Profiling k-Iteration Paths : A Generalization of the Ball-Larus Profiling Algorithm

Subhajit Roy
Department of Computer Science and Automation
Indian Institute of Science, Bangalore, India
subhajit@csa.iisc.ernet.in

Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science, Bangalore, India
srikant@csa.iisc.ernet.in

Abstract

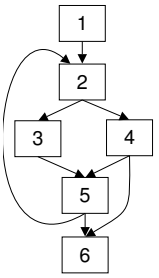
The Ball-Larus path-profiling algorithm is an efficient technique to collect acyclic path frequencies of a program. However, longer paths — those extending across loop iterations — describe the runtime behaviour of programs better. We generalize the Ball-Larus profiling algorithm for profiling k-iteration paths — paths that can span up to to k iterations of a loop. We show that it is possible to number such k-iteration paths perfectly, thus allowing for an efficient profiling algorithm for such longer paths. We also describe a scheme for mixed-mode profiling: profiling different parts of a procedure with different path lengths. Experimental results show that k-iteration profiling is realistic.

1. Introduction

An efficient algorithm for acyclic path profiling — profiling paths that either terminate at backedges or at procedure exits — was proposed by Ball and Larus [2]. The key idea of the algorithm was to assign unique path-identifiers to all acyclic paths in a manner such that the paths-identifiers of the traversed paths can be efficiently reconstructed during the profile run of the program. More importantly, the algorithm computes a *perfect* numbering of these path-identifiers: if there are `n` static acyclic paths in the program, the paths are assigned identifiers from `0` to `n-1`.

Though acyclic path profiles are very useful at driving many compiler optimizations, more opportunities can be exploited in the presence of information about longer paths — paths extending *across* loop iterations. Figure 1 shows a control-flow graph being profiled, and the frequency counts for acyclic and two-iteration paths (paths including two iterations of the loop) for a program trace reading `1-(2-3-5-2-4-5)`^{100}`-6` — within the loop, the acyclic paths `2-3-5` and `2-4-5` execute alternately. Hence, it may be beneficial to unroll the loop and perform trace scheduling along the path `2-3-5-2-4-5`. Note that this information is lacking in the acyclic path profile; it only shows that both the acyclic paths `2-3-5` and `2-4-5` are equally likely to occur

Subhajit Roy is supported by doctoral fellowship provided by Philips Research, India.



Acyclic	freq
1-2-3-5	1
2-3-5	99
2-4-5	99
2-4-5-6	1

2-Itr Path	freq
1-2-3-5-2-4-5	1
2-3-5-2-4-5	98
2-4-5-2-3-5	99
2-3-5-2-4-5-6	1

Figure 1: Motivation for k-iteration path profiling

in the trace. The only conclusion that can be drawn out of it for the longer set of paths is that the paths `2-3-5-2-4-5` and `2-4-5-2-3-5` as well as the paths `(2-3-5)`^{2} and `(2-4-5)`^{2} are all likely to be hot. Acyclic paths — due to the fact that they terminate at loop backedges — do not give any information about the correlation among the loop paths. Multiple-iteration paths can uncover such information.

Tallam et al. [3] have proposed the use of longer paths — paths that cover two iteration of a loop — to enable multiple optimizations. They estimate two-iteration paths within loops from *Overlapping Ball-Larus paths* — paths that extend a fixed number of nodes beyond the backedge.

In this paper, we propose a profiling algorithm to collect frequencies for such longer paths. In contrast to the algorithm given by Tallam et al. [3], which gives an *approximation* of the frequency counts for the longer paths, our algorithm provides the *exact* frequency counts. Our algorithm is a generalization of the Ball-Larus profiler: it can provide information about paths that span multiple iterations of a loop; for one-iteration paths, it reduces to the Ball-Larus profiler.

However, when profiling longer paths, the number of static paths increases tremendously. Interestingly, our algorithm allows *mixed-mode profiling*: use of different path-lengths (in terms of the number of iterations considered for the various loops) for different loops in the same procedure. Hence, the program analyst may use higher iteration paths for "interesting" regions of the procedure while counting acyclic paths for the remaining program; thus controlling the number of static paths.

Following are our contributions in this paper:

- we generalize the Ball-Larus algorithm for more gen-

Acyclic Paths

0: 1-2-3-5-6 **1**: 1-2-3-5 **2**: 1-2-4-5-6 **3**: 1-2-4-5 **4**: 1-2-4-6 **5**: 2-3-5-6 **6**: 2-3-5 **7**: 2-4-5-6 **8**: 2-4-5 **9**: 2-4-6

Two-Iteration Paths

0: 1-2-3-5-2-3-5-6 **1**: 1-2-3-5-2-3-5 **2**: 1-2-3-5-2-4-5-6 **3**: 1-2-3-5-2-4-5 **4**: 1-2-3-5-2-4-6 **5**: 1-2-3-5-6
6: 1-2-4-5-2-3-5-6 **7**: 1-2-4-5-2-3-5 **8**: 1-2-4-5-2-4-5-6 **9**: 1-2-4-5-2-4-5 **10**: 1-2-4-5-2-4-6 **11**: 1-2-4-5-6
12: 1-2-4-6 **13**: 2-3-5-2-3-5-6 **14**: 2-3-5-2-3-5 **15**: 2-3-5-2-4-5-6 **16**: 2-3-5-2-4-5 **17**: 2-3-5-2-4-6
18: 2-4-5-2-3-5-6 **19**: 2-4-5-2-3-5 **20**: 2-4-5-2-4-5-6 **21**: 2-4-5-2-4-5 **22**: 2-4-5-2-4-6

Figure 2: Acyclic and Two-Iteration Static Paths.

eral k-iteration paths — we show that it is possible to achieve *perfect numbering* even for these longer paths (section 3);
- we provide an instrumentation algorithm for collecting k-iteration path profiles (section 4);
- we propose mixed-mode profiling: profiling different regions of the same procedure with differing path lengths (section 5);
- we present experimental data indicating that k-iteration profiling is realistic (section 6).

2. Background

Ball and Larus [2] proposed a simple and fast algorithm for counting the frequency of acyclic paths — paths that terminate at a backedge or reach the end of the procedure. The Ball-Larus algorithm assigns weights to the edges in the control-flow graph in such a manner that the sum of the edge-weights on each acyclic path is unique — a *path-identifier*. The Ball-Larus algorithm reduces graphs with loops into DAGs while retaining all acyclic static paths using a set of dummy edges — edges from a dummy entry node[1] to all targets of backedges and from each source of a backedge to a dummy exit node. The algorithm then traverses the nodes in the DAG in a reverse topological order, assigning weights to each outgoing edge of a node as it is visited. The instrumentation code, added on the edges of the control-flow graph, uses a dedicated register to accumulate the edge-weights as the edges are traversed during the execution of the instrumented program.

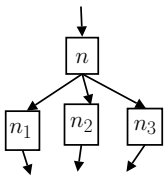


Figure 3: Numbering Paths

The key idea of the Ball-Larus algorithm is as follows: in Figure 3, if all the paths from the nodes n_1, n_2 and n_3 were uniquely numbered from `0` to N_1, `0` to N_2 and `0` to N_3 respectively, then assigning a weight of 0 to $n \rightarrow n_1$, N_1 to $n \rightarrow n_2$ and $N_1 + N_2$ to $n \rightarrow n_3$ will uniquely number each path from n.

