# Call Graph Correction Using Control Flow Constraints

by

**Byeongcheol Lee, B.E.**

**Thesis**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Arts**

**The University of Texas at Austin**

May 2006

# Call Graph Correction Using Control Flow Constraints

**Approved by**
**Supervising Committee:**

________________________________

________________________________

Dedicated to my family

# Acknowledgments

This thesis represents joint work with Kevin Resnick, Mike Bond, and Prof. Kathryn McKinley. My contributions are the design and implementation of the call graph correction algorithms and the FDOM (frequency dominator) computation algorithms for general control flow graph. Kevin's contributions are the design and implementation of the fast FDOM computation algorithms and method counters which made possible the Static FDOM CF Correction and the Dynamic Interprocedural CF Correction. I would like to thank my supervisor Kathryn McKinley for advising this research. This accomplishment would be not be possible without the valuable advice from Mike.

I thank Prof. Calvin Lin for reviewing this thesis. I thank Xianglong Huang, Robin Garner, David Grove, and Matthew Arnold for help with Jikes RVM and the benchmarks. I thank Jennifer Sartor and Curt Reese for their helpful suggestions for improving the paper.

Living and studying in Austin has been a great pleasure due to many kind people in the department: the Speedway group members, Tawwon Cho, Hyuk Cho, Youngri Choi, Jungwoo Ha, Jaehyuk Huh, Eunjin Jung, Changkyu Kim, Doosoon Kim, Bongjune Kwon, Gene Moo Lee, Sangmin Park, Youngin Shin, Hanhee Song and Bongsoo Sohn. I am also thankful to friends from Korea for their support and friendship.

I thank Korean Ministry of Information and Communication for financial support toward my M.A. degree. Also, I thank Verizon for the fellowship that allowed me to continue this research 2005 Summer.

Lastly and most importantly I would like to thank my parents, my dear brother and sister for their selfless love and support.

Byeongcheol Lee

*The University of Texas at Austin*
*May 2006*

# Call Graph Correction Using Control Flow Constraints

Byeongcheol Lee, M.A.
The University of Texas at Austin, 2006

Supervisor: Kathryn S. McKinley

Dynamic optimizers for object-oriented languages collect a variety of profile data to drive optimization decisions. In particular, the *dynamic call graph* (DCG) informs key structural optimizations such as which methods to optimize and how to optimize them. Unfortunately, current low-overhead call-stack hardware and software sampling methods are subject to sampling bias, which loses accuracy of 40 to 50% when compared with a perfect call graph.

This paper introduces *DCG correction*, a novel approach that uses static and dynamic *control-flow graphs* (CFGs) to improve DCG accuracy. We introduce the static *frequency dominator* (FDOM) relation, which extends the dominator relation on the CFG to capture relative execution frequencies and expose static constraints on DCG edges, which we use to correct DCG edge frequencies. Using conservation of flow principles, we further show how to use dynamic CFG basic block profiles to correct DCG edge frequencies intraprocedurally and interprocedurally.

We implement and evaluate DCG correction in Jikes RVM on the SPEC JVM98 and DaCapo benchmarks. Default DCG sampling attains an average accuracy of 52–59% compared with perfect, whereas FDOM correction improves average accuracy to 64–68%, while adding 0.2% average overhead. The dynamic correction raises accuracy to 85% on average, while adding 1.2% average overhead. We then provide dynamically corrected DCGs to the inliner with mixed results—1% average degradations and improvements across a variety of configurations. However, prior work shows that increased DCG accuracy in production VMs has benefits. We believe that high-accuracy DCGs will become more important in the future as the complexity and modularity of object-oriented programs increases.

# Contents

# Chapter 1

# Introduction

Dynamic optimizers for object-oriented languages ameliorate the overhead of online compilation by detecting and optimizing the most frequently executed *hot* methods. To detect hot methods and call edges, dynamic optimizers use hardware or software call-stack sampling to build a *dynamic call graph* (DCG) and selectively target optimizations. A key optimization for these systems is method inlining because well designed object-oriented programs achieve reusability, reliability, and maintainability by decomposing functionality into many small methods, and virtual dispatch obscures the hot target. Inlining exposes opportunities for other optimizations and decreases call overhead, but inlining must be applied judiciously because it also increases code size and compilation time. Good decisions depend on the accuracy of the DCG.

Dynamic optimizers trade accuracy for low overhead by using sampling to collect the DCG. Hardware-based sampling lowers overhead by examining hardware performance counters [16] instead of the call stack but gives up portability. Software-based sampling periodically examines the call stack and updates the DCG [3, 13, 19, 22]. All DCG sampling approaches suffer from sampling error, and approaches that use a timer to take samples suffer from *timing bias* [3] due to a mismatch

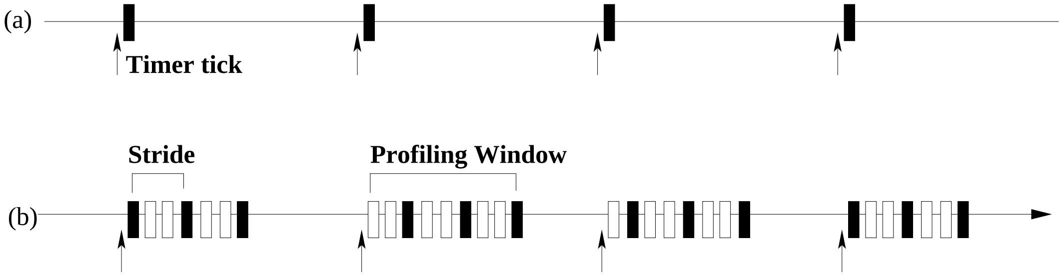


Figure 1.1: Timer-based sampling. Filled boxes are taken samples; unfilled boxes are skipped samples. (a) One sample per timer tick. (b) Counter-based Sampling (CBS) takes multiple samples per timer tick and strides between samples.

between the sampling mechanism and the desired information: the timer based sampling reports where time is spent, but DCG is *frequency* information. Arnold and Grove [3] were the first to measure and note that the DCG is not very accurate. They introduce counter-based sampling (CBS), which takes multiple samples per timer tick and strides over some samples to reduce sampling error and timing bias. The overhead of CBS is directly proportional to the number of samples, and they recommend configurations that achieve an average of 58% accuracy of a perfect call graph with 0.1% additional overhead in Jikes RVM and J9.

This paper presents new *DCG correction* algorithms to improve DCG accuracy with extremely low overhead (1% on average). The key insight is that a program's static and dynamic *control-flow graph* (CFG) constrains possible DCG frequency values. For example, two calls must execute the same number of times if their basic blocks execute the same number of times. To leverage this insight, we introduce the static *frequency dominator* (FDOM) relation, which extends the dominator and strong region relations as follows: given statements $x$ and $y$, $x$ FDOM $y$ if and only if $x$ executes at least as many times as $y$. DCG correction applies FDOM constraints to the sampled DCG to improve its accuracy. For example, given two call sites $cs_1$ and $cs_2$, if $cs_1$ FDOM $cs_2$ and in the sampled DCG, $|cs_1| < |cs_2|$, then

DCG correction sets $|cs_1|$ to $|cs_2|$. Because these relationships are static, the correction algorithm computes them once and then periodically combines them with the DCG.

DCG correction uses dynamic *basic block profiles* to improve DCG accuracy. Most dynamic optimizers collect high-accuracy control-flow profiles such as basic block and edge profiles to make better optimization decisions [1, 2, 4, 6, 9, 13, 18, 19]. We show how to combine these constraints and method counter profiling to further improve the accuracy of the DCG. This correction requires a single pass over the basic block profile, which we perform periodically.

We evaluate DCG correction in the Jikes RVM on the SPEC JVM98 and DaCapo benchmarks. We compare our approach to the CBS sampling configurations recommended by Arnold and Grove [3] that are now standard in Jikes RVM. For our benchmarks, CBS attains 52 to 60% average accuracy compared to a perfect call graph. FDOM constraints improve DCG accuracy to 64% to 68% on average, while adding less than 0.3% average overhead. Static and dynamic control-flow information together raise DCG accuracy to 85% on average, while adding just 1% overhead.

We evaluate the performance impact of corrected DCGs using inlining in Jikes RVM with its adaptive hotspot compiler that periodically recompiles and inlines hot methods. We add to this system DCG correction immediately before the system recompiles. We measure steady-state performance and find that DCG correction does change optimization decisions and performance but that average performance does not change. However, we find modest 1% average improvements when we provide an initial profile with a perfect call graph, and prior work shows that an accurate DCG can improve performance in other virtual machines [3]. These and other results [10] suggest that there are inlining policies that could be tuned to benefit from DCG correction. The main contributions of this paper are:

- Call graph correction algorithms using control-flow graph consistency constraints.
- The frequency dominator relation and its computation.
- Implementation and evaluation of call graph correction schemes.
- A technique that yields highly accurate call graphs with very little overhead that can easily be added to many virtual machines.

# Chapter 2

# Background and Related Work

This section presents background material and compares dynamic call graph (DCG) correction to previous work. We first discuss how dynamic optimizers collect a high-accuracy DCG that represents the relative frequencies of direct and virtual method calls. We then compare the frequency dominator relation to previous work. Finally we compare DCG correction to previous work that uses control-flow information to generate the DCG.

## 2.1 Using and Collecting Dynamic Call Graphs

Dynamic optimizers must balance increases in compilation time and code space costs with application improvement. They use call graph and control-flow information to help select optimization candidates, tailor optimizations to a specific run, and balance compile and application time. For example, while *static inlining* makes inlining decisions based on trivial criteria in dynamic optimizers (e.g., always inline a direct method call that is smaller than the calling sequence), *dynamic inlining* decisions are based on the DCG and method sizes.

