Preferential Path Profiling: Compactly Numbering Interesting Paths

Kapil Vaswani
Indian Institute of Science, Bangalore
kapil@csa.iisc.ernet.in

Aditya V. Nori
Microsoft Research India
adityan@microsoft.com

Trishul M. Chilimbi
Microsoft Research
trishulc@microsoft.com

Abstract

Path profiles provide a more accurate characterization of a program's dynamic behavior than basic block or edge profiles, but are relatively more expensive to collect. This has limited their use in practice despite demonstrations of their advantages over edge profiles for a wide variety of applications.

We present a new algorithm called preferential path profiling (PPP), that reduces the overhead of path profiling. PPP leverages the observation that most consumers of path profiles are only interested in a subset of all program paths. PPP achieves low overhead by separating interesting paths from other paths and assigning a set of unique and compact numbers to these interesting paths. We draw a parallel between arithmetic coding and path numbering, and use this connection to prove an optimality result for the compactness of path numbering produced by PPP. This compact path numbering enables our PPP implementation to record path information in an array instead of a hash table. Our experimental results indicate that PPP reduces the runtime overhead of profiling paths exercised by the largest (ref) inputs of the SPEC CPU2000 benchmarks from 50% on average (maximum of 132%) to 15% on average (maximum of 26%) as compared to a state-of-the-art path profiler.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; E.4 [*Coding and Information Theory*]: Data Compaction and Compression

General Terms Algorithms, Measurement, Reliability

Keywords Profiling, preferential paths, arithmetic coding, dynamic analysis

1. Introduction

Path profiles are a succinct and pragmatic abstraction of a program's dynamic control-flow behavior. Recording program paths has proved valuable in a wide variety of areas such as computer architecture, compilers, debugging, program testing, and software maintenance [4]. Path profiles capture much more control-flow information than basic block or edge profiles, and are much smaller than complete instruction traces. Several compiler optimizations perform better when trade-offs are driven by accurate path profiles [1]. Program paths are also a more credible way of measuring coverage of a given test suite. In addition, abstractions of paths can help automatic test generation tools generate more robust test cases. Finally, program path histories often serve as a valuable debugging aid by revealing the instruction sequence executed in the lead up to interesting program points.

Unfortunately, the benefits of using path profiles come at a cost – profiling paths is expensive. Our measurements of an implementation of a state-of-the-art path profiler [3] indicate an average execution time overhead of 50% with as much as a 132% overhead in the worst case and other studies report similarly high overheads [6]. This high overhead has limited the use of path profiles in favor of basic block or edge profiles. Basic block and edge profiles are cheaper to collect but less accurately capture a program's dynamic behavior as compared to paths. While Ball et al. found that 80% of program paths could be attributed from an edge profile [5], more recent work found that just 48% of paths could be attributed from an edge profile [6]. In both cases, the most complex (and likely most interesting) paths were not predictable from an edge profile. This represents an opportunity as basic block and edge profiles are still preferred over path profiles for measuring test coverage and driving profile-guided optimizations such as code placement, inlining, unrolling, and superblock scheduling.

Apart from these traditional usage scenarios, we envisage the use of path profiling in several other cost-sensitive enviornments. For instance, in residual path profiling, a user is interested in determining the set of paths that a deployed program executed in the field that were not exercised during testing. This information could be used to improve and augment test suites, and if included with bug reports resulting from field failures, could help pinpoint the root cause of errors. Another scenario involves ascertaining whether paths that were identified as hot paths during testing and used to optimize the program continue to remain hot during field usage. In addition, we might want to gather detailed information about these paths, such as cache misses, page faults, and variations in execution time, without resorting to sampling techniques [12]. Finally, we might be interested in efficiently tracking a subset of paths in deployed software that meet a certain criteria, for example, paths that access safety or security critical resources, or that exercise an error prone code region. A common trait in all these scenarios is the need for efficiently and accurately profiling a known subset of paths.