Our algorithm extends this idea to longer paths — paths that span a finite number of loop iterations (say `k`); we call such paths as *k-iteration paths*.

Definition 1: A k-iteration path is a path in the procedure's flow graph that either ends at

- a backedge after the loop-body has been executed `k` times, or
- the procedure exit.

1. Dummy entry (connecting to the root node) and exit nodes (connected from all the procedure return nodes) are added to each graph.

Though Ball-Larus paths are essentially one-iteration paths, we will refer to the Ball-Larus paths as BL paths or acyclic paths — the term *k-iteration paths* will strictly imply that `k>1`. Figure 2 shows all the acyclic and k-iteration paths for the graph in Figure 1.

What is not a valid k-iteration path ? A path must meet the following requirements to be a valid k-iteration path:

- Like acyclic paths, a k-iteration path may start only at the procedure entry or a loop-head (destination of a backedge) and end either at a loop-tail (source of a backedge) or the procedure exit. Note that this requirement allows a path that iterates a loop `L` less than k times if it does not both start at the loop-head of `L` and end at the loop-tail of `L`.
- Any path that starts with the loop-head of loop `L` must iterate through `L` exactly `k` times.
- Any path that terminates at the loop-tail of loop `L` must also iterate through `L` exactly `k` times.

Example 1: Consider Figure 1: the paths `2-3-5-6` and `1-2-3-5` are invalid two-iteration path as they start with the loop-head (node `2`), and terminate at the loop-tail (node `5`) respectively, but iterate through the loop just once. However, the path `1-2-3-5-6`, that also iterates through the loop just once is a valid two-iteration path as it does not either start with a loop-head or terminate at a loop-tail.

If such invalid paths are not eliminated during path-numbering, the "perfect numbering" property would be violated. Our k-iteration path-numbering algorithm perfectly numbers the k-iteration paths, identifying and eliminating such invalid paths while assigning the path-identifiers (this problem does not occur for acyclic paths).

The program instrumentation algorithm uses an array of `k` counters to construct the k-iteration path-identifiers during the profile run of the program.

Understandably, the number of static paths increases exponentially in `k` — for `n` static acyclic paths, there will be $O(n^k)$ k-iteration static paths in the worst case. The number of static paths in a procedure is important: for a small number of static paths, the profiler can choose to use an array-based implementation of the path-frequency table; otherwise, it would need an expensive hashtable-based implementation. To counter the same, our algorithm allows *mixed-mode profiling:* small, more interesting parts of the procedure are profiled using k-iteration profiling, while acyclic path profiling is used for the rest of the procedure.

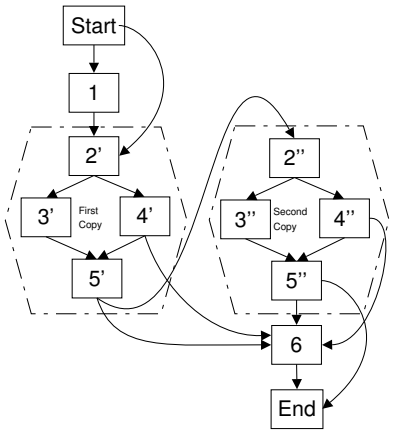


Figure 4: Two-iteration Loop-Unrolled DAG.

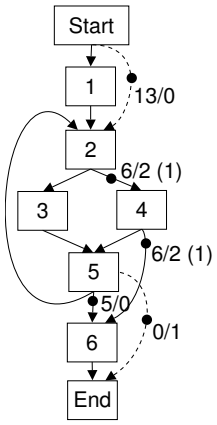


Figure 5: Edge-weights for tracing two-iteration paths.

3. The Path-Numbering Algorithm

We define a few terms to ease the following discussion: *loop-head* and *loop-tail* are the target and source nodes of a loop backedge (respectively). A *loop-entry* is a node in a loop that is the target of an edge from a node outside the loop; the edge is called the *loop-entry* edge. Similarly, a *loop-exit* is a node in a loop that is the source of an edge to a node outside the loop; the edge is called the *loop-exit* edge. We call the source of the loop-entry edge as the *pre-entry* node and the target of the exit edge as the *post-exit* node. Any node that is part of a loop L is said to be a *loop-node* of L. An edge $u \rightarrow v$ is called a loop-edge of `L` if both `u` and `v` are loop-nodes of `L`. For a reducible graph, the loop-head and the loop-entry nodes are the same. We add a dummy start node (denoted *Start*) and a dummy end node (denoted *End*) to the graph.

Example 2: In Figure 1, node `2` is the loop-entry node as well as the loop-head. Node `4` and `5` are the loop-exits. Node `5` is also the loop-tail. Edge `1`→`2` is the loop-entry edge, `5`→`2` is the backedge while `4`→`6` and `5`→`6` are the loop-exit edges. Nodes `1` and `6` are the pre-entry and post-exit nodes respectively.

In the following sub-sections, we discuss the path-numbering and path-identification algorithms. We assume a reducible graph with no nested loops. Similar to the Ball-Larus algorithm, we ignore self-loops — in our algorithms, we assume the absence of self-loops.

3.1. The loop-unrolled DAG

Before discussing the algorithms, let us look at a special graph — the loop-unrolled DAG — which will enable us to understand the following algorithms better. For k-iteration paths, the k-iteration loop-unrolled DAG, $G_{lu(k)}$, can be formed from a graph G via the following steps (for each loop in the graph):

- unroll the loop, creating `k` copies of it: as we are interested in tracing paths that contain `k` iterations of the loop, unrolling the loop reveals all such paths; a node n^i in the DAG is called the i^{th} version of node n in the CFG if n^i corresponds to the node n in the i^{th} copy of the loop;
- add a dummy edge from the `Start` node to the first version of the loop-head: paths originating from this edge denote k-iteration paths in G that begin with the loop-head;
- add a dummy edge from the k^{th} version of the loop-tail to the `End` node: paths terminating with this edge denote k-iteration paths in G that end at the loop-tail;
- remove the backedges.

Similar to acyclic path-profiling, the dummy edges create new paths in $G_{lu(k)}$ that either start at the loop-head or terminate at the loop-tail. As the loop needs to be traversed exactly `k`-times for such k-iteration paths, only the first version of the loop-head in $G_{lu(k)}$ is a target of a dummy edge from the `Start` node and only the k^{th} version of the tail-node is the source of a dummy edge to the `End` node — this disallows all paths that are traversed fewer than `k` times. As the loop is unrolled to contain exactly `k` copies, and all the backedges are removed, all paths in G that traverse the loop more than `k` times are absent in $G_{lu(k)}$.

Example 3: Figure 4 shows the two-iteration loop-unrolled DAG ($G_{lu(2)}$) for the graph in Figure 1.