Dynamic optimizers can profile every call in order to collect a *perfect DCG*,

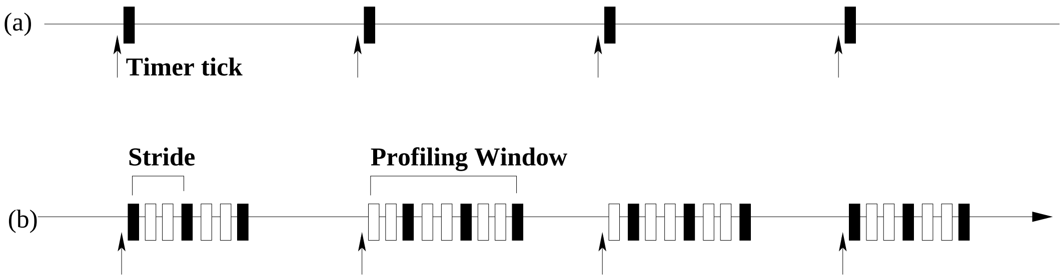


Figure 2.1: Timer-based sampling. Filled boxes are taken samples; unfilled boxes are skipped samples. (a) One sample per timer tick. (b) Counter-based Sampling (CBS) takes multiple samples per timer tick and strides between samples.

but this overhead is too high [3, 14]. Some dynamic optimizers profile calls fully for some period of time and then turn off profiling to keep overhead down. For example, HotSpot adds call graph instrumentation only for unoptimized code [19]. Suganama et al. use *code patching* to insert call instrumentation, collect call samples for a period of time, and then remove the instrumentation [22]. These *one-time profiling* approaches keep overhead down but may lose accuracy if behavior changes. Jikes RVM uses a similar approach for computing the control-flow edge profiles only for unoptimized code. DCG correction can improve the accuracy of an out-of-date DCG using up-to-date control-flow information, although this paper does not specifically evaluate DCG correction for this purpose.

Many current dynamic optimizers use software sampling to profile calls and identify hot methods while keeping overhead low [3, 5, 13, 28]. 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 causes the next *yield-point* in the code to examine the call stack and update the DCG [5, 13]. Figure 2.1(a) illustrates basic 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 the results and possibly leads to poor inlining decisions. They present *counter-based sampling* (CBS), which takes multiple samples per timer tick and *strides* to skip some samples in the profiling window and thus reduces timing bias. Figure 2.1(b) shows CBS configured to take 3 samples for each timer tick and to stride by 3 (take every third sample). By widening the profiling windows, CBS improves DCG accuracy, but the large profiling windows also increase profiling overhead. For example, DCG correction achieves 85% accuracy with 1% overhead whereas CBS's overhead is 6 to 20% in J9 to achieve the same accuracy. In Jikes RVM, achieving 75% accuracy comes at an overhead of 14% or more (85% accuracy hits some pathological case, costing 1000% overhead).

Other dynamic optimizers periodically examine hardware performance counters such as those in the Itanium [16] to update the DCG. All sampling approaches suffer from sampling error, and timer-based sampling approaches suffer from timing bias as well. DCG correction can improve the accuracy without introducing significant overhead of any DCG collected by sampling.

## 2.2 The Dominator Relation and Strong Regions

This paper introduces the *frequency dominator* (FDOM) relation, which extends the *dominator* and *strong regions* relations [7]. Prosser first introduced dominators, which have a rich history [11, 20, 24, 25]. 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 which frequency dominates $x$, on the other hand, is the subset which 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 goes further and also finds vertices $x$ and $y$ where $y$ must execute at least as many times as $x$.

## 2.3 Constructing the DCG using Control-Flow Information

This section compares DCG correction to previous work that uses control-flow information to construct the DCG. Hashemi et al. use static heuristics to construct an estimated call frequency profile [15]. Wu and Larus construct an estimated edge profile, which they use to construct an estimated call frequency profile [29]. These approaches rely solely on control-flow information to estimate call frequencies, whereas DCG correction starts with an inaccurate DCG and applies control-flow constraints to improve its accuracy. Also, these approaches use static *heuristics* to estimate frequencies, while DCG correction uses static *constraints* and dynamic profile information. They report high accuracy but use an accuracy metric that considers only the relative rank of call sites, while our overlap accuracy metric uses call edge frequencies. They construct profiles for C programs, while we target Java, which has richer DCGs and mulitple call targets for a call site because of virtual dispatch.

# Chapter 3

# 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) relation that captures the relative frequencies of program statements based on control-flow graph constraints. We show how to apply these static constraints to improve the accuracy of the DCG, and how to combine them with dynamic CFG frequencies to further improve the DCG. In particular, we show how DCG correction updates the relative frequencies of call sites (which we refer to as the call sites' $OUTFLOW$) in the DCG to reflect the relative execution frequencies of basic blocks from the CFG.

## 3.1 Terminology

A *control-flow graph* (CFG) represents the *static* intraprocedural control flow in a method, and consists of basic blocks and edges between them. Figure 3.1 shows an example control-flow graph $CFG_p$ that consists of basic blocks $ENTRY$, $a$, $b$, $c$, $d$, $e$,

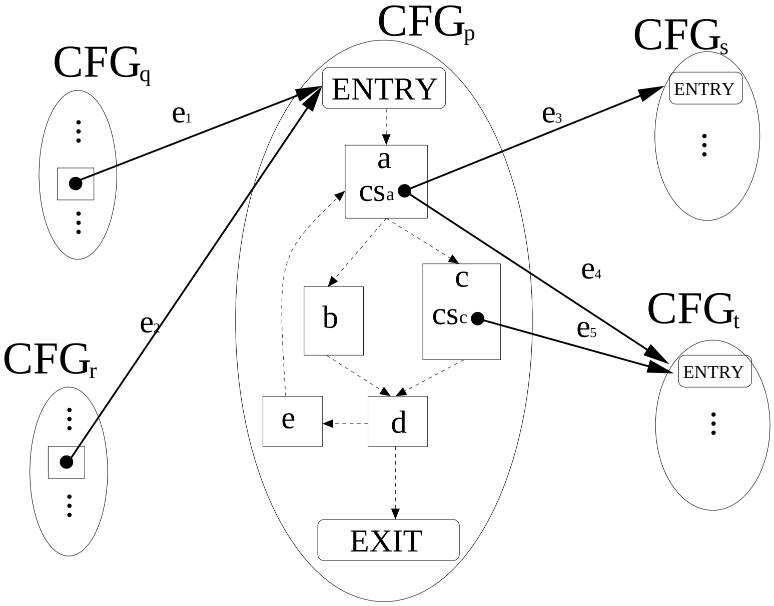


Figure 3.1: Example dynamic call graph (DCG) and control flow graphs (CFGs).

and *EXIT*, as well as edges between them. A *basic block profile* gives the *dynamic* execution frequency of each basic block in the program, from some execution.

A *call edge* represents a method call, and consists of a *call site* and a *callee*. An example call edge in Figure 3.1 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 3.1. 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 3.1.

We define the *INFLOW* of a method $m$ as the total *flow* (execution frequency) coming into $m$:

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

where $f(e)$ is the execution frequency of call edge $e$. $\mathit{INFLOW}(m)$ in a perfectly accurate DCG is thus equal to the number of times $m$ executes.

We define the $\mathit{OUTFLOW}$ of a call site $cs$ as the total flow going out of $cs$:

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

$\mathit{OUTFLOW}(cs)$ in a perfectly accurate DCG is thus equal to the number of times $cs$ executes.

Because the collected DCG is not accurate due to sampling error and timing bias, the DCG reports inaccurate $\mathit{OUTFLOW}$ and $\mathit{INFLOW}$ values. DCG correction's goal is to correct the $\mathit{OUTFLOW}$ of call sites using constraints provided by static and dynamic control-flow information (doing so indirectly corrects method $\mathit{INFLOW}$ as well).

DCG correction does not correct the relative execution frequencies of multiple call edges coming out of the same call site. Instead, it uses the existing relative call edge frequencies from the uncorrected DCG, and scales them to update call sites' $\mathit{OUTFLOW}$ in the corrected DFG. So for example in Figure 3.1, if sampling data yields flow such that $f(e_5) > f(e_3) + f(e_4)$, DCG correction increases the total flow of $cs_a$, and scales it by their current relative frequencies.

## 3.2 The Frequency Dominator (FDOM) Relation

This section introduces the *frequency dominator* (FDOM) relation, a major contribution of this paper. FDOM is a static property of CFGs that represents constraints on program statements' relative execution frequencies. We show two constraints (theorems) it provides from the CFG on the DFG, and we prove these relations in the appendix.

**DEFINITION 1.** *Frequency Dominator (FDOM). Given statements $s$ and $t$ in the same method, $s$ FDOM $t$ if and only if for every possible path through the method, $s$ must execute at least as many times as $t$. We also define FDOM(t) $\equiv \{s \mid s$ FDOM $t\}$.*

Like the dominator relation, FDOM is reflexive and transitive.

**THEOREM 1.** ***FDOM OUTFLOW Constraint:*** *Given method $m$ and two call sites $cs_1$ and $cs_2$ in $m$, if $cs_1$ FDOM $cs_2$,*

$$\text{OUTFLOW}(\text{cs}_1) \geq \text{OUTFLOW}(\text{cs}_2) \tag{3.1}$$

Intuitively, the *OUTFLOW* constraint tells us that flow on two call edges is related if they are related by FDOM. For example, in Figure 3.1, $\text{cs}_a$ FDOM $\text{cs}_c$ and thus each time $\text{cs}_c$ executes, $\text{cs}_a$ must have executed.

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

$$\text{INFLOW}(m) \leq \text{OUTFLOW}(\text{cs}) \tag{3.2}$$

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.

## 3.3 Static FDOM Correction

This section shows how to use the *INFLOW* and *OUTFLOW* constraints to correct DCG frequencies. (The next section shows how to use dynamic CFG basic block profiles to further correct *INFLOW* and *OUTFLOW*.)

Figure 3.2 applies the *FDOM OUTFLOW* constraint to a sampled DCG. The algorithm *FDOMOutflowConstraint* 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 so the sum remains the same as before.

Similarly, Figure 3.3 applies the *FDOM INFLOW* constraint to a sampled DCG. For every call site *cs* that must execute each time the method executes (*cs FDOM ENTRY*), *FDOMInflowConstraint* sets *OUTFLOW*($cs$) to *INFLOW*($m$) if *OUTFLOW*($cs$) $<$ *INFLOW*($m$).

## 3.4 Dynamic Basic Block Profile Constraints

This section describes constraints on DCG frequencies provided by basic block profiles, and the following section shows how to correct the DCG with them. The appendix proves these relations.