Let us first examine why existing path profiling schemes incur relatively high overhead. The efficient path profiling scheme proposed by Ball and Larus, which forms the basis of all path profilers, assigns weights to edges of a control flow graph (CFG) such that all paths are allocated unique identifiers (i.e., the sum of the weights of the edges along every path is unique) [3]. During program execution, the profiler accumulates weights along the edges and updates an array entry that corresponds to this path identifier. Unfortunately, for functions with a large number of paths, allocat-

POPL'07 January 17–19, 2007, Nice, France.

procedure *computeBLIncrements* (G)
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) $W : E \rightarrow \mathbb{Z}$ is an empty map.
Returns: The map W defined for all edges such that every path in G is assigned a unique weight.

1: $N_t := 1$;
2: **for all** nodes $v \in V$ in reverse topological order **do**
3: $N_v := 0$;
4: **for all** edges $e \in out(v)$ **do**
5: $W(e) := N_v$;
6: $N_v := N_v + N_{\mathsf{dest}(e)}$;
7: **end for**
8: **end for**

Figure 1. The Ball-Larus Algorithm.

ing an array entry for all program paths is prohibitively expensive, if not infeasible. Consequently, path profiler implementations are forced to use a hash table to record path information for such functions. Although using a hash table is space efficient as program's typically execute only a small subset of all possible paths, it incurs significantly higher execution time overhead as compared to updating an array entry. Previous work has shown that hash tables account for a significant fraction of the overhead attributable to path profiling [8].

To address this problem, we propose *preferential path profiling* (PPP), a novel path profiling scheme that efficiently profiles arbitrary path subsets, which we refer to as *interesting paths*. Our algorithm can be viewed as a generalization of the Ball-Larus algorithm, which forms the core of most existing path profiler implementations. As mentioned earlier, the Ball-Larus algorithm assigns weights to the edges of a given CFG such that the sum of the weights of the edges along each path through the CFG is unique. Our algorithm generalizes this notion to a subset of paths; it assigns weights to the edges such that the sum of the weights along the edges of the interesting paths is unique. Furthermore, our algorithm attempts to achieve a minimal and compact encoding of the interesting paths; such an encoding significantly reduces the overheads of path profiling by eliminating expensive hash operations during profiling. In addition, our profiling scheme separates interesting paths from other paths and is able to classify paths during program execution. The ability to classify paths is important for many scenarios such as residual path profiling described earlier.

Interestingly, we find that both the Ball-Larus algorithm and PPP are essentially a form of arithmetic coding [13, 15], a technique commonly used for universal data compression. We make use of this connection to prove an optimality result for the compactness of path numbering produced by PPP. We have implemented PPP and our experimental evaluation using benchmarks from the SPEC CPU2000 suite shows that PPP reduces the overheads of profiling paths exercised by their largest (ref) inputs from 50% on average (maximum of 132%) to 15% on average (with a maximum of 26%) as compared to Ball-Larus profiling.

This paper makes the following main contributions. First, we describe a new algorithm, called preferential path profiling (PPP), for compactly numbering arbitrary path subsets that improves upon Ball-Larus numbering (Section 3). Next, we draw a parallel between arithmetic coding and path numbering, and use this connection to prove an optimality result for the compactness of path numbering produced by PPP (Section 4). Finally, we present an experimental evaluation of our PPP implementation that demonstrates that it results in significantly lower overheads than Ball-Larus profiling (Section 5).

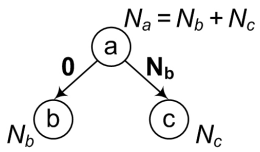

Figure 2. Assignment of weights to edges using the Ball-Larus algorithm

2. Preliminaries

In this section, we briefly describe the Ball-Larus algorithm for profiling acyclic, intra-procedural paths through a CFG of a program and motivate our problem using a simple example.

2.1 Definitions

Profiling algorithms for acyclic, intra-procedural paths (henceforth referred to as *paths*) first convert the CFG of a procedure into a *directed acyclic graph* (DAG). Each DAG is a graph $G = (V, E, s, t)$, where V represents nodes or basic blocks in the procedure, and E is the set of edges between nodes. The maps $src(e)$ and $dest(e)$ denote the source and destination nodes respectively, of an edge e. For every node $v \in V$, $out(v)$ denotes the set of edges emanating from v in G, and $succ(v)$ represents all the immediate successor nodes of v. Each acyclic, intra-procedural path p is a sequence of nodes from s to t. The function $paths(G)$ refers to all acyclic, intra-procedural paths in G. The function $paths_G(e) : E \rightarrow 2^{paths(G)}$ represents all paths in G that traverse an edge e. Conversely, the function $edges : P \rightarrow 2^E$ maps every path p in G to the set of edges that belong to p.