3.2. The Path-Numbering Algorithm

The goal of the path-numbering algorithm (Algorithm 1) is to assign unique identifiers to all the possible k-iteration paths in a given graph G. Instead of unrolling the loops to create $G_{lu(k)}$, we rather construct a new graph G_δ from G, simply by adding dummy edges from the `Start` node to each loop-head (we denote all such dummy edges by δ^{start}) and dummy edges from the loop-tail to the `End` node (we denote all such dummy edges by δ^{end}) to allow static paths beginning at a loop-head and ending at a loop-tail (for a loop L, we denote the dummy edges for it by δ_L^{start} and δ_L^{end}; note that it is possible that $\delta_{L_1}^{start} = \delta_{L_2}^{start}$ for two different loops L_1 and L_2). The k^{th} iteration through a loop L in G_δ corresponds to the traversal through the k^{th} copy of L in $G_{lu(k)}$.

The path-numbering algorithm simulates the traversal through $G_{lu(k)}$ by iterating through loops in G_δ `k`-times while ignoring appropriate edges in each iteration; the `Ignored` function indicates if an edge is ignored in a particular iteration.

3.2.1. Numbering all the k-iteration static paths. The algorithm computes a set of weights for all edges in a graph G such that the sum of these weights along any path gives an unique identifier for the respective k-iteration path. For this

purpose, the algorithm needs to compute `numPaths(n)`, the number of paths through each node `n`. For example, in Figure 4, there exists six paths through `3'` and seven paths through `4'`. Hence, `numPaths(2')=13`.

Algorithm 1 Path Numbering of k-iteration static paths; *Ignored* gives the set of all edges that need to be ignored in a particular iteration of the loop.

Input

- *g: graph with dummy edges (G_δ)*
- *k: the k-iteration profiling parameter*

Output

- *val, cval: set of edge-weights and compensation edge-weights*

begin
$list$: reverse topological sequence of nodes in g

$\forall u \in g$, set $itrNum(u) := \begin{cases} k & \text{if } u \text{ is a loop node,} \\ 1 & \text{otherwise} \end{cases}$

while $u \neq Start$ **do**
  u := Get the next item from $list$
  $i := itrNum(u)$
  if $u \neq End$ **then**
    $numPaths(u) := 0$
  else
    $numPaths(u) := 1$
  end if
  $invalidPaths(u) := 0$
  for all edges $(e : u \rightarrow v) \in g$ **do**
    if $\neg$(Ignored(e, i)) **then**
      $val(e, i) := numPaths(u)$
      $numPaths(u)$ += $numPaths(v)$
      if $i < k \ \wedge u \in \{$loop-nodes of $L\} \cup \{Start\}$ **then**
        $cval(e, i) := invalidPaths(u)$
        if edge $e \in$ loop-exit-edges of L **then**
          $invalidPaths(u)$ += $numPaths(v)$
        else
          $invalidPaths(u)$ += $invalidPaths(v)$
        end if
      else
        $cval(e, i) := 0$
      end if
    end if
  end for
  if $u =$ head-node of some loop $L \wedge i > 1$ **then**
    Set $itrNum(x) := (i - 1)$ for all nodes $x \in L$
    Set the list pointer to the tail node of the loop L
  end if
end while
end

Function: Ignored($u \rightarrow v$: edge, i: iteration number)
if $i > 1 \wedge u \rightarrow v$ is a loop-entry edge $\vee\ u \rightarrow v \in \delta^{start}$ **then**
  return **true**
end if
if $i < k \wedge u \rightarrow v \in \delta^{end}$ **then**
  return **true**
end if
if $i = k \wedge u \rightarrow v$ is the loop backedge **then**
  return **true**
end if
return **false**

The *Path-Numbering* algorithm (Algorithm 1) traverses the nodes in G_δ in a reverse topological order, computing the number of paths originating from each node, and assigning edge-weights to its outgoing edges (similar to the Ball-Larus algorithm). On entering a loop `L`, it starts off by computing the `val` and `numPaths` values for the last iteration of the loop (iteration `k`) — as the graph is traversed in a backward direction, the k^{th} iteration is seen first. However, on reaching the loop-head, it ignores the loop-entry edges and δ_L^{Start} (as decided by the `Ignored` function); as the list pointer is set back to the loop-tail, the algorithm is forced to follow the backedge again, performing another backward traversal of L, now computing the `val` and `numPaths` values for iteration `k-1`. The algorithm continues traversing the loop for a previous iteration each time till the first iteration is reached. On reaching the loop-head in the first iteration, it is the backedge that is ignored, and the algorithm exits the loop via the loop-entry edges and δ_L^{Start}. The vector `itrNum(n)` keeps track of the iteration number for which the computation of `numPaths(n)` and `val(n,i)` still needs to be done. It is easy to see that the k-iteration paths in G_δ — ignoring the edges ignored by the `Ignored` function — is exactly that in $G_{lu(k)}$.

The computation of `val` and `numPaths` is similar to the Ball-Larus algorithm; however, instead of being simple integers, `val` for an edge `e` is now a vector indexed by the iteration number. Also, `numPaths(n)` gives the number of paths from node `n` *corresponding to the last iteration for which this value was computed* (as the order of visiting the nodes in G_δ simulates a reverse topological traversal of $G_{lu(k)}$, the value of `numPaths(n)` for the last iteration for which `n` was visited is sufficient to compute `val` for all remaining iterations).

3.2.2. Eliminating invalid k-iteration paths. The loop-unrolled DAG, $G_{lu(k)}$, contains all the static k-iteration paths for a given graph G; however, the ***valid*** k-iteration paths are still lesser. The DAG $G_{lu(k)}$ also includes ***invalid*** k-iteration paths: paths in G that *begin at a loop-head* and *iterate through the loop fewer than k times*, exiting via a loop-exit edge in an iteration `i<k`. In $G_{lu(k)}$, these paths originate from a δ^{start} edge, and later follow an exit-edge of an i^{th} copy of the loop, where i<k.

For example, in Figure 4, for the paths from the `Start` node that reach `2'`, the ones beginning with the dummy edge $Start \rightarrow 2'$ and passing via $4' \rightarrow 6$ or $5' \rightarrow 6$ would be invalid as these paths originate from a dummy edge, and exit the loop in the first iteration itself; having three such paths, `invalidPaths(2')=3`.

In our algorithm, these paths are compensated via *compensation weights (cval):* an additional set of weights assigned to edges such that using $\sum_{e \in path} val(e,i) - cval(e,i)$ as the edge-weight instead of $\sum_{e \in path} val(e,i)$ for appropriate edges in an iteration `i`, still numbers the valid paths uniquely. This compensation is applied for an edge `e` in G_δ if:

- the edge `e` appears in a path `p` that commences with

$$path\text{-}identifier(p) = \sum_{e:u \to v \in p} val(e,i) - \begin{cases} cval(e,i) & \text{if } \delta_L^{start} \in p \wedge (e = \delta_L^{start} \vee (u \in L \wedge i < k)) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

δ_L^{start} (which implies that in G the path starts with a loop-head) i.e. $\delta_L^{start} \in p$; and

- either
  - `e` is the dummy edge δ_L^{start} i.e. $e = \delta_L^{start}$, or
  - $e : u \to v$ is either a loop-edge or a loop-exit edge, and `e` being traversed for the i^{th} iteration of the loop L, where `i<k` i.e $u \in L \ \wedge \ i < k$.

Hence, after compensation, the path-identifier for a valid k-iteration path is given by equation (1).