The *Dynamic OUTFLOW* constraint says that the ratio between the execution frequencies of two call sites specified by the basic block profile can be applied to the *OUTFLOW* of these two call sites:

**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,*

$$\frac{\text{OUTFLOW}(\text{cs}_1)}{\text{OUTFLOW}(\text{cs}_2)} = \frac{f_{\text{bprof}}(\text{cs}_1)}{f_{\text{bprof}}(\text{cs}_2)} \tag{3.3}$$

```
procedure FDOMOutflowCorrection
input:
  $CALLSITES$: a set of call sites to be corrected
  $FDOM(cs)$: a set of call sites that frequency-dominate $cs$
  $f_{sample}$(e): a function that returns the frequency of call edge $e$ from sampling
  $f_{sample}$(cs): a function that returns the frequency sum of call edges in $OutEdge(cs)$
                from sampling
output:
  $f_{corrected}$(e): corrected frequency of call edge $e$

{STEP1: Initialize outflow for each call site and its sum.}
$sum_{old} \leftarrow 0$
for all $cs \in CALLSITES$ do
    $Outflow(cs) \leftarrow f_{sample}(cs)$
    $sum_{old} \leftarrow sum_{old} + Outflow(cs)$
end for
{STEP2: satisfy FDOM Outflow constraint.}
$sum_{new} \leftarrow sum_{old}$
for all $cs_y \in CALLSITES$ do
    for all $cs_x \in FDOM(cs_y)$ do
        {constraint: $OUTFLOW(cs_x) \geq OUTFLOW(cs_y)$}
        $outflow_{old} \leftarrow Outflow(cs_x)$
        $Outflow(cs_x) \leftarrow Max(Outflow(cs_x), Outflow(cs_y))$
        $diff = Outflow(cs_x) - outflow_{old}$
        $sum_{new} \leftarrow sum_{new} + diff$
    end for
end for
{STEP3: Use new outflow to derive the corrected frequency.}
$scale \leftarrow sum_{old}/sum_{new}$
for all $cs \in CALLSITES$ do
    for all $e \in OutEdges(cs)$ do
        $fraction = f_{sample}(e)/f_{sample}(cs)$
        {Preserve the call target fraction and the frequency sum.}
        $f_{corrected}(e) \leftarrow Outflow(cs) \times scale \times fraction$
    end for
end for
```

Figure 3.2: DCG Correction with FDOM OUTFLOW Constraints

```
procedure FDOMInflowCorrection
input:
  $p$: a procedure to apply FDOM inflow constraint
  $FDOM(ENTRY)$: a set of call sites that FDOMs the entry of the procedure P
  $f_{sample}(e)$: a function that returns the frequency of call edge $e$ from sampling
  $f_{sample}(cs)$: a function that returns the frequency sum of call edges in $OutEdge(cs)$
                    from sampling
output:
  $f_{corrected}(e)$: corrected frequency of call edge $e$

{STEP1: Initialize outflow for each call site and its sum.}
$inflow \leftarrow 0$
for all $e \in InEdges(p)$ do
    $inflow \leftarrow inflow + f_{sample}(e)$
end for
{STEP2: satisfy FDOM inflow constraint.}
for all $cs_x \in FDOM(ENTRY)$ do
    {constraint: $\text{OUTFLOW}(\text{cs}_x) \geq \text{INFLOW}(p)$}
    $Outflow(cs_x) \leftarrow Max(Outflow(cs_x), inflow))$
end for
{STEP3: Use new outflow to derive the corrected frequency.}
for all $cs \in FDOM(ENTRY)$ do
    for all $e \in OutEdges(cs)$ do
        $fraction = f_{sample}(e)/f_{sample}(cs)$
        {Preserve the call target fraction}
        $f_{corrected}(e) \leftarrow Outflow(cs) \times fraction$
    end for
end for
```

Figure 3.3: DCG Correction with FDOM INFLOW Constraints

The *Dynamic OUTFLOW* constraint can be applied to two call sites in different methods if basic block frequencies from different methods are accurate relative to each other (i.e., if the basic block profiles have *interprocedural accuracy*). In our implementation, basic block profiles do *not* have interprocedural accuracy. Thus, we do not apply the *Dynamic OUTFLOW* constraint to call sites in different methods. If basic block profiles do not have interprocedural accuracy, then the *Dynamic OUTFLOW* constraint provides no help for correcting the *OUTFLOW* of call sites in methods with a single basic block. We experiment with using low-overhead method invocation counters to give basic block profiles interprocedural accuracy, and in this case we apply *Dynamic OUTFLOW* to call sites in different methods (Section 3.6).

The *Dynamic INFLOW* constraint says that the call edge flow (frequency) coming into a method with a single basic block constrains the flow leaving any call site in the method:

**THEOREM 4.** ***Dynamic INFLOW Constraint:*** *Given a method $m$ with a single basic block and a call site* cs *in $m$,*

$$\mathrm{INFLOW}(m) = \mathrm{OUTFLOW}(cs) \tag{3.4}$$

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 the single basic block (when basic block profiles do not have interprocedural accuracy). The *Dynamic INFLOW* constraint uses the total flow (frequency) coming into the method to constrain call sites' *OUTFLOW*.

## 3.5 Dynamic Basic Block Profile Correction

```
procedure DynamicOutflowCorrection
input:
  $CALLSITES$: a set of call sites
  $f_{bprof}(cs)$: a function that returns the frequency of the call site $cs$ from basic block profiles
  $f_{sample}(e)$: a function that returns the frequency of call edge $e$ from sampling
output:
  $f_{corrected}(e)$: a function that returns the corrected frequency for the call edge $e$

{STEP1: Iterate call sites to find scale factor from basic block profile count to the sample count.}
$sum_{sample} \leftarrow 0$
$sum_{bprof} \leftarrow 0$
for all $cs \in CALLSITES$ do
  $sum_{sample} \leftarrow sum_{sample} + f_{sample}(cs)$
  $sum_{bprof} \leftarrow sum_{bprof} + f_{bprof}(cs)$
end for
$scale \leftarrow sum_{sample} / sum_{bprof}$
{STEP2: assign corrected call edgefrequency}
for all $cs \in CALLSITES$ do
  for all $e \in OutEdges(cs)$ do
    $fraction = f_{sample}(e) / f_{sample}(cs)$
    {constraint: $OUTFLOW(cs)/f_{bprof}(cs)$ is constant }
    {Preserve the call target fraction and the frequency sum}
    $f_{corrected}(e) \leftarrow f_{bprof}(cs) \times scale \times fraction$
  end for
end for
```

Figure 3.4: DCG Correction with Dynamic OUTFLOW Constraints

```
procedure DynamicInflowCorrection
input:
  $p$: a single basic block procedure
  $f_{sample}(e)$: a function that returns the frequency of call edge $e$ from sampling
  $f_{sample}(cs)$: a function that returns the frequency sum of call edges in $OutEdge(cs)$
                    from sampling
output:
  $f_{corrected}(e)$: corrected frequency of call edge $e$

$CALLSITES \leftarrow$ getCallSitessInsideProcedure($p$)
{STEP1:Compute maxflow for the procedure $p$.}
$inflow \leftarrow 0$
for all $e \in InEdges(p)$ do
   $inflow \leftarrow inflow + f_{sample}(e)$
end for
$maxoutflow \leftarrow \max_{cs \in CALLSITES} f_{sample}(cs)$
$maxflow \leftarrow \max(inflow, maxoutflow)$
{STEP2: assign corrected frequency}
for all $cs \in CALLSITES$ do
   for all $e \in OutEdges(cs)$ do
      $fraction \leftarrow f_{sample}(e)/f_{sample}(cs)$
      {constraint: INFLOW(p) = OUTFLOW(cs).}
      $f_{corrected}(e) \leftarrow maxflow \times fraction$
   end for
end for
```

Figure 3.5: DCG Correction with Dynamic INFLOW Constraints

| Correction algorithm | Correction range | Algorithms |
|---|---|---|
| Static FDOM CF Correction | Call sites within a method to be optimized | Figures 3.2 and 3.3 |
| Dynamic Intraprocedural CF Correction | Call sites within a method to be optimized | Figures 3.4 and 3.5 |
| Dynamic Interprocedural CF Correction | All call sites in the DCG | Figures 3.4 and 3.5 |

Figure 3.6: Call Graph Correction Implementations

Figure 3.4 presents the algorithm for applying the *Dynamic* $OUTFLOW$ constraint. *DynamicOutflowCorrection* 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. This scaling helps to maintain the frequencies due to sampling across disparate parts of the DCG.

Figure 3.5 presents the algorithm for applying the *Dynamic INFLOW* constraint to the DCG. 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.

## 3.6 Implementing Online DCG Correction

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

We minimize the call graph correction overhead by limiting the frequency and scope of DCG correction. We limit DCG correction's frequency by delaying DCG correction until the optimizing compiler requests DCG information. The correction overhead is thus proportional to the number of methods optimized during an execution. Correction overhead is thus naturally minimized as a result of the dynamic optimizers' selective choices about when and which methods to recompile.

We limit the scope of DCG corretion 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.

Figure 3.6 summarizes the correction algorithms and the correction range. All the correction 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 one limits the call site set to be within a procedural boundary, and the second one corrects all the reachable methods. Since many dynamic compilation systems support only high precision intraprocedural basic profiles, the first configuration indicates how much DCG correction would benefit these systems.

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

## 3.7 Call Graph Accuracy and Correction Upper Bound

We use the overlap accuracy metric from prior work to compare the accuracy of DCGs [3].

$$overlap(DCG_1,\ DCG_2) =$$

$$\textstyle\sum_{e \in CallEdges} min(weight(e, DCG_1),\ weight(e,\ DCG_2))$$

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

$$accuracy(DCG) = overlap(DCG,\ DCG_{perfect})$$

We compute an upper bound for our correction schemes based on the call edges in the sampled *DCG*. For this accuracy upper bound, we consider $DCG_{sample}$, a sampled DCG, and $DCG_{perfect}$. Given $DCG_{perfect}$ and $DCG_{sample}$, we define $DCG_{bound}$ by taking the call nodes and edges from the $DCG_{sample}$ and by taking the edge frequencies from $DCG_{perfect}$. Then, the frequencies of call edges in $DCG_{bound}$ are completely unbiased, but $DCG_{bound}$ only contains the methods and calls edges in the sampled graph. Therefore, $DCG_{bound}$ is the best that we can hope for correction to achieve.

