized frequencies of the 150 most frequently executed call edges from the perfect DCG. The call edges on the x-axis are grouped by their callers, and the vertical dashed lines show the group boundaries. The dots show the frequency from the sampled or corrected DCG. In the base case, call edges have different frequencies due to timing bias and sampling error. *Static FDOM CF Correction* eliminates many of these errors and improves the shape of the DCG; Figure 5(b) shows that FDOM eliminates frequency variations in call edges in the same routine. Since FDOM takes the maximum of edge weights, it raises some frequencies above their true values. *Dynamic Intraprocedural CF Correction* further improves the

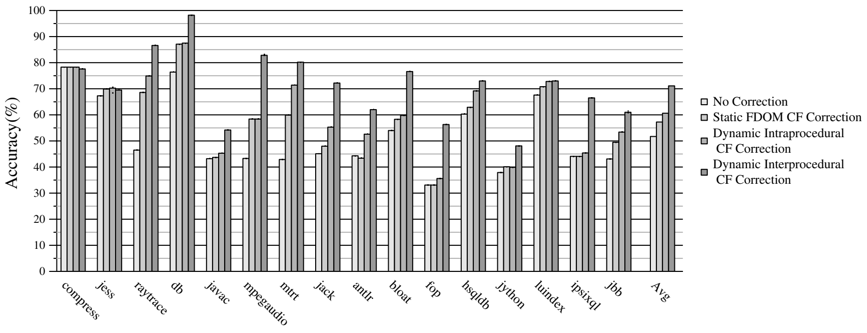


Fig. 3. Accuracy of DCG correction over the *CBS(1,1)* configuration

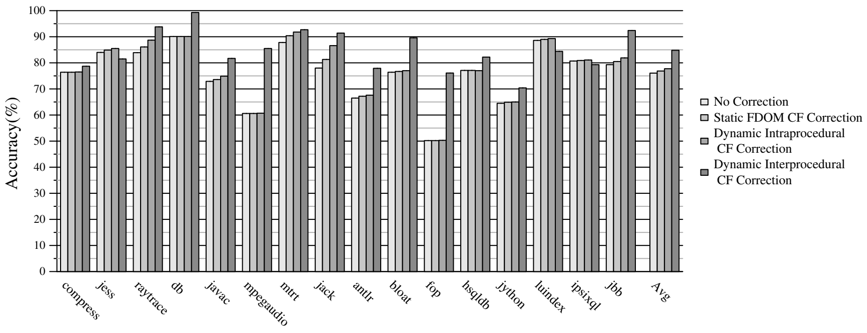
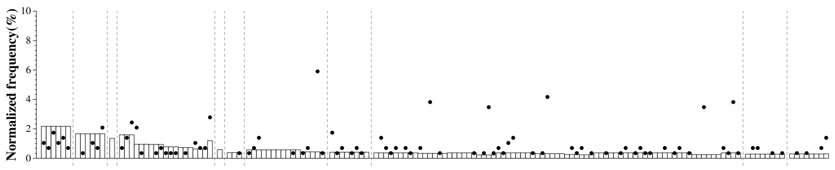


Fig. 4. Accuracy of DCG correction over the *CBS(16,3)* configuration

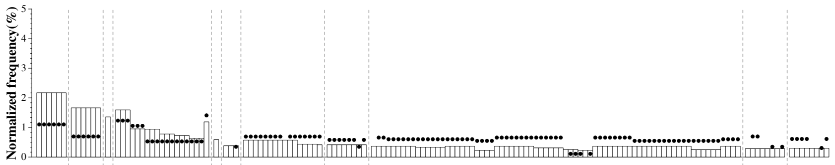
DCG because it uses fractional frequency between two call sites, while FDOM gives only relative frequency. We can see in Figure 5(c) several frequencies are now closer to their perfect values. Finally, *Interprocedural CF Correction* further improves the accuracy by eliminating interprocedural sampling bias. The most frequently executed method calls, on the left of Figure 5(d), show particular improvement.