With these identifiers, not only are the paths numbered ***uniquely***, but also ***perfectly***: if there are `n` **valid** k-iteration paths through a node, the paths through it would be assigned path-identifiers from `0` to `n-1`, with no two valid paths having the same identifier.

Refer to the computation of `invalidPaths` and `cval` in Algorithm 1: `invalidPaths(n)` computes the number of invalid paths passing through the node `n` *for the last iteration for which this value was computed* if this node was reached via δ^{start}. Traversing the nodes of the loop in reverse topological order, the algorithm computes the number of such invalid paths through each node, and accordingly assigns compensation weights to the outgoing edges (the number of invalid paths through a node is equal to the sum of the invalid paths through all its successors). The number of invalid paths via a loop-exit edge $u \to v$ is accounted for by the number of possible paths through `v`: for each path that originates from an edge δ^{start} and reaches `u`, all paths that exit via the edge $u \to v$ are invalid — and that is exactly equal to the number of paths through `v`. The values for `cval(n,i)` is computed using the values for `invalidPaths(n)`; the computation is similar to how `val(n)` is computed using `numPaths(n)`.

Example 4: Figure 5 shows the values for `val` computed by the path-numbering algorithm. The weights for the two iterations are shown separated by '`/`'; the compensation weights for the iteration number 1 are indicated in the brackets. The edges not having any weights (or having a weight zero) are left unmarked.

Table 1 shows the execution trace of how the number of paths(n), the number of invalid paths(in), edge-weights(v), and compensation edge-weights(cv) get computed for this graph.

3.3. The Path Identification Algorithm

Given a path-identifier `n`, the *Path Identification* algorithm (Algorithm 2) finds a valid path from the `Start` node to the `End` node such that the sum of the edge-weights is equal to `n`. Beginning at the `Start` node, for each node identified to be in the path, it greedily select an outgoing edge with the highest `val`, but not allowing the partial path-identifier `pathWeight` (adjusted with the compensated edge-weight of the edge `e`) to exceed `n`. Once an edge `e` is selected to be on the path, `pathWeight` is updated by adding the compensated edge-weight of `e` to it. The edge-weight `val` is compensated with the compensation weight `cval` in the following cases:

- the current edge $u \to v \in \delta^{start}$; or
- the δ_L^{start} edge was included in the path to the current edge $u \to v$, $u \in L$; and the algorithm has not yet iterated through the loop L `k` times.

The algorithm accumulates edges till a whole path from the `Start` node to the `End` node is formed. The algorithm needs to be run on the graph G_δ, i.e. before the dummy edges are removed.

Table 1: An execution trace of the path-numbering algorithm. `S` and `E` represent the `Start` and `End` nodes respectively.

Reverse Topological Order : [E, 6, 5, 3, 4, 2, 1, S]

u	e	$i(u)$	$v(e,i(u))$	$n(u)$	$cv(e,i(u))$	$in(u)$
6	6 → E	1	**0**	**1**	**0**	-
5	5 → 6	2	**0**	**1**	**0**	-
5	5 → E	2	**1**	**2**	**0**	-
3	3 → 5	2	**0**	**2**	**0**	-
4	4 → 5	2	**0**	**2**	**0**	-
4	4 → 6	2	**2**	**3**	**0**	-
2	2 → 3	2	**0**	**2**	**0**	-
2	2 → 4	2	**2**	**5**	**0**	-
5	5 → 2	1	**0**	**5**	**0**	**0**
5	5 → 6	1	**5**	**6**	**0**	**1**
3	3 → 5	1	**0**	**6**	**0**	**1**
4	4 → 5	1	**0**	**6**	**0**	**1**
4	4 → 6	1	**6**	**7**	**1**	**2**
2	2 → 3	1	**0**	**6**	**0**	**1**
2	2 → 4	1	**6**	**13**	**1**	**3**
1	1 → 2	1	**0**	**13**	**0**	-
S	S → 1	1	**0**	**13**	**0**	**0**
S	S → 2	1	**13**	**26**	**0**	**3**

(u: selected node; e: selected edge; i: itrNum; v: val; n: numPaths; cv: cval; in: invalidPaths)

3.4. Correctness Results

Let `numPaths(n)` denote the number of *all paths* through a node n; `invalidPaths(n)` denotes the number of invalid paths through a node $n \in L$, when `n` lies on a longer path originating from the edge δ_L^{start}.

We prove the correctness results on the loop-unrolled DAG $G_{lu(k)}$ rather than the CFG G as it is actually $G_{lu(k)}$ that is traversed by the path-numbering algorithm (the condition $\neg(Ignored(e,i))$ controls this traversal). Each vertex-iterationNumber pair $\langle v, i \rangle, 1 \leq i \leq k$ forms an unique node in $G_{lu(k)}$ with itrNum($\langle v, i \rangle$)=i, indicating the copy of the loop that `v` belongs to. Hence, in our proofs, `val(v)` and

$$invalidPaths(u) = \sum_{v \in succ(u)} \begin{cases} numPaths(v) & \text{if } u \in L,\ itrNum(u) < k,\ u \to v \text{ is loop-exit edge of L} \\ invalidPaths(v) & \text{if } u, v \in L \text{ or } u\text{=}\texttt{Start} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Algorithm 2 Path Identification

Input

- *g: graph with dummy edges (G_δ)*
- *n: path-identifier whose path is to be identified*
- *k: the k-iteration profiling parameter*
- *val, cval: edge-weights and compensation edge-weights*

Output

- *path: set of edges identifying the path for identifier n*

begin
$u := Start$
$pathWeight := 0$
$path := [\,]$
$\forall x \in g$, set $visitCount(x) := 0$

while $u \neq End$ **do**
  $i := visitCount(u)$
  if u = loop-tail of loop L **then**
    Set $visitCount(x) := (i+1)$ for all nodes $x \in L$
  else
    $visitCount(u)$ += 1
  end if
  $i := visitCount(u)$
  $maxVal := -1$
  $maxWeight := 0$
  for all edges $e : u \to v \wedge \neg$ Ignored(e, i) **do**
    if $(e = \delta_L^{start}) \vee (\delta_L^{start} \cap path \neq \emptyset \ \wedge u \in L \wedge \ i \neq k)$ **then**
      $cweight := val(e,i) - cval(e,i)$
    else
      $cweight := val(e,i)$
    end if
    if $val(e,i) \geq maxVal \ \wedge \ pathWeight + cweight \leq n$ **then**
      $w := v$
      $maxVal := val(e,i)$
      $maxWeight := cweight$
    end if
  end for
  $pathWeight += maxWeight$
  Add $u \to w$ to $path$
  $u := w$
end while
end

`cval(v)` take single arguments (instead of `val(v, i)` and `cval(v, i)` as used in our algorithm).

In our proofs, we often consider the following disjoint set of paths:

- N: A set of **normal** paths that begin with the edge $start \to entry$, where `entry` is the actual entry-point to the procedure.
- D_i: A set of **dummy** paths that begin with the dummy edge $\delta_{L_i}^{start}$.

Lemma 1: The values `numPaths(v)` and `invalidPaths(v)` are computed correctly for each node `v` in the k-iteration loop-unrolled DAG.

Proof: The proof is by induction on the height of a node in the DAG i.e. the length of its longest path to the `End` node.