Theorem 5 shows that the every call graph from our correction schemes can not achieve better accuracy than that of $DCG_{bound}$. $DCG_{bound}$ characterizes theoretical accuracy upper bound for correction algorithms which preserve the call edge set in $DCG_{sample}$. The correction algorithm can further improve over the accuracy upper bound by adding more call edges to the input DCG, but we only limit our interest to correction algorithm that does not infer new call edges. Also, $DCG_{bound}$ shows the tightest accuracy upper bound for these correction algorithms.

**THEOREM 5.** ***Call Graph Correction Upper Bound:*** *Given* $\mathrm{DCG}_{sample}$, $\mathrm{DCG}_{\mathrm{bound}}$ *and* $\mathrm{DCG}_{\mathrm{perfect}}$, *for every* $\mathrm{DCG}_{corrected}$ *with the same set of call edges as the* $\mathrm{DCG}_{sample}$ *and* $\mathrm{DCG}_{\mathrm{bound}}$,

$$accuracy(DCG_{corrected}) \leq accuracy(DCG_{\text{bound}}) \tag{3.5}$$

*Proof.* For $DCG_x$, let $E_x$ be the set of call edge and $f_x(e)$ be the call edge frequency of a call edge $e$ in the $DCG_x$. For brevity, $f_x(E)$ denotes $\sum_{e \in E} f_x(e)$. Consider $DCG_{sample}$, $DCG_{perfect}$ and $DCG_{bound}$, and note that $E_{sample}$ equals to $E_{bound}$, and for every call edge $e \in E_{bound}$, $f_{bound}(e)$ equals to $f_{perfect}(e)$. Because $DCG_{bound}$ is subgraph of $DCG_{perfect}$, and call edge frequency in these call graphs are non-negative, $f_{perfect}(E_{bound})$ is less than or equal to $f_{perfect}(E_{perfect})$. Note that for call edge $e$ in $DCG_x$, $weight(e, DCG_x)$ is $f_x(e)/f_x(E_x) \times 100$. For every call edge $e \in E_{bound}$, the following holds: $weight(e, DCG_{bound}) = \frac{f_{bound}(e)}{f_{bound}(E_{bound})} \times 100 = \frac{f_{perfect}(e)}{f_{perfect}(E_{bound})} \times 100$ $\leq \frac{f_{perfect}(e)}{f_{perfect}(E_{perfect})} \times 100 = weight(e, DCG_{perfect})$. Therefore, for every call edge $e \in E_{bound}$, the following holds:

$$min\{weight(e, DCG_{bound}), weight(e, DCG_{perfect})\} = weight(e, DCG_{perfect}) \tag{3.6}$$

Let $DCG_{corrected}$ be the resulting call graph from correction that preserves the set of call edges such that $E_{corrected}$, $E_{bound}$ and $E_{sample}$ equal to each other. The following formulas hold:

$accuracy(DCG_{corrected}) = overlap(DCG_{corrected}, DCG_{perfect})$
$= \sum_{e \in E_{corrected}} min\{weight(e, DCG_{corrected}), weight(e, DCG_{perfect})\}$
$\leq \sum_{e \in E_{corrected}} weight(e, DCG_{perfect})$ (by definition of min)
$= \sum_{e \in E_{bound}} min\{weight(e, DCG_{bound}), weight(e, DCG_{perfect})\}$ (by Equlation (3.6))
$= accuracy(DCG_{bound})$ □

# Chapter 4

# Computing FDOM

This section presents our algorithm for computing the *frequency dominator* (FDOM) relation for statements in a method. We first show a correlation between *simple cycles* and FDOM that applies to both *irreducible* and *reducible* graphs. We then present our algorithms for computing FDOM. An algorithm for both types of graphs is simple but slow, and one for only reducible graph is the opposite. Throughout this section, we use $s$ in place of $ENTRY$ and $e$ in place of $EXIT$ for brevity. Both algorithms assume the existence of a back edge from $e$ to $s$. This edge simplifies the analysis but does not affect the FDOM relation, since following this edge is equivalent to executing the method again.

## 4.1 General Control Flow Graphs

In this section, we show that the FDOM relation relates to *simple cycles* for general CFGs (both irreducible and reducible). Then, we present a simple exhaustive search algorithm to compute FDOM by exploiting this relation. Lemma 1 shows that FDOM can be computed by considering *cycles* in the CFG.

**LEMMA 1.** *Given vertices $x$ and $y$ in* CFG*,*

*$x$ FDOM $y$*

*if and only if*

*$x$ is contained in every cycle containing $y$*

*Proof.* Show both forward and backward directions.

($\Rightarrow$) (by contradiction) Suppose there is a cycle that contains $y$ but not $x$. Let this cycle be $c_y = \langle a, \ldots, y, \ldots, a \rangle$. From the definition of CFG, there is one path from the CFG entry, $s$, to $a$ and another path from $a$ to the CFG exit, $e$. By concatenating these three paths, we can build a path from $s$ to $e$, $c'_y = \langle s, \ldots, a, \ldots, y, \ldots, a, \ldots, e \rangle$. If $x$ is not in $c'_y$ then $y$ is already executed more times than $x$, a contradiction. So let $x$ in some execution path that includes $c'_y$ be executed $n$ times, then we can repeat $c_y$ $n+1$ times, resulting in $y$ being executed more than $x$, a contradiction.

($\Leftarrow$) (by contrapositive) Suppose there is a path $p$ from $s$ to $e$ where the number of executions of $y$ exceeds that of $x$. For the number of $y$ to exceed $x$, $p$ must contain at least one $y$. If the number of executions of $y$ in the path is $n$, then path $p$ should be of the form, $p = \langle s, \ldots, y_1, \ldots, y_2, \ldots, y_n, \ldots, e \rangle$. The path, $p$, can be divided into $(n+1)$ subpaths:$\langle s, \ldots, y_1 \rangle$, $\langle y_1, \ldots, y_2 \rangle$, ..., $\langle y_n, \ldots, e \rangle$. Because there are fewer executions of $x$ than $y$ in $p$, $x$ appears in all of the subpaths less than or equal to $(n-1)$ times. From the pigeonhole principle, there exist at least two $x$-free subpaths. If one of the two $x$-free subpaths is $\langle y_i, \ldots, y_{i+1} \rangle$, this $x$-free subpath contradicts our assumption that there exists no cycle containing $y$ and not $x$. If two subpaths are $\langle s, \ldots, y_1 \rangle$ and $\langle y_n, \ldots, e \rangle$, another $x$-free cycle $\langle y_n, \ldots, e, s, \ldots, y_1 \rangle$ can be constructed since we have a backedge from $e$ to $s$, again a contradiction. ☐

Lemma 1 states that FDOM can be computed by considering every cycle in the CFG. However, it is impractical to check if $x$ is contained in every cycle that contains $y$ because the number of cycles may be unbounded. The number of simple cycles in a method is bounded, and Lemma 2 shows that *simple cycles* are sufficient.

**LEMMA 2.** *Given vertices $x$ and $y$ in* CFG*,*

*$x$ is in every cycle containing $y$*
*if and only if*
*$x$ is in every simple cycle containing $y$*

*Proof.* Forward direction is trivial because every simple cycle is a cycle. To show the backward direction:

($\Rightarrow$)(by contradiction) Suppose that $x$ is in every simple cycle containing y, but let there exist $x$-free cycle $c$ that contains $y$, $c = \langle a, \ldots, y, \ldots, a\rangle$ or $c = \langle y, \ldots, y\rangle$. By our assumption, $c$ cannot be simple, so there must exist some element in $c$ other than the beginning or end. There exists a cycle, therefore, in $c$ that is from $\langle w, \ldots, w\rangle$ that is simple, which we will call $c'$. Since by our assumption, $x$ is in every simple cycle containing $y$, if there exists a $y$ in $c'$, then there also exists an $x$, and since $c'$ is simple, there can exist at most one of each. Therefore if $c$ is a valid cycle including $c'$, then another valid cycle is $c$ with $c'$ replaced with the single element $w$. Further, this new $c$ still has the property that there exists an $x$-free cycle that contains $y$. Since $c$ is of finite length, this process can be continued until there are no simple cycles left in $c$. Note that $c$ still contains an $x$-free path that contains $y$, but now $c$ is simple too, a contradiction. This proof holds for the entire program because there exists a back edge, by assumption, from $e$ to $s$, and hence it is a cycle.

□

**THEOREM 6.** *Given vertices $x$ and $y$ in* CFG*,*

*$x$ FDOM $y$*
*if only if*
*$x$ is in every simple cycle containing $y$.*

*Proof.* From Lemma 1 and Lemma 2. ☐

Theorem 6 provides a method for computing FDOM. Let ***ToSet(p)*** be a function from a path $p$ to the set of all elements in $p$. For example, for the $p = \langle 1, 4, 2, 5, 1\rangle$, $ToSet(p)$ is $\{1, 2, 4, 5\}$. The following equation holds directly from Theorem 6:

$$FDOM(y) = \bigcap_{p\text{: simple cycle containing } y} ToSet(p) \tag{4.1}$$

Figure 4.1 and 4.2 illustrate an irreducible control flow graph and its frequency-dominators. The second column of Figure 4.2 shows the set of all the simple cycles that contain each $y$ in the first column. Equation (4.1) validates that the frequency dominator for each $y$ can be computed by intersecting simple cycles on the same row. The problem of computing frequency dominators for a basic block $y$ is reduced to the problem of finding all the simple cycles containing $y$. Among various algorithms for finding simple cycles[26, 27, 23] in a directed graph, we chose Tiernan"s algorithm [26] for its simplicity.

Figure 4.3 presents an algorithm that computes the FDOM relation for basic block $y$ by exhaustively searching for every simple cycle. ***SimpleCycles*** is a specialized form of Tiernan's simple cycle search algorithm [26]. Tiernan's algorithm enumerates all the simple cycles, but our specialized one searches only for simple cycles containing a specific vertex. It directly follows from Tiernan' proof [26] that the search procedure terminates and correctly attains all the simple cycles at line 10 in Figure 4.1 at most once. Figure 4.4 illustrates how the search procedure identifies every simple path or cycle from basic block 1 and 2 in Figure 4.1.