An assignment of weights to the edges of G is represented as a map $W : E \rightarrow \mathbb{Z}$ (where $\mathbb{Z}$ is the set of integers). The relation $pathid : P \rightarrow \mathbb{Z}$ maps each path to a *path identifier*, and is defined as follows.

$$pathid(p) \stackrel{\mathsf{def}}{=} \sum_{e \in edges(p)} W(e)$$

2.2 Ball-Larus Profiling

Given a DAG G for a procedure, the Ball-Larus algorithm assigns weights to the edges of the graph such that for every path $p \in paths(G)$, $pathid(p)$ is unique, and is equal to a number between 0 and $N - 1$, where $N = |paths(G)|$. The algorithm *computeBLIncrements*, shown in Figure 1, performs one bottom-up pass through G and processes its nodes in reverse topological order. With each node v, the algorithm associates a count N_v that indicates the number of paths from v to the exit node t (Line 5). At each node, the *computeBLIncrements* traverses the list of successor nodes and assigns weights to the corresponding outgoing edges. This algorithm is based on a simple idea that is stated in the following lemma [3].

LEMMA 1. *Let $G = (V, E, s, t)$ be a DAG. The number of paths from any node v in G to the exit node t is equal to the sum of the number of paths from each of v's successor nodes to t.*

Figure 2 illustrates how *computeBLIncrements* uses this invariant to compute an edge assignment. Assume that the algorithm is processing node a with two successors b and c that have N_b and N_c paths to the exit node t. Also assume that these paths have already been assigned identifiers from 0 to $N_b - 1$ and $N_c - 1$ respectively. The algorithm assigns a weight 0 to the edge (a, b), and a weight N_b to the edge (a, c). This ensures that the paths from a to t are assigned identifers from 0 to $N_b + N_c - 1$, which is also equal to $N_a - 1$ (from Lemma 1). In general, the weight assigned to an

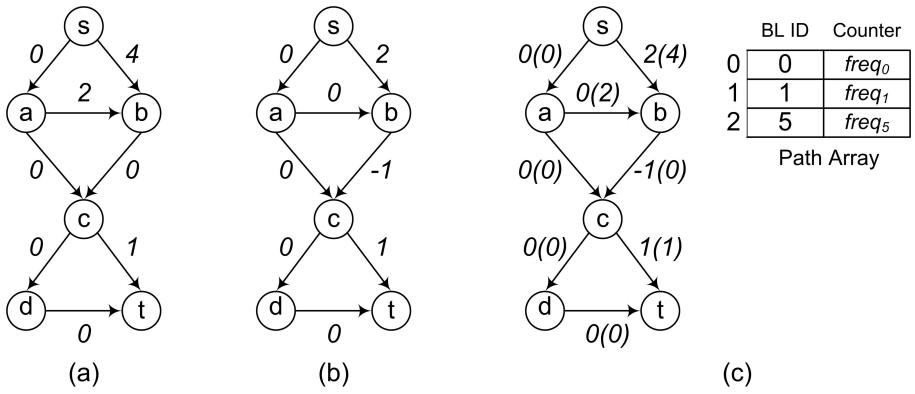

Figure 3. Motivating example for PPP. (a) A DAG G with 6 paths with edges numbered using the Ball-Larus algorithm. (b) G with edges having only PPP assigned numbers for three interesting paths $I = \{sacdt, sact, sbct\}$. (c) G with edges assigned two numbers, a PPP number and a Ball-Larus number (in parenthesis). The path array is accessed using the PPP counter.

procedure *computePathIdentifier* (G, W, p)
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) The map $W : E \rightarrow \mathbb{Z}$.
(c) A path p is a sequence of nodes through G.
Returns: The path identifier for the path p
1: return $\sum_{e \in edges(p)} W(e)$;

Figure 4. Computing the Ball-Larus identifier of a path from an edge assignment.

edge is equal to the sum of the number of paths from all previously processed successor nodes to t (Line 6).