Base Case: It is trivially satisfied for height `H = 0`, i.e. for the `End` node.

Induction Step: Consider a node u at a height `H > 0`.

Computation of numPaths(v): Surely all paths from its successors are numbered uniquely and perfectly (by inductive hypothesis) as they are at a height less than `H` and the graph is a DAG. Also, the number of paths through a node is simply the sum of paths through all of its successors; thus, numPaths(u)=$\sum_{v \in Succ(u)} numPaths(v)$.

Computation of invalidPaths(v): The number of invalid paths through a node is equal to the sum of the invalid paths via all its outgoing edges: invalidPaths(u)=$\sum_{v \in Succ(u)} invalidPathsVia(u \to v)$.

Case I: If `u=Start` and $u \to v \in \delta^{start}$, then $invalidPathsVia(u \to v) = invalidPaths(v)$.

Case II: If $u \neq Start$ and `u` does not belong to a loop, $invalidPathsVia(u \to v) = 0$ as there cannot be such invalid paths outside loops.

Case III: Let $u \in L$, where `L` is a natural loop:

- If $v \notin L$ and $itrNum(u) = k$; then $u \to v$ is a loop-exit edge for `L`. Also, as $itrNum(u) = k$, all paths originating from δ_L^{Start} that reach `u` have already seen `k` iterations of `L`; hence, $invalidPathsVia(u \to v) = 0$.
- If $v \notin L$ and $itrNum(u) < k$; then $u \to v$ is a loop-exit edge for `L`. Also, as $itrNum(u) < k$, all paths originating from δ_L^{Start} that reach `u` have not yet seen `k` iterations of `L`. Hence, $invalidPathsVia(u \to v) = |\delta_L^{Start}| * numPaths(v)$ (where $|\delta_L^{Start}|$ denotes the number of δ_L^{Start} edges in the graph): a path originating from δ_L^{Start} that passes through the loop-exit edge $u \to v$ is an invalid k-iteration path (as the path began with the loop-head for `L,` but exited the loop via $u \to v$ before iterating through the loop `k`-times). Also, for a loop `X`, there can be only one δ_X^{Start} edge, so $|\delta_L^{Start}| = 1$.
- If $v \in L$, then $u \to v$ is a loop edge for `L`: $invalidPathsVia(u \to v) = invalidPaths(v)$.

To summarize, the invalid paths for a node `u` is given by equation (2).

□

Lemma 2: For all paths $p \in D_i$, the *valid* sub-paths from a node u are assigned *consecutive* path-identifiers (from `0` to `numPaths(u)-invalidPaths(u) -1`) by the path-numbering algorithm (while the last `invalidPaths(u)` path-identifiers remain unassigned).

Proof: We will prove the lemma by induction on the height of the DAG.

Base Case: The theorem is trivially satisfied for height `H=0`.

Inductive Step: Consider a node `u` at a height `H > 0`. Now, if $u \notin \{Start\} \cup L_i$, or $u \in L_i \wedge itrNum(u) < k$; then

`invalidPaths(u)=0` — as there are no invalid paths, in these cases the lemma is trivially satisfied.

Otherwise, the edge $u \rightarrow v_i$ may be:

- a loop-exit edge for the loop L_i with $itrNum(u) < k$: in this case, $invalidPathsVia(u \rightarrow v_i) = numPaths(v_i)$, i.e. all the paths through this edge are invalid. So, all the $numPaths(v_i)$ path-identifiers through $u \rightarrow v_i$ can be considered unassigned;
- either $\delta_{L_i}^{start}$, or an edge in L_i with $itrNum(u) < k$: in this case, $invalidPathsVia(u \rightarrow v_i) = invalidPaths(v_i)$, and all the valid paths through v_i are consecutively numbered from `0` to $numPaths(v_i) - invalidPaths(v_i) - 1$ by the induction hypothesis.

Thus, the valid paths through $u \rightarrow v_1$ (where v_1 is the first successor of u) are numbered from `0` to $(numPaths(v_1) - invalidPathsVia(u \rightarrow v_1)) - 1$. In general, the algorithm numbers all the valid paths till the edge $u \rightarrow v_i$ from `0` to $\sum_{j \leq i}(numPaths(v_j) - invalidPathsVia(u \rightarrow v_j)) - 1$. As the valid paths through $u \rightarrow v_{i+1}$ are also consecutively numbered, the paths till $u \rightarrow v_{i+1}$ get consecutively numbered from `0` to $\sum_{j \leq i+1}(numPaths(v_j) - invalidPathsVia(u \rightarrow v_j)) - 1$.

Hence, all valid sub-paths for the paths $p \in D_i$ through the node u get consecutively numbered from `0` to $\sum_{j \leq n}(numPaths(v_j) - invalidPathsVia(u \rightarrow v_j)) - 1$ which is exactly same as $numPaths(u) - invalidPaths(u) - 1$. The last $invalidPaths(u)$ path-identifiers, thus, remain unassigned.

□

Theorem 1: The path numbering algorithm uniquely and perfectly numbers all the valid paths.

Proof: We again prove the same by induction on the height `H` of the DAG. Let the successors of u be $v_1, v_2, \ldots, v_n$. The edge weights are set as, $val(u \rightarrow v_i) = \sum_{i<n} numPaths(v_i)$ and $cval(u \rightarrow v_i) = \sum_{i<n} invalidPaths(v_i)$.

We first prove the theorem for the sub-paths of each of the following disjoint set of paths separately:

1) If $p \in N$, all the paths are valid and hence no compensation weights are used.
   Base Case: The theorem holds trivially for height `H = 0`.
   Induction Step: Consider a node `u` at a height `H > 0`. By induction hypothesis, the sub-paths of `p` from each of its successor node v_i are numbered uniquely from `0` to $numPaths(v_i)$. As, the number of the sub-paths from u to all its successor from v_1 to v_{i-1} is $\sum_{j<i} numPaths(u \rightarrow v_j)$, the algorithm numbers the sub-paths through $u \rightarrow v_i$ from $\sum_{j<i} numPaths(u \rightarrow v_j)$ to $(\sum_{j<i} numPaths(u \rightarrow v_j)) + numPaths(v_i) - 1$ — these are clearly unique and perfect.
2) If $p \in D_i$, all paths through the loop-exit paths of the loop L_i are invalid for all copies of the loop till `k-1`. Base Case: The theorem holds trivially for height `H=0`. Induction Step: Consider a node `u` at a height `H > 0`. By induction hypothesis, the valid sub-paths of `p` from each of its successor node v_i are numbered uniquely from `0` to $numPaths(v_i) - invalidPaths(v_i) - 1$, while the last $invalidPaths(v_i)$ identifiers are unassigned by Lemma 2. We may assign these unassigned path identifiers to other valid sub-paths, i.e. ones that pass through the next successor node. So, ignoring all the invalid sub-paths, the valid sub-paths through $u \rightarrow v_i$ are numbered from $\sum_{j<i}(numPaths(v_j) - invalidPaths(v_j))$ to $\sum_{j<i}(numPaths(v_j) - invalidPaths(v_j)) + numPaths(v_i) - invalidPaths(v_i) - 1$. Note that numbering of sub-paths through each edge $u \rightarrow v_i$ leaves $\sum_{j \leq i} invalidPaths(v_j)$ path-identifiers of the $\sum_{j \leq i} numPaths(v_j)$ paths unassigned. Conceptually, to ignore the invalid paths through each of the edges $u \rightarrow v_i$, the path-identifiers for edges $u \rightarrow v_j, j > i$ are "shifted" by $invalidPaths(v_i)$. The total compensation (or the shift) that needs to be assigned to any edge $u \rightarrow v_i$ is $cval(u \rightarrow v_i) = \sum_{j<i} invalidPathsVia(v_j)$. These path identifiers for all the valid paths are:
   - ***unique*** as subtracting $cval(v_i)$ only allows assignment of the unassigned path-identifiers to other valid sub-paths but never overlaps it with already assigned path-identifiers of valid paths;
   - ***perfect*** as path-identifiers of all the invalid sub-paths are now assigned to valid sub-paths (or remain unassigned with values greater than the path-identifier of the last valid sub-path).