## 4.2 Reducible Control Flow Graphs

This section presents our near-linear-time algorithm for computing the FDOM relation for reducible CFGs. The algorithm uses CFG properties from previous work

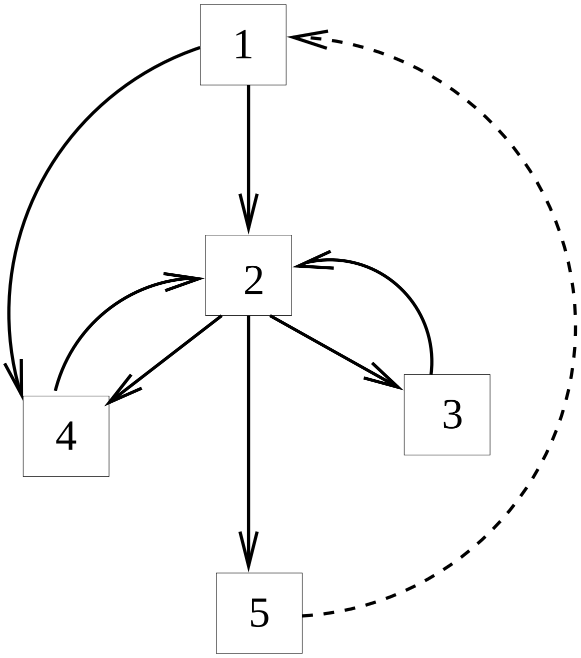


Figure 4.1: Irreducible Control Flow Graph

| y | **simple cycles containing y** | **FDOM(y)** |
|---|---|---|
| 1 | $\langle 1,4,2,5,1\rangle$, $\langle 1,2,5,1\rangle$ | $1,2,5$ |
| 2 | $\langle 2,4,2\rangle$,$\langle 2,5,1,4,2\rangle$,$\langle 2,5,1,2\rangle$,$\langle 2,3,2\rangle$ | $2$ |
| 3 | $\langle 3,2,3\rangle$ | $2,3$ |
| 4 | $\langle 4,2,5,1,4\rangle$, $\langle 4,2,4\rangle$ | $2,4$ |
| 5 | $\langle 5,1,4,2,5\rangle$ $\langle 5,1,2,5\rangle$ | $1,2,5$ |

Figure 4.2: Frequency dominators from simple cycles

```
global
  $fdom_y$: a set of vertices that FDOMs the input vertex $y$

procedure FDOM (y: Vertex): Set
 1: {compute FDOM(y)}
 2: $fdom_y \leftarrow V$ {$V$ is the vertex set in CFG}
 3: $path \leftarrow$ EmptyStack()
 4: Push($path, y$)
 5: {search all the simple cycles of the form: $\langle y, \ldots, y \rangle$}
 6: SimpleCycles($path$)
 7: return $fdom_y$

procedure SimpleCycles (path: Stack)
 1: {Simple Path Extension}
 2: $v \leftarrow$ Peek($path$);
 3: for all $w \in$ Succ($v$) and $w \notin path$ do
 4:     Push($path, w$)
 5:     SimpleCycles($path$)
 6: end for
 7: {Simple Cycle Confirmation}
 8: $y \leftarrow$ Bottom($path$) {the beginning of path}
 9: if $y \in$ Succ($v$) then
10:     $fdom_y \leftarrow fdom_y \cap$ ToSet($path$) {a simple cycle found}
11: end if
12: {Backtracking}
13: Pop($path$);
```

Figure 4.3: Simple FDOM computation algorithm using exhaustive search.

| **SimpleCycles($\langle 1 \rangle$)** | | **SimpleCycles($\langle 2 \rangle$)** | |
|---|---|---|---|
| **path** | **actions on the path** | **path** | **actions on the path** |
| 1 4 2 5 | a simple cycle found | 2 4 | a simple cycle found |
| 1 4 2 3 | | 2 5 1 4 | a simple cycle found |
| 1 4 2 | | 2 5 1 | a simple cycle found |
| 1 4 | | 2 5 | |
| 1 2 4 | | 2 3 | |
| 1 2 5 | a simple cycle found | 2 | |
| 1 2 3 | | | |
| 1 2 | | | |
| 1 | | | |

Figure 4.4: Evaluation of SimpleCycles. Columns with *path* heading show trace of path variable at line 8 of SimpleCycles in Figure 4.3. Left column and right column show path traces respectively for the start basic block 1 and 2.

to compute the FDOM relation. We first present background information on these properties. We then prove that FDOM can be defined in terms of these properties. Finally, we present an algorithm that first computes these properties and then uses the resulting data structures to compute the FDOM relation.

We first describe CFG properties from previous work. For each loop header $h \in V$, the following set is defined by Ball [7]:

$$\begin{aligned}
\mathit{backsrcs}^*(h) &= \{v \mid v \rightarrow h \text{ is a backedge.}\} \\
\mathit{natloop}^*(h) &= \{v \mid \text{there is an } h\text{-free path from} \\
&\qquad v \text{ to a basic block in } \mathit{backsrcs}(h)\} \\
\mathit{exits}^*(h) &= \{v \mid \exists v \rightarrow w \text{ such that } v \in \mathit{natloop}(h) \\
&\qquad \text{and } w \notin \mathit{natloop}(h)\}
\end{aligned}$$

We extend these three definitions to include any basic block $y \in V$.

$$\begin{aligned}
\mathit{loophead}(y) &= \text{loop entry of innermost loop that contains } y \\
\mathit{backsrcs}(y) &= \mathit{backsrcs}^*(\mathit{loophead}(y)) \\
\mathit{natloop}(y) &= \mathit{natloop}^*(\mathit{loophead}(y)) \\
\mathit{exits}(y) &= \mathit{exits}^*(\mathit{loophead}(y))
\end{aligned}$$

Ball defines "$w$ pd $v$ with respect to a set of vertices $V$" to capture post dominance within a natural loop [7]. For our algorithm, we only use two sets of vertices, back edges and loop exits for a given natural loop. Thus we use *PDBE* $y$ to mean *backsrcs*($y$) and *PDLE* $y$ to mean *exits*($y$). We combine these terms and make a new term, *PDL* $y \equiv$ *PDBE* $y$ and *PDLE* $y$.

Theorem 7 defines FDOM in terms of other CFG properties. Ball [7] and

Johnson et al. [17] use sufficient and necessary conditions for FDOM in Lemma 1 to characterize *control regions* [12]. The algorithm in this section is motivated by Ball's paper [7].

**THEOREM 7.** *Given two vertices $x$ and $y$ in* CFG*,*

$$x \text{ FDOM } y$$
$$\textit{if and only if}$$
$$x \in \text{natloop}(y) \textit{ and } (x \text{ DOM } y \textit{ or } x \text{ PDL } y)$$

*Proof.* ($\Rightarrow$) If $x$ is not in *natloop*($y$), then there exists an $x$-free cycle that contains $y$, a contradiction due to Lemma 1. So suppose that $x$ *DOM* $y$ or $x$ *PDL* $y$ is not true. If $x$ *DOM* $y$ is not true, then there exists a path from *loophead*($y$) to $y$ that does not include $x$. If $x$ *PDL* $y$ is not true, then there exists a path from $y$ to either a back edge or an exit of *natloop*($y$). Hence there exists either a path $\langle \textit{loophead}(y), \ldots, y, \ldots, \textit{exits}(y) \rangle$, which means there exists a path from $s$ to $e$ that includes $y$ and not $x$, or there exists a cycle $\langle \textit{loophead}(y), \ldots, y, \ldots, \textit{backsrcs}(y) \rangle$, which is a cycle that includes $y$ and not $x$, also a contradiction. Hence $x$ *DOM* $y$ or $x$ *PDL* $y$ must also be true.

($\Leftarrow$) Show that every cycle that starts and ends at $y$, contains $x$. Let $h_y$ be *loophead*($y$), and every path from $y$ to $y$ is one of this form, $c_y = \langle y, \ldots, z, \ldots, h_y, \ldots, y \rangle$, where $z$ is in $\textit{backsrcs}(y) \cup \textit{exits}(y)$. Suppose that x $\in$ *natloop*($y$), and $x$ *DOM* $y$, then every path from $h_y$ to $y$ must contain $x$. If $x$ *PDL* $y$, then this implies that $\langle y, \ldots, z \rangle \subset c_y$ always contain $x$. Therefore $c_y$ contains $x$ if either of the two conditions is satisfied.

□

*FDOM*($y$) is the set of all $x$ s.t. $x$ *FDOM* $y$, and from Theorem 7, the following equations hold:

$$
\begin{aligned}
FDOM(y) &= natloop(y) \cap (DOM(y) \cup PDL(y)) && (4.2)\\
&= FDOM_D(y) \cup FDOM_P(y) && (4.3)\\
FDOM_D(y) &= natloop(y) \cap DOM(y) && (4.4)\\
FDOM_P(y) &= natloop(y) \cap PDL(y) && (4.5)
\end{aligned}
$$

We create algorithms that compute $FDOM_D(y)$ and $FDOM_P(y)$ in near-linear time. Here, we describes the straight forward but less efficient version from fast FDOM computation algorithms for reducible CFG [21]. Because of the earlier implementation of other variants, we use this algorithm for the online call graph correction. This algorithm computes a set of frequency dominators for a basic block in near-linear time ($O(V + E\alpha(E, V))$ due to the cost of loop structure tree construction [7].

Figure 4.6 shows the fast FDOM computation algorithm. The algorithm takes a basic block $y$ and returns a set of basic blocks that FDOMs $y$. The algorithms has two parts. *Part1* computes the dominator part within the loop by constructing loop structure tree($tree_{lst}$) and dominator tree($tree_{dom}$). Figure 4.7 illustrates control flow graph and these two trees. For instance, consider computing $FDOM_D(3)$. From the loop structure tree, the innermost loop header is a basic block 2, and the loop body set is $\{2, 3, 4, 5, 6\}$. By walking up the dominator tree from basic block 3 and eliminating basic blocks outside the loop, $FDOM_D(3)$ becomes $\{2, 3\}$.