6.2 Overhead

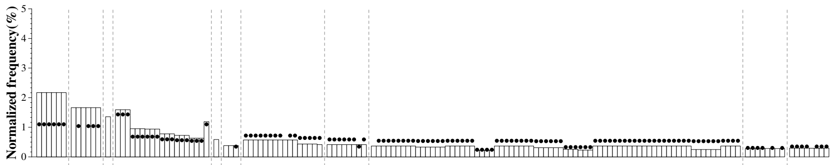
Figure 6 presents the execution overhead of DCG correction, which occurs each time the optimizing compiler recompiles a method. Correction could occur on every sample, but our approach aggregates the work and eliminates repeatedly correcting the same edges. We take the median out of 10 trials (shown as dots). *Static FDOM Correction* and *Dynamic Intraprocedural CF Correction* add no detectable overhead. The overhead of the interprocedural correction is on average 1% and at most 3% (on jython). This overhead stems from method counter instrumentation (Section 4).



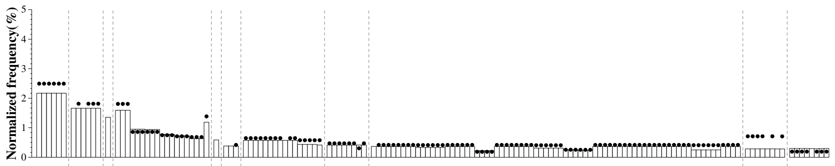
(a) No Correction



(b) Static FDOM CF Correction



(c) Dynamic Intraprocedural CF Correction



(d) Dynamic Interprocedural CF Correction

Fig. 5. Call graph frequencies for *raytrace* in *CBS(1,1)* configuration

6.3 Performance

We evaluate the costs and benefits of using DCG correction to drive one client, inlining. We use the default inlining policy with *CBS(1,1)*. Figure 7 shows application-only (second iteration) performance (median of 10 trials) with several DCG correction configurations. The graphs are normalized to the execution time without correction. We first evaluate feeding a perfect DCG to the inliner at the beginning of execution (*Perfect DCG*). The perfect DCG improves performance by a modest 2.3% on average, showing that Jikes RVM's inliner does not currently benefit significantly from high-accuracy DCGs. This result is not surprising, since the heuristics were developed together with poor-accuracy DCGs.

Static FDOM CF Correction shows the improvement from static FDOM correction, which is 1.1% on average. *Dynamic Intraprocedural CF Correction* improves performance by 1.7% on average. *Dynamic Interprocedural CF Correction*

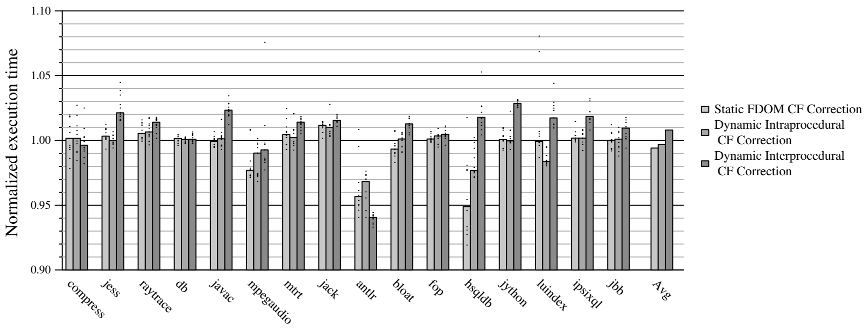


Fig. 6. The runtime overhead of call graph correction in *CBS(1,1)* configuration