Now, consider the $Start$ node: the paths through its outgoing edges are numbered uniquely and perfectly as proved above. The algorithm also numbers the paths through an edge $start \rightarrow v_i$ in exactly the same manner: from $\sum_{j<i}(numPaths(v_j) - invalidPaths(v_j))$ to $\sum_{j<i}(numPaths(v_j) - invalidPaths(v_j)) + numPaths(v_i) - invalidPaths(v_i) - 1$. By exactly the same argument as above, the path-identifiers for all the paths from the `Start` node are also unique and perfect.

□

4. Program Instrumentation

4.1. Removal of the dummy edges

As the dummy edges do not represent actual control-flow, for instrumentation, their effect needs to be "simulated" by adjusting their weights on the weights of the actual control-flow edges appropriately. The dummy edges can be removed

in a manner exactly similar to that described by Ball and Larus[2]: to remove a dummy edge $Start \rightarrow v$, the weight on the edge is subtracted from all incident edges of `v` and added to all outgoing edges of `v`.

For k-iteration profiling, we also need to be careful about updating the correct *iteration* edge-weights for a loop `L`:

- on removing δ_L^{start}, the edge weights of only the first iteration of the loop `L` are affected;
- on removing δ_L^{end}, the edge weights of only the k^{th} iteration of the loop `L` are affected.

However, for the exit-edge of a loop `L`, the edge-weights for all the iterations of L may be affected on removal of a dummy edge of another loop `L'`.

For example, consider figure 8: the loop-unrolled DAG shows that on removing $\delta_{L_3}^{start}$, the edge-weight on it needs to be added to the edge-weights of both its incoming edges $3' \rightarrow 4'$ and $3'' \rightarrow 4'$; this indicates that the edge-weights for both the iterations of the edge $3 \rightarrow 4$ in the original graph needs adjustment. Similarly, on removing $\delta_{L_3}^{end}$, the edge-weights of both the iterations of $5 \rightarrow 6$ get affected.

Example 5: The adjusted graph weights on removing the dummy edges can be seen in Figure 6.

4.2. The Instrumentation Algorithm

The instrumentation algorithm (Algorithm 3) needs `k` accumulators ($c[0], c[1], \ldots, c[k-1]$), where `c[i]` accumulates the partial *(i+1)-iteration* path-identifier of a loop in execution; as k-iteration paths can at most be `k` iterations long, `k` such counters are sufficient. For example, say a program is executing the 20^{th} iteration of a loop — for two-iteration profiling, the counter `c[0]` will accumulate the partial path-identifier for the acyclic path executed in the 20^{th} iteration while `c[1]` will accumulate the two-iteration path-identifier spanning the 19^{th} and the 20^{th} iteration. The algorithm also needs a flag (f_δ) to remember if or not the k-iteration path starts with the loop-head (i.e. the path in the loop-unrolled DAG starts with δ^{start}). As, at least `k` iterations through a loop is needed to form a k-iteration path, f_δ also indicates if the loop has already seen k iterations. The saturating counter (`itr`) identifies the initiating `k-1` iterations of the loop. It is important to identify the iterations $i < k$, as till then the path-counters $c[j], j \geq i$ contain garbage values. After `k` iterations of the loop, all the path counters contain valid values.

The path-frequency table is maintained either as an array or a hashtable; the function `SelectImplementation` selects of one of them depending on the number of static paths in the current procedure. The function `IncrPathFreq(p)` increments the frequency count of the path with identifier `p` while profiling. Pointers to the path-frequency tables for each procedure are maintained in a program-wide global table. The initialization code, inserted in the respective procedure's `Start` node, initializes the local variables and selects an appropriate implementation of the path-frequency table.

To accumulate the path-identifiers, as in the Ball-Larus scheme, our algorithm splits appropriate edges and adds instrumentation code. For an edge `e` in the CFG, the i^{th} counter in the counter array, `c[i-1]`, is incremented with the edge weight `val(e,i)`, adjusted with the

Algorithm 3 The Instrumentation Algorithm: the notation `X<<Y` implies that the entity `X`, which can either be a node or an edge, is instrumented with profiling code `Y`.

Input

- *g: control-flow graph*
- *k: the k-iteration profiling parameter*
- *val, cval: edge-weights and compensation edge-weights after removing dummy edges*

Output

- *the instrumented program*

begin

Start « `c[0] := c[1] := ...:= c[k-1] := 0`; f_δ `:= false`; `itr := 0`; `SelectImplementation(numPaths(g))`

for all edges e ∈ cfg **do**
  for all i ∈ 0, 1, …, k-1 **do**
    if cval(e,i) ≠ 0 ∧ i ≠ k-1 **then**
      e « `c[i-1] +=` $\begin{cases} val(e,i) - cval(e,i) & \text{if } f_\delta\text{=true} \\ val(e,i) & \text{otherwise} \end{cases}$
    else
      if val(e,i) ≠ 0 **then**
        e « `c[i-1] += val(e,i)`
      end if
    end if
  end for
end for
for all k-iteration profiled loop-backedges e ∈ cfg **do**
  e « `if (`f_δ`) then IncrPathFreq(c[itr])`; `for i := k-1, k-2, ..., 1: c[i] := c[i-1]`; `c[0] := 0`; `if (itr < k-1) then {`; `itr += 1`; `if (itr = k-1) then` f_δ `:= true`; `}`
end for
for all k-iteration profiled loop-exit edges e ∈ cfg **do**
  e « `c[0] := c[itr]`; `for i := 1, 2, ..., k-1: c[i] := 0`; f_δ`:=false`; `itr := 0`
end for
for all acyclically profiled loop-backedges e ∈ cfg **do**
  e « `IncrPathFreq(c[0])`; `c[0] := 0`
end for
for all edges e:$[u \rightarrow End] \in cfg$ **do**
  e « `IncrPathFreq(c[0])`
end for
end

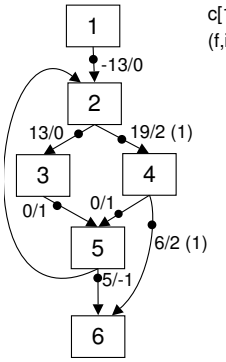


Figure 6: Removing dummy edges.