*Part2* transforms a innermost loop body of $y$ into a control flow graph with its entry the loop header and its exit a newly created *TMP* vertex [7]. Figure 4.8 illustrates this transformation process for the inner loop body. Clearly, postdominator in the transformed loop exactly corresponds to the PDL in the original CFG.

| **y** | ***natloop*(y)** | ***FDOM*$_D$(y)** | ***FDOM*$_P$(y)** | ***FDOM*($y$)** |
|---|---|---|---|---|
| 1 | 1,2,3,4,5,6,7 | 1 | 1,2,5,7 | 1,2,5,7 |
| 2 | 2,3,4,5,6 | 2 | 2,5 | 2,5 |
| 3 | 2,3,4,5,6 | 2,3 | 3,5 | 2,3,5 |
| 4 | 2,3,4,5,6 | 2,4 | 2,5 | 2,4,5 |
| 5 | 2,3,4,5,6 | 2,5 | 5 | 2,5 |
| 6 | 2,3,4,5,6 | 2,5,6 | 6 | 2,5,6 |
| 7 | 1,2,3,4,5,6,7 | 1,2,5,7 | 7 | 1,2,5,7 |

Figure 4.5: Computing FDOM using natural loop and dominator trees

For instance, by walking up from 3 in the postdominator tree up to the TMP vertex, $FDOM_P(3)$ becomes $\{3, 5\}$. Figure 4.5 shows the frequency dominators for all the vertices using this algorithm.

**procedure FDOM (*cfg*:CFG, $y$:Vertex): Set;**

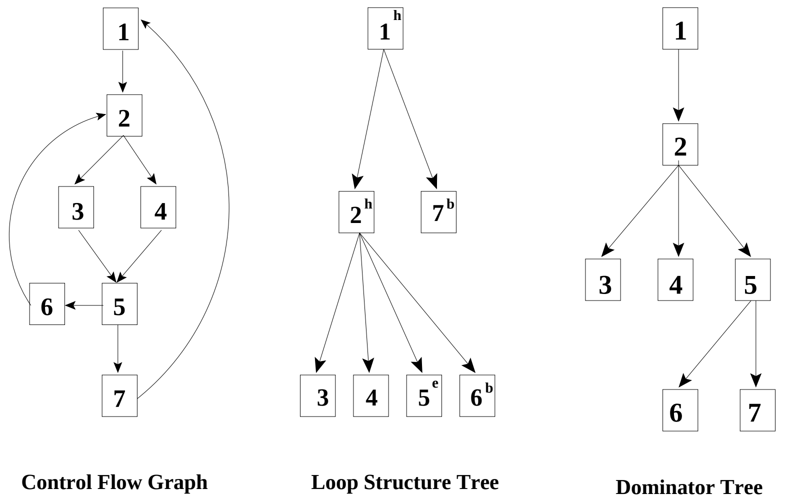
```
{Part1: compute $FDOM_D(y)$ by walking up the dominator tree}
$tree_{dom} \leftarrow$ BuildDominatorTree($cfg$)
$tree_{lst} \leftarrow$ BuildLoopStructureTree($cfg, tree_{dom}$)
$natloop_y \leftarrow$ Natloop($y, tree_{lst}$)
$fdom_y \leftarrow \{y\}$
$x \leftarrow$ Parent($y, tree_{dom}$)
while $x \neq nil$ do
   if $x \in natloop_y$ then
      $fdom_y \leftarrow fdom_y \cup \{x\}$
   end if
   $x \leftarrow$ Parent($x, tree_{dom}$)
end while

{Part2: compute $FDOM_P(y)$ by walking up the PDL tree}
$cfg_{loop} \leftarrow$ BuildLoopCFG($cfg, tree_{lst}, y$)
$tree_{pdl} \leftarrow$ BuildPostDominatorTree($cfg_{loop}$)
$x \leftarrow$ Parent($y, tree_{pdl}$)
while not $IsTMP$(x) do
   $fdom_y \leftarrow fdom_y \cup \{x\}$
   $x \leftarrow$ Parent($x, tree_{pdl}$)
end while
return $fdom_y$
```

**procedure BuildLoopCFG(*cfg*:CFG, *tree*$_{loop}$:LSTree, $y$:Vertex): CFG;**
takes a subgraph of CFG that consists of only vertices within the inner-most natural loop containing the input vertex, and redirects the back edges and exit edges to a newly created TMP vertex.

```
$h_y \leftarrow$ Loophead($tree_{lst}, y$);
$V_{loop} \leftarrow$ Natloop($h_y, tree_{lst}$) $\cup \{TMP\}$
$E_{loop} \leftarrow \varnothing$
for all $v \in V_{loop}$ do
   for all $w \in$ Succ($v, cfg$) do
      if $w = h_y$ or $w \notin V_{loop}$ then
         $E_{loop} \leftarrow E_{loop} \cup \{(v, TMP)\}$ {redirect to TMP if back edge or exit}
      else
         $E_{loop} \leftarrow E_{loop} \cup \{(v, w)\}$ {otherwise, copy the old control flow edge}
      end if
   end for
end for
$cfg_{loop} \leftarrow (V_{loop}, E_{loop}, h_y, TMP)$ {a new CFG with $h_y$ entry and $TMP$ exit}
return $cfg_{loop}$
```

Figure 4.6: A Fast FDOM computation algorithm

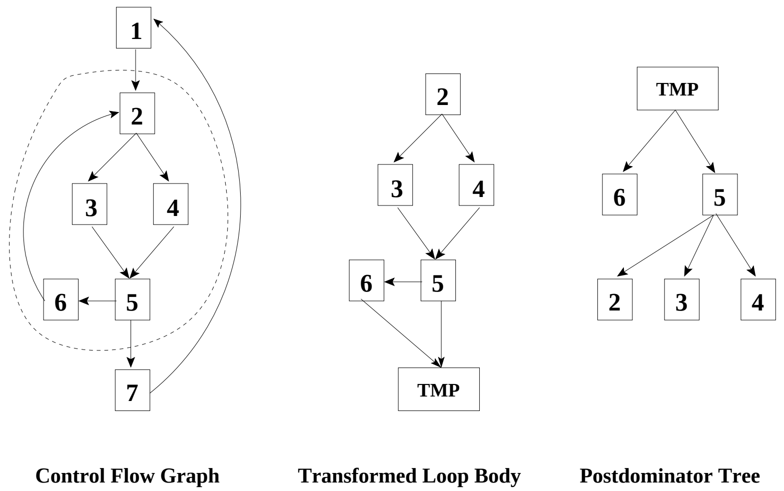


Control Flow Graph

Loop Structure Tree

Dominator Tree

Figure 4.7: Control flow graph, loop structure tree and dominator tree. In the loop structure tree, loop header, back source and exit source are marked by $h$, $b$ and $e$ respectively.



Control Flow Graph

Transformed Loop Body

Postdominator Tree

Figure 4.8: Loop body transformation to compute $FDOM_P(y)$ for basic block $y$ inside the loop body

# Chapter 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 implemented and evaluated DCG correction algorithms in Jikes RVM (CVS HEAD 2005/08/13 21:10:06 UTC) a Java-in-Java VM in its production configuration [2]. This configuration pre-compiles the VM methods (e.g., compiler and garbage collector) and any libraries it calls into the boot image. Jikes RVM contains two compilers: the *baseline compiler* and *optimizing compiler* with three optimization levels. (There is no interpreter in this system.) When a method is first executed, 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.

We use the SPEC JVM98 benchmarks and the DaCapo benchmarks [8]. We omit the DaCapo benchmarks *xalan*, and *hsqldb* because we could not get *xalan* to run correctly with Jikes RVM, and *hsqldb* showed very large run-to-run variation in its execution due to Jikes RVM's thread scheduling implementation. We omit the DaCapo benchmark *antlr* from the inlining performance experiments because we could not get it to run correctly for 25 iterations.

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 12K$\mu$ops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 1GB 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 optimizing compiler to eliminate non-determinism due to sampling and since call graph accuracy is not influenced by code quality. By the end of the application's execution, the sampler will have collected the perfect call graph. 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 calls 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 final perfect DCG to the final corrected DCG generated by our approach. Because DCG clients use incomplete graphs to make optimization decisions, we could have compared the accuracy of the instantaneous perfect and corrected DCGs as a function of time. We follow prior work in comparing the final graphs [3] rather than a time series, and believe these results are representative of the instantaneous DCGs.

We measure accuracy against two (final) perfect DCGs: (1) *without inlining* and (2) *with trivial inlining.* The first configuration has the largest DCG, but the second is more representative of the base call graph presented to the optimizing compiler. Trivial inlining in Jikes RVM inlines a call site if the size of the callee is smaller than the calling sequence. The inliner will therefore never need to use the frequency information for these call sites.

**Overhead Methodology.** To measure the overhead of DCG correction without including its influence on optimization decisions, we configure the correction algorithms to report uncorrected information to the method inliner. The correction does not affect inlining decisions, but the execution time includes the correction overhead.

**Performance Methodology.** We use the following configuration to measure the performance of using corrected DCGs to drive inlining. We correct the DCG on the fly by performance correction as the VM optimizes the application, providing a realistic measure of DCG correction's ability to affect inlining decisions. We measure steady-state performance by executing 25 iterations of the application in one run of the VM, and we take the median time of iterations 13 through 25, which we find decreases variation due to the non-determinism of the adaptive compiler.

# Chapter 6

# Results

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

## 6.1 Accuracy

We use the overlap accuracy metric from prior work to compare the accuracy of DCGs [3]. Chapter 3.7 discusses the overlap accuracy and upper bound for our correction algorithms.

We use the notation *CBS(SAMPLES, STRIDE)* to refer to a counter based sampling configuration which we discuss in Chapter 1. For example, *CBS(1,1)* is equivalent to timer-based sampling as shown in Figure 1.1(a): it takes one sample per timer tick without striding. *CBS(3,3)* takes three samples per timer tick and strides up to three samples as show in Figure 1.1(a). To compare the effect of sampling configuration on call graph correction, we use two sampling configurations: *CBS(1,1)* and *CBS(16,3)*. The default sampling configuration is *CBS(16,3)* if not explicitly stated. *CBS(16,3)* is the recommended sampling and striding settings[3] that take more samples to increase DCG accuracy, but keep average overhead under

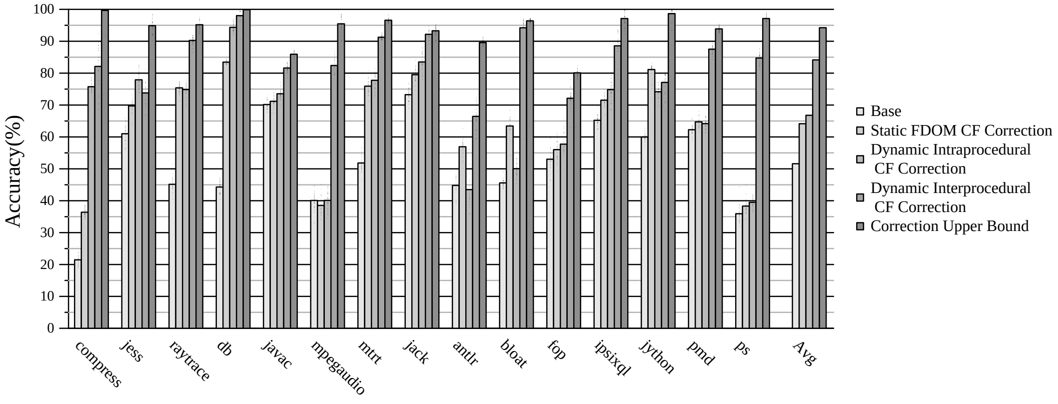


Figure 6.1: Accuracy of DCG correction on the complete DCG.