The Ball-Larus path profiler instruments the edges of the CFG with instructions to increment a counter by the weight assigned to the edge. When the instrumented program executes, it simulates the procedure *computePathIdentifier* (Figure 4). When a path terminates, the value in the counter represents the path that just executed and can be used for book-keeping.

2.3 An illustrative example

We start with an example that illustrates a drawback of the Ball-Larus profiling scheme, and also shows how our profiler works on this example. Consider the function in Figure 3(a). The DAG G in Figure 3(a) is obtained from the CFG of the function. This figure also shows the weights assigned by the Ball-Larus algorithm to edges of G. Note that the sum of the weights of edges along every path from the start node s to the final node t is unique, and all paths are allocated identifiers from 0 to $N-1$, where N is the total number of paths from s to t. If N is reasonably small (less than some threshold value), the profiler can allocate an array of counters of size N, and track path frequencies by indexing into the array using the path identifier and incrementing the corresponding counter. However, the number of potential paths in a procedure can be arbitrarily large (exponential in the number of nodes in the graph) and allocating a counter for each path can be prohibitively expensive, even infeasible in many cases. Path profilers overcome this problem by using a hash table of counters instead of an array, relying on the fact that only a small number of paths are traversed during any given execution. Therefore, a combination of a suitably sized hash table and a good hash function almost always guarantees the absence of conflicts. In the current example, if the threshold value is set to 4, the Ball-Larus profiler would use a hash table since there are 6 paths from s to t.

Let us assume that we are interested in profiling only a subset $I = \{sacdt, sact, sbct\}$ (interesting paths) of paths. The Ball-Larus identifiers for the paths $sacdt$, $sact$, $sbct$ are 0, 1 and 5 respectively. This means that one would have to allocate a hash table even though there are only 3 paths of interest. In such a scenario, it would be ideal if we could compute an edge assignment that allocates identifiers 0, 1 and 2 to these paths, and identifiers > 2 to the other paths. In Section 3, we show that computing an edge assignment W and a number β such that (a) $\forall p \in I$, $pathid(p) \leq \beta$, and (b) $\forall p \notin I$, $pathid(p) > \beta$ is not always feasible. Therefore, we relax the constraints on this problem by eliminating condition (b) (which is a condition over uninteresting paths), and ask the question if it is possible to label the edges in G such that the paths in the set I have path identifiers in $\{0, 1, 2\}$. Figure 3(b) shows that such an assignment of weights to edges indeed exists, and this is precisely the assignment computed by the preferential path profiling PPP algorithm described in Section 3. Therefore, our profiler incurs lower overheads since we can now use an array to track frequencies instead of a hash table. Note that while the interesting paths $sacdt$, $sact$, $sbct$ have been assigned unique identifiers from 0 to 2, the uninteresting paths $sabct$ and $sbcdt$ alias with the interesting paths $sacdt$ and $sact$ respectively. We resolve these "aliases" using Ball-Larus path identifiers, which are unique for every path. In PPP, edges are annotated with a second weight computed using the Ball-Larus algorithm (these weights are shown in parentheses in Figure 3(c)). The profiler also stores the Ball-Larus identifiers of all interesting paths along with their counters. The occurrence of an interesting path can be detected by comparing the Ball-Larus identifier computed during the traversal with the Ball-Larus identifier stored in the array – a match indicates that an interesting path was just traversed and vice versa. For example, when the uninteresting path $sbcdt$ (PPP identifier is 2 and Ball-Larus identifier is 4) occurs, before incrementing the count at index 2 in the path array, the Ball-Larus identifier at index 2 is compared with the Ball-Larus identifier of $sbcdt$ – since they are different, the profiler infers that this path is not interesting (or it might be a residual path not exercised by the test suite) and takes necessary action.

3. Preferential Path Profiling

We will now address the problem of encoding arbitrary subsets of paths over a DAG $G = (V, E, s, t)$. Informally, we wish to compute an edge assignment that allows us to uniquely identify paths as well as differentiate interesting paths from uninteresting ones. First, consider the possibility of finding an edge assignment