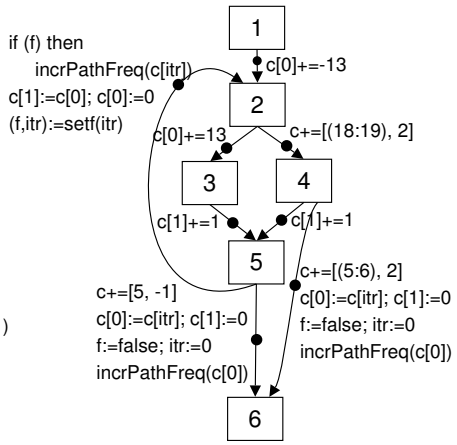


Figure 7: The graph shows the instrumented CFG for two-iteration profiling.

compensation-weight `cval(e,i)`, if any.

For the backedge of loops profiled using k-iteration profiling, the path-frequency table can be updated only after `k` iterations as a k-iteration path is not created until the end of the k^{th} iteration. Each path-accumulator `c[i]` loads the partial path-identifier currently held in `c[i-1]` and `c[0]` is cleared. The iteration counter `itr` is incremented (if not saturated); if `k` iterations of the loop have been seen, the flag f_δ is set.

For exit-edges of all k-iteration profiled loops, the partial path-identifier of the last valid path accumulator is transferred to `c[0]` — if the loop has been iterated `i` times, where `i<k`, then `c[i-1]` is loaded into `c[0]`; otherwise, `c[k-1]` is loaded into `c[0]` (this is because when the program is not executing in a loop, only `c[0]` is needed to accumulate the path identifier). The path accumulators `c[1..k-1]`, the iteration counter and the flag are cleared to make them ready the the next iteration.

The paths leading to the `End` node of the procedure is updated along the edges leading to the `End` node. For mixed-mode profiling, all acyclically profiled loop backedges have instrumentation code exactly as for the Ball-Larus scheme.

Example 6: Figure 7 shows the instrumented CFG for Figure 1 for two-iteration profiling. Table 2 shows how some of the two-iteration paths are identified for the trace `1-(2-3-5-2-4-5)`^{100}`-6` .

The notation used in the figure is explained below:

- the flag f_δ is shown simply as `f`;
- `c+=[a,b]` implies `{c[0]+=a; c[1]+=b;}`;
- `(a:b)` refers to the selection: (`f`$_\delta$`?a:b`);
- the function `setf()` updates the values of both f_δ and `itr` according to the following code segment: `if (itr < k-1) then { itr += 1; if (itr = k-1) then f := true; }`.

For example, when the edge $5 \rightarrow 2$ is seen for the first time, no edge is recorded as the value of `f` was `false` (see the previous row of the table entry). Hence, the algorithm was able to identify that the path seen till then, `1-2-3-5-2` is not a two-iteration path. The next time the edge is seen, the two-iteration path identifier `3` is identified and its frequency incremented, which corresponds to the two-iteration path `1-2-3-5-2-4-5`.

Table 2: Execution trace of the instrumented code.

Edge	c	f	itr	Incr	Path
$1 \rightarrow 2$	[-13, 0]	false	0		
$2 \rightarrow 3$	[0, 0]	false	0		
$3 \rightarrow 5$	[0, 1]	false	0		
$5 \rightarrow 2$	[0, 0]	true	1		
$2 \rightarrow 4$	[18, 2]	true	1		
$4 \rightarrow 5$	[18, 3]	true	1		
$5 \rightarrow 2$	[0, 18]	true	1	3	1-2-3-5-2-4-5
$2 \rightarrow 3$	[13, 18]	true	1		
$3 \rightarrow 5$	[13, 19]	true	1		
$5 \rightarrow 2$	[0, 13]	true	1	19	2-4-5-2-3-5
$2 \rightarrow 4$	[18, 15]	true	1		
$4 \rightarrow 5$	[18, 16]	true	1		
$5 \rightarrow 2$	[0, 18]	true	1	16	2-3-5-2-4-5
...	...	...	...	...	...
$4 \rightarrow 5$	[18, 16]	true	1		
$5 \rightarrow 6$	[15, 0]	false	0	15	2-3-5-2-4-5-6

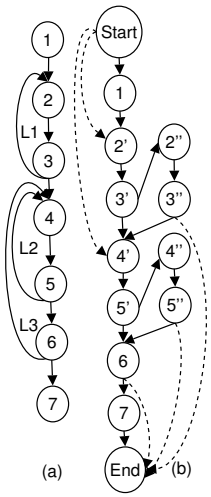
5. Mixed-Mode Profiling: Profiling different regions with varied length paths

The k-iteration profiling algorithm can be used in mixed-mode: different regions of the same procedure can be profiled with different values of k, thereby keeping the number of static paths in check — the "interesting" regions may be profiled using larger values of k while using acyclic profiling for the remaining procedure.

Mixed-mode profiling is also useful for nested loops: k-iteration profiling of nested loops needs multiple sets of edge weights for the inner loop — one of each iteration of the outer loop. Instead, one may use k-iteration profiling for the innermost loop and acyclic profiling for the outer loops. If the profiling information for an outer loop is desired, all the contained inner loops can be collapsed into single node, effectively reducing the outer loop into an innermost loop. Such schemes for profiling programs at various granularity is suggested in [5], [6].

Figure 8 illustrates mixed-mode profiling: it shows an example graph, its mixed-mode loop-unrolled DAG and the static paths recognised by the mixed-mode profiler.

It is easy to see why the path-numbering and path-identification algorithms work for mixed-mode profiling; in the interest of space, we provide an informal argument using Figure 8(a):



1-2-3-2-3-4-5-4-5-6-7	1-2-3-4-5-6-7
1-2-3-2-3-4-5-4-5-6	1-2-3-4-5-6
1-2-3-2-3-4-5-4-5	2-3-2-3-4-5-4-5-6-7
1-2-3-2-3-4-5-6-7	2-3-2-3-4-5-4-5-6
1-2-3-2-3-4-5-6	2-3-2-3-4-5-4-5
1-2-3-2-3	2-3-2-3-4-5-6-7
1-2-3-4-5-4-5-6-7	4-5-4-5-6-7
1-2-3-4-5-4-5-6	4-5-4-5-6
1-2-3-4-5-4-5	4-5-4-5

Figure 8: Mixed-mode profiling: The loop L_3 is profiled with `k=1` (acyclically), while loops L_1 and L_2 with `k=2` (the backedge $6 \rightarrow 4$ occurs at most once in the paths while the backedges $3 \rightarrow 2$ and $5 \rightarrow 4$ occur twice in some of the paths).

- Consider disjoint loops that use different values of the parameter `k`: Assume the absence of the edge $6 \rightarrow 4$. Let L_1 use `k=`k_1 while L_2 use `k=`k_2. As the path-numbering algorithm numbers the paths in a bottom-up fashion, the paths from node `4` are still numbered correctly as the value k_1 does not influence it in any way. Similarly, numbering paths for the nodes of L_1 is only dependent on the number of paths through the node `4` (which is computed correctly by the argument above). Hence, all the nodes in L_1 are also numbered correctly.
- Consider nested-loops where the outer loop is acyclically profiled: Let L_2 be k-iteration profiled while L_3 is acyclically profiled. Figure 8(a) shows the loop-unrolled DAG for this case; the dummy edges $Start \rightarrow 4'$ and $6 \rightarrow End$ are added to take care of acyclic paths due to backedge $6 \rightarrow 4$. It is easy to see that the path-numbering algorithm would still be able to number all the paths in this DAG uniquely and perfectly.