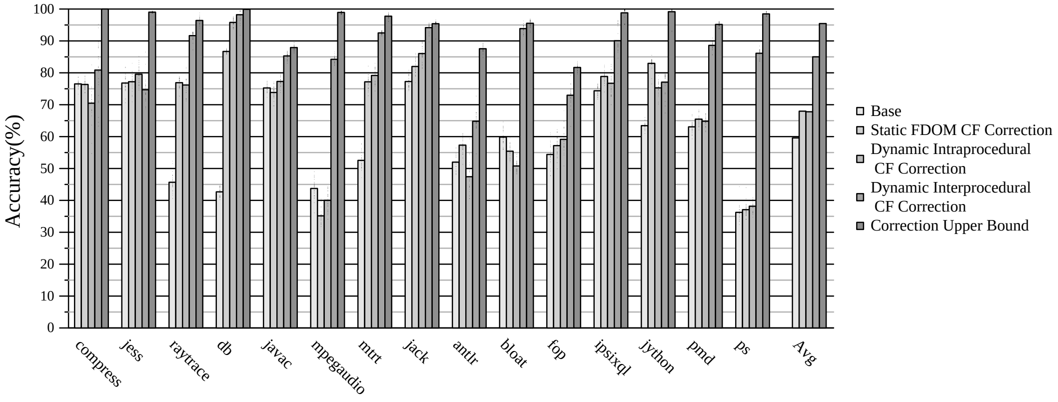


Figure 6.2: Accuracy of DCG correction on the DCG with trivially inlined call sites.

a few percent. This configuration has an average accuracy of 52% (no inlining) and 60% (static inlining), and drops as low as 20%.

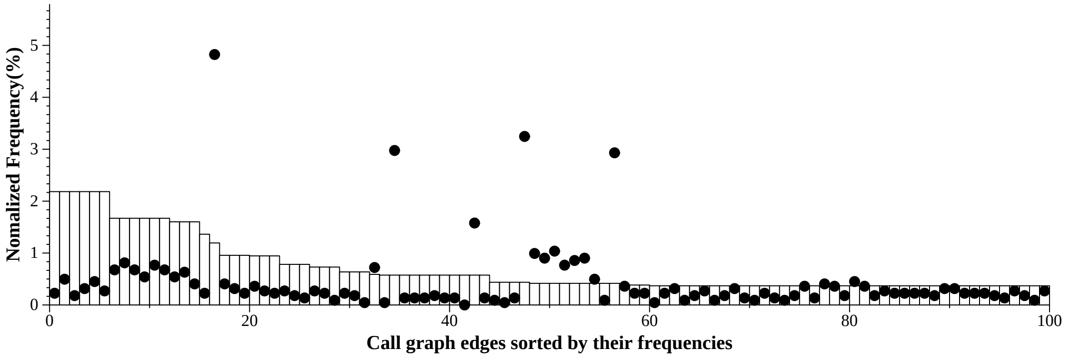
Figures 6.1 and 6.2 compare DCG accuracy for the no-inlining and trivially-inlining configurations. For the individual benchmarks, we report average profile accuracy over 25 trials (shown as dots). The perfect DCG is 100% (not shown). Since the DCGs are similar between the two figures, the accuracies are as expected also similar. The graphs compare the perfect DCG to the base system without correction (*Base*), *Static FDOM CF Correction*, *Dynamic Intraprocedural CF Cor-*

*rection*, *Dynamic Interprocedural CF Correction*, and the *Correction Upper Bound* ($DCG_{bound}$). These configurations are discussed in Chapter 3.6 and Chapter 3.7.

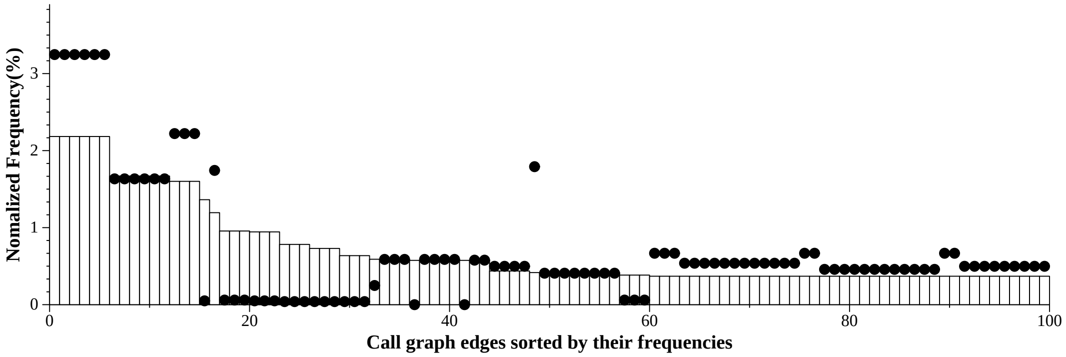
Static and dynamic DCG correction together significantly improve accuracy to an average of 85%. The second bar, *Static FDOM CF Correction*, teases apart the contribution just from static FDOM, which improves over sampling by 7 to 10% on average. Intraprocedural correction performs about the same, but interprocedural correction applies global dynamic constraints and shows much better accuracy than the local methods. In addition, interprocedural correction comes within 10% of the bound, which loses only on average 5% to missing call edges. Interprocedural correction is 10% less than the upper bound because (1) DCG correction does not correct the relative frequencies of call edges coming out of the same call site, and (2) the basic block profiles are slightly inaccurate because they measure only initial execution behavior [9].

Figure 6.3 shows how the correction algorithms change the shape of the DCG for *raytrace*. The vertical bar presents normalized frequencies of 100 most frequently executed call edges from the perfect DCG. The circles shows the frequency from the sampled DCG or corrected DCGs. In base case, call edges have different frequencies due to sampling bias although they have the same frequency in the perfect DCG. Static FDOM CF Correction eliminates some of these abnormality and improves the shape of DCG. Dynamic Intraprocedural CF Correction further improves the DCG because it uses fractional frequency between two call sites, while FDOM gives only relative frequency. However, Intraprocedural CF Correction can not achieve the correction upper bound due to interprocedural error in DCG. Finally, Interprocedural CF Correction reaches very close to the correction upper bound by eliminating interprocedural sampling bias.

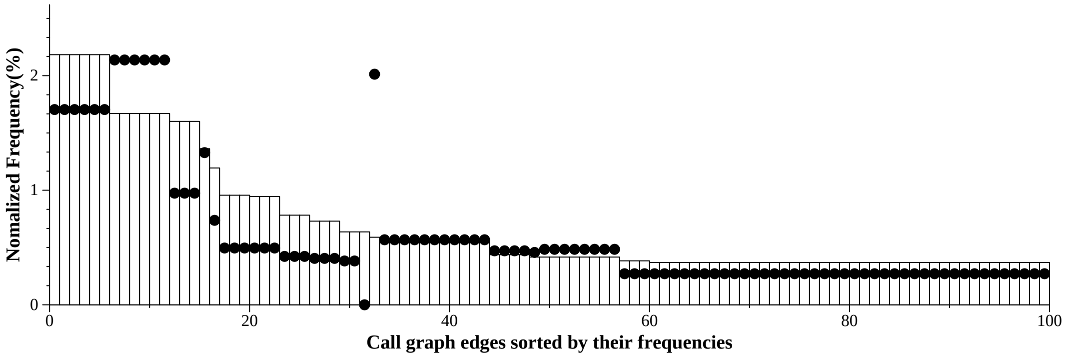
Figures 6.4 compares average DCG accuracy by varying sampling configuration and inlining configuration. We take the average over all the benchmarks



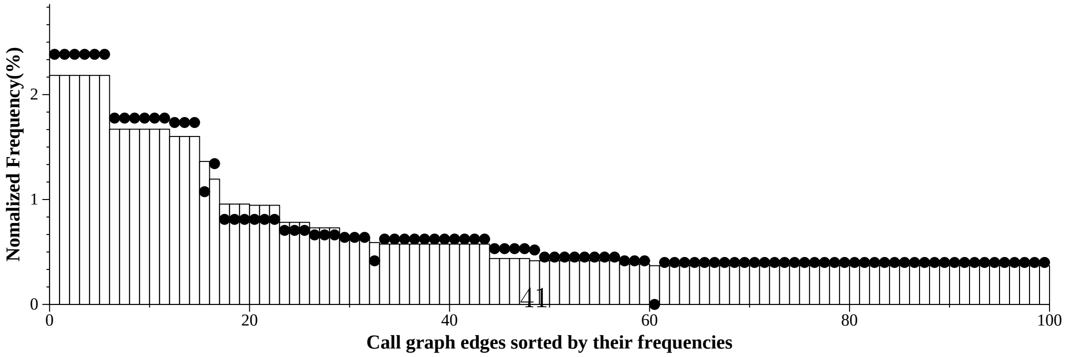
(a) Base



(b) Static FDOM CF Correction



(c) Dynamic Intraprocedural CF Correction



(d) Dynamic Interprocedural CF Correction

Figure 6.3: Call graph frequencies for raytrace

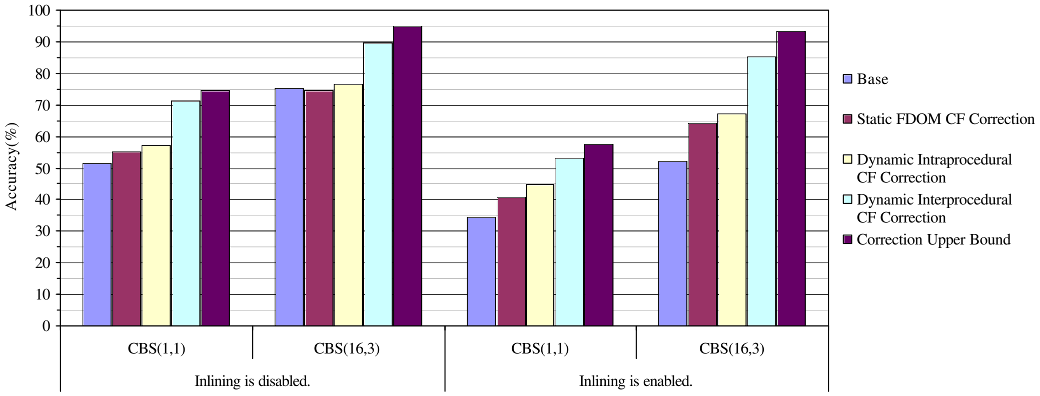


Figure 6.4: Average accuracy of DCG correction by varying sampling and inlining configuration.