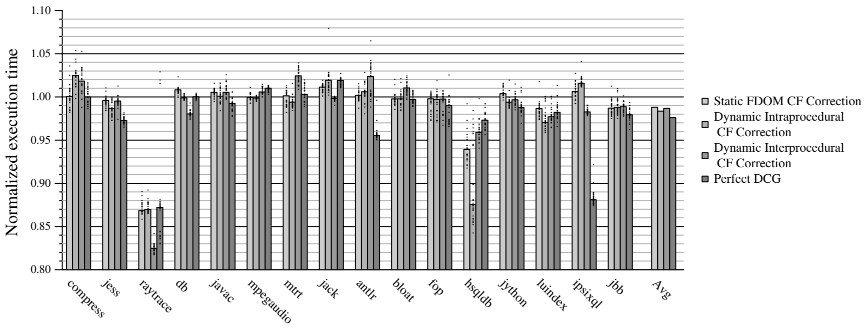


Fig. 7. The performance of correcting inlining decisions in *CBS(1,1)* configuration

shows 1.3% average improvement. However, a perfect call graph does improve two programs significantly: *raytrace* and *ipsixql* by 13% and 12% respectively, and DCG correction gains some of these improvements: 18% and 2% respectively. For *raytrace*, corrected (but imperfect) information yields better performance than perfect information. This perfect information has strictly more call edges and tends to report smaller normalized call edge frequencies as shown in Figure 5(d), leading the optimizer not to inline one important call edge. This effect occurs in *hsqldb* as well.

7 Conclusion

This paper introduces *dynamic call graph (DCG) correction*, a novel approach for increasing DCG accuracy using static and dynamic control-flow information. We introduce the *frequency dominator* (FDOM) relation to constrain and correct DCG frequencies based on control-flow relationships in the CFG. We also show how to combine these constraints with intraprocedural and interprocedural basic block profiles to correct the DCG. By adding just 1% overhead on average, we show that DCG correction increases average DCG accuracy over sampled graphs by 12% to 36% depending on the accuracy of the original. We believe DCG correction will be increasingly useful in the future as object-oriented programs become more complex and more modular.

Acknowledgments

We thank Xianglong Huang, Robin Garner, Steve Blackburn, David Grove, and Matthew Arnold for help with Jikes RVM and the benchmarks. We thank Calvin Lin, Curt Reese, Jennifer Sartor, Emmett Witchel, and the anonymous reviewers for their helpful suggestions for improving the paper.

References

1. J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Symposium on Operating Systems Principles*, pages 1–14, 1997.
2. M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. pages 52–64, Boston, MA, July 2000.
3. M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.
4. M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Symposium on Code Generation and Optimization*, pages 51–62, Mar. 2005.
5. M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM Press.
6. M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 2000.
7. T. Ball. What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, 1993.
8. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-oriented programing, systems, languages, and applications*, Portland, OR, USA, Oct. 2006. http://www.dacapobench.org.
9. M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 130–140, Barcelona, Spain, 2005.
10. K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

11. J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 93–102, New York, NY, USA, 1995. ACM Press.
12. N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Virtual Machine Research and Technology Symposium*, pages 151–162, 2004.
13. A. Hashemi, D. Kaeli, and B. Calder. Procedure mapping using static call graph estimation. In *Workshop on Interaction between Compiler and Computer Architecture*, San Antonio, TX, 1997.
14. J. Henkel. Colorado Bench. http://www-plan.cs.colorado.edu/henkel/projects/-colorado_bench.
15. B. Lee, K. Resnick, M. D. Bond, and K. S. McKinley. Correcting the Dynamic Call Graph Using Control-Flow Constraints. Technical report, University of Texas at Austin, 2006.
16. M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. mei W. Hmu. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *International Symposium on Computer Architecture*, pages 59–70, 2000.
17. M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, April 2001.
18. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.
19. Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.
20. T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 180–195, 2001.
21. T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. July 2002.
22. R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal of Computing*, 3(1):62–89, 1974.
23. Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *ACM/IEEE International Symposium on Microarchitecture*, pages 1–11, 1994.
24. X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 263–271, 2006.