The key idea in mixed-mode path-numbering is: while numbering a node `n`, we just need to know *the number of paths through its successor node*s and not *how the paths in the successor nodes were numbered*.

6. Experimental Results

We have implemented acyclic and k-iteration profiling for C source code using Lance[7]. It profiles each function in a given program, using separate tables for keeping the path frequency counts for each procedure. The profiler attempts to use an array-based implementation for a path-frequency table if the number of static paths is low, but switches to a hashtable-based implementation if a procedure has more than 6000 static paths. It completely aborts profiling a procedure if the number of static paths in it exceeds 60,000. For nested loops it uses mixed-mode profiling: only the innermost loop is k-iteration profiled while the outer loops are acyclically profiled. Our implementation, both of the acyclic and k-iteration path-profiler, currently lacks the optimizations described by Ball-Larus, like optimizing the number of edges to be instrumented [1] and replacing the first increment of a counter by a load instruction (which also removes the initialization of counters).

The cost of path profiling can be classified into two categories: *computational cost* (the overhead of the instrumentation code that computes the path-identifiers for each k-iteration path traversed) and the *table-update cost* (the cost of updating the path-frequency table using the path-identifiers). Path profilers generally use an array-based implementation of the path-frequency table if the number of static paths is small; they switch to an expensive hashtable-based implementation if the number of static paths crosses a certain threshold. We use small programs[8] as benchmarks so that the frequency counts for both acyclic, two- and three-iteration profiles can be kept in an array (for most cases) to give an exact comparison of the computational overhead of k-iteration profiling. We also show how fast the number of static paths grow in these programs, and instances where the use of hashtable slows down the programs significantly, to indicate the table-update cost of k-iteration profiling.

Table 3: Slowdown factor of k-iteration profiling w.r.t acyclic path (BL) profiling ($ExcTime_{k-itr}/ExcTime_{BL}$).

Program	$\frac{2-Itr}{BL}$	$\frac{3-Itr}{BL}$
compress	1.14	3.92
crc	1.12	1.20
edn	1.56	1.87
fdct	1.06	1.10
fibcall	1.09	1.19
fir	1.57	1.75
jfdctint	1.00	1.10
ludcmp	1.21	4.47
ndes	6.12	—

Table 3 shows the program slowdown factors (ratio of the instrumented program's running times for collecting k-iteration profiles over that for collecting acyclic path profiles): for most of the benchmarks, the ratio is small — indicating that the computational cost does not increase significantly from acyclic to k-iteration profiling.

A few programs, however, show comparatively large slowdowns (`compress` and `ludcmp` for three-iteration profiling, and `ndes` for two-iteration profiling). Table 4 gives a comparison of how the number of static paths increase for two of these programs. The program `ndes` slows down significantly because the profiler switches to a hashtable-based implementation of the path-frequency table for the function `cyfun` (`14294` two-iteration static paths). As this function has more than `60,000` three-iteration static paths, the three-iteration profiler aborts profiling this function; hence, we do not provide any data for this case. For `compress`, the slowdown is very small for two-iteration profiling but increases significantly for three-iteration profiling due to the same reason — the number of static paths in

Table 4: The table shows the number of static BL paths, and k-Iteration paths for varying values of k. (× indicates that path-numbering for the procedure was aborted as the number of static paths in the procedure exceeded 60,000).

Program	Function	BL	k=2	k=3	k=4	k=5
compress	*compress*	713	5887	44063	×	×
	cl_hash	10	16	24	34	46
	writebytes	27	84	255	768	2307
	output	416	1328	4064	12272	36896
ndes	*des*	568	5318	35800	×	×
	ks	276	466	710	1008	1360
	cyfun	429	14294	×	×	×

the function `compress` increases from `5887` two-iteration paths to `44063` three-iteration paths; thus requiring the hashtable-based implementation for three-iteration profiles. The reason for the slowdown of `ludcmp` is also same: the function `ludcmp` in this program has `4144` two-iteration paths and `9110` three-iteration paths.

7. Related Work

Tallam et al. [3] used Ball-Larus paths to record slightly longer overlapping paths, and proposed an instrumentation algorithm to collect their frequencies; these overlapping paths were then used to estimate the frequencies of much longer paths. The cost of collecting the path frequencies was found to be around 4.2 times that of Ball-Larus paths. The average imprecision in estimating the flows derived from overlapping path frequencies was found be to low — from -4% to +8%. The k-iteration profiling algorithm, on the other hand, is designed to collect the exact path frequencies for longer paths.

Structural Path Profiling (SPP) [6] is an interesting idea of partitioning a procedure into a hierarchy of nested graphs based on the loop structure, and then to profile each graph independently. The method has been proposed as an efficient online path profiling technique for JIT compilers. k-iteration mixed-mode path profiling can be combined with SPP — the different partitions may be profiled with different path lengths (by varying k) depending on the hotness of the respective code section.

Hierarchical Path Profiling [5] represents inner program regions as single nodes in the outer regions. Path profiles are collected separately for the different regions, which can be used by region-based compilers to drive local optimizations. As these regions would typically be small in size, hierarchical k-iteration profiling seems to be an interesting possibility.

Preferential Path Profiling (PPP) [4] is a recent attempt at profiling selective paths in a procedure, thereby reducing the profiling overheads. Preferential k-iteration profiling, profiling some of the selected paths with longer path lengths, seems to be an useful option.

8. Conclusions

Ball and Larus proposed an algorithm to profile acyclic paths in a program efficiently; k-iteration path profiling is a generalization of the Ball-Larus algorithm, allowing paths spanning multiple loop iterations. Such longer paths would be useful in program understanding and profile-directed compiler optimizations. The k-iteration profiling algorithm allows mixed-mode profiling: profiling different parts of the same procedure with different path lengths, allowing the user to focus on the interesting regions, while keeping the profiling cost low. A base profiler can be used to indicate hot regions of a procedure; k-iteration profiling can then be used on these hot regions to generate better information. Use of k-iteration profiling in combination with some other recent efforts in profiling like SPP [6] and PPP [4] seems promising; we are interested in exploring such schemes in the future.

References

[1] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Trans. Program. Lang. Syst.*, 16(5):1399–1410, 1994.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[3] S. Tallam, X. Zhang, and R. Gupta. Extending path profiling across loop backedges and procedure boundaries. In *In International Symposium on Code Generation and Optimization (CGO)*, pages 251–264, 2004.

[4] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. *SIGPLAN Not.*, 42(1):351–362, 2007.

[5] Y. Wu, A. Adl-Tabatabai, D. Berson, J.Z. Fang, and R. Gupta. US Patent 6,848,100 : Hierarchical Software Path Profiling, 2005.

[6] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for java just-in-time compilers. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 148. IEEE Computer Society, 2003.

[7] Lance C Compiler. *http://www.lancecompiler.com*.

[8] WCET Project/Benchmarks. *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*.