with 10 trials. The graph shows how the counter based sampling and inlining affects accuracy and how our call graph correction algorithm interacts with these two configurations. First considering the *Base* case. Increasing CBS configuration parameters from *CBC(1,1)* to *CBS(16,3)* improves the accuracy by 24% or 18%, depending on whether inlining is turned on or not. Also, inlining hurts accuracy by 17% or 23%. The Jikes RVM stops updating call site frequency after the call site is inlined, and it compensates the lack of this information by inlining always a call site once it's inlined before [2].

The call graph corrections orthogonally improves accuracy regardless of sampling and inlining configuration. Static FDOM CF Correction and Dynamic Intraprocedural CF Correction improve accuracy by 2% to 15%. The accuracy improvement is 2% to 6% when inlining is turned off, and 10% to 15% when inlining is turned on. When some call site in a method is inlined, and others are not, frequencies for the inlined call site are not updated. Static or dynamic intraprocedural correction takes care of this type of inaccuracy by comparing call site frequencies between inlined and non-inlined call sites. Dynamic Interprocedural CF Correc-

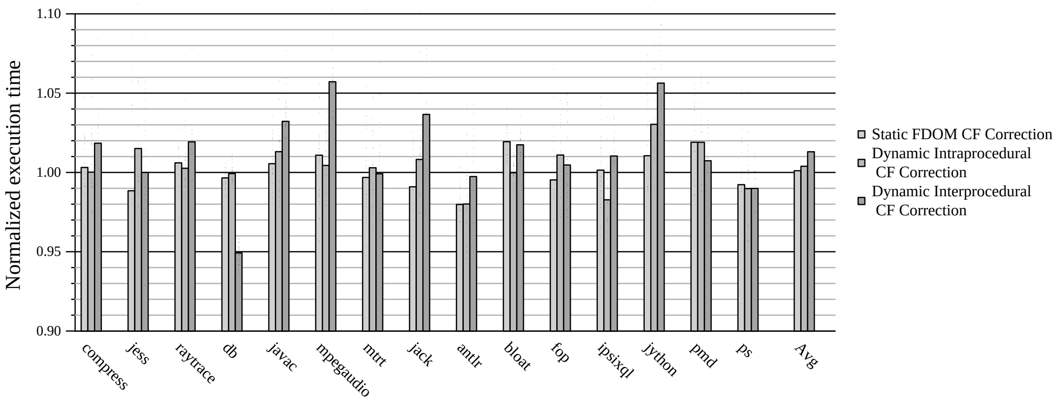


Figure 6.5: The runtime overhead of call graph correction.

tion significantly improves accuracy by 8% to 18 % over the two intraprocedural corrections across all the cases.

## 6.2 Overhead

Figure 6.5 presents the execution times for various DCG correction configurations using the adaptive compiler in which the corrected DCGs are never used, but are computed each time the optimizing compiler recompiles a method. Correction could occur on every sample, but this approach aggregates the work and eliminates repeatedly correcting the same edges. We take the median out of 25 trials (shown as dots) to eliminate high variability of the adaptive runs. 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 6% on mpegaudio and jython.

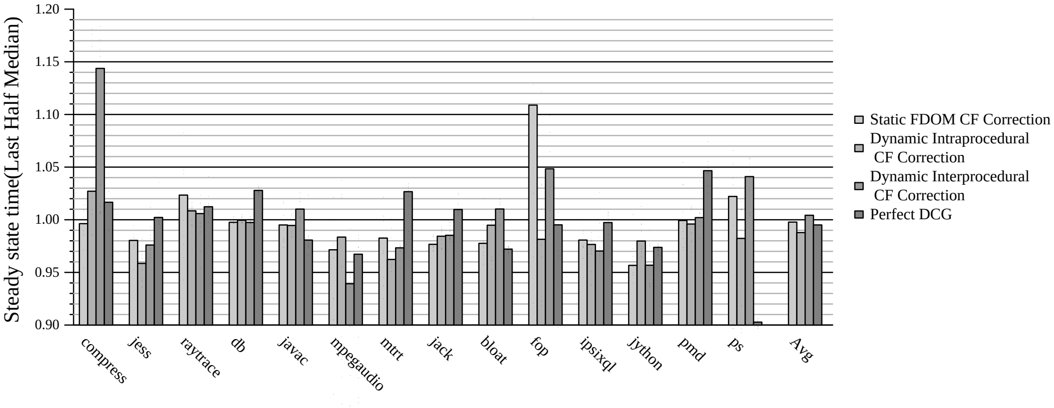


Figure 6.6: The steady state performance of correcting inlining decision using adaptive methodology.

## 6.3 Performance

We evaluate the costs and benefits of using DCG correction to drive optimization in Jikes RVM. Figure 6.6 shows steady-state performance (median of iterations 13 through 25) with several DCG correction configurations. We repeat each experiment 10 times (shown as dots) and take the median. The graphs are normalized to the exection time without correction. *Static FDOM CF Correction* shows the improvement from static FDOM correction, which is about 0.2% on average. *Dynamic Intraprocedural CF Correction* improves performance by almost 1% on average. *Dynamic Interprocedural CF Correction* uses method invocation counters (Section 3.6) to get basic block profiles with interprocedural accuracy. While the method counters provide higher accuracy, they hurt performance slightly (by 0.4% on average) in steady state, particularly for *compress*, *fop* and *ps*.

We also evaluate the performance of *Perfect DCG*, which feeds a perfect DCG to the inliner at the beginning of execution, rather than computing and correcting it on the fly as in the other configurations. The perfect DCG improves performance by only about 0.5% on average, suggesting that Jikes RVM's inliner cannot benefit significantly from high-accuracy DCGs. Previous work confirms that higher accuracy does not help Jikes RVM's inliner much but that other VMs can benefit by up to 9% from higher accuracy [3].

# Chapter 7

# Conclusion

This paper introduces *dynamic call graph (DCG) correction*, a novel approach for increasing DCG accuracy with existing static and dynamic control-flow information. We introduce the *frequency dominator* (FDOM) relation to constrain and correct DCG frequencies, and also use intraprocedural and interprocedural basic block profiles to correct the DCG. By adding just 1% overhead on average, DCG correction increases average DCG accuracy from 52-60% to 85%. Although we obtain only a modest performance boost from using corrected DCGs to drive inlining, prior work shows other VMs benefit from higher accuracy DCGs, and we believe DCG correction will be increasingly useful in the future as object-oriented programs become more complex and more modular.

# Bibliography

[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. 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.

[3] 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.

[4] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, UT, 2001.

[5] 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.

[6] V. Bala, E. Duesterwald, and . Banerjia. Dynamo: A Transparent Dynamic Op-

timization System. In *ACM Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–12, Boston, MA, 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, K. S. McKinley, J. E. B. Moss, S. Augart, P. Cheng, A. Diwan, S. Guyer, M. Hirzel, C. Hoffman, A. Hosking, X. H. M. Jump, A. Khan, P. McGachey, D. Stefanović, and B. Wiedermann. The dacapo benchmarks. Technical report, 2006. http://ali-www.cs.umass.edu/DaCapo/Benchmarks.

[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] J. Cavazos and M. F. P. O'Boyle. Automatic tuning of inlining heuristics. In *ACM/IEEE Conference on Supercomputing*, pages 14–14, Washington, DC, 2005.

[11] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[12] J. F., K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[13] 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.

[14] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 108–123, New York, NY, USA, 1995.

[15] 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.

[16] Intel Corporation. Intel itanium 2 processor. http://www.intel.com/-/products//processor//itanium2/index.htm.

[17] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *ACM Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[18] 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.

[19] 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.

[20] R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Eastern Joint Computer Conference*, pages 133–138, NY, 1959. Spartan Books.

[21] K. Resnick. Call graph correction using control flow constraints. B.S. thesis, The University of Texas at Austin Computer Science Department, Austin, Texas, 2006.

[22] 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.

[23] R. E. Tarjan. Enumeration of the elementary circuits of a directed graph. Technical report, Ithaca, NY, USA, 1972.

[24] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal of Computing*, 3(1):62–89, 1974.

[25] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[26] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM*, 13(12):722–726, 1970.

[27] H. Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *J. ACM*, 19(1):43–56, 1972.

[28] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM Conference on Java Grande*, pages 78–87, 2000.

[29] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *ACM/IEEE International Symposium on Microarchitecture*, pages 1–11, 1994.

# Vita

Byeongcheol Lee was born in Jeonju, S. Korea on August 30, 1976, the son of Dr. Sanghyun Lee and Gymok Chang. He received the double B.E. in Electronic & Electrical Engineering and Computer Science & Engineering, Magna Cum Laude from POSTECH, S. Korea in August, 2004. In September, 2004, he entered The Department of Computer Sciences at The University of Texas at Austin.

Permanent Address: 103-304, Seogok Hyundai APT, Hyoja3-ga, Wansan-gu, Cheunju, cheonbuk, South Korea

This thesis was typeset with LaTeX $2_\varepsilon$[1] by the author.

---

[1] LaTeX $2_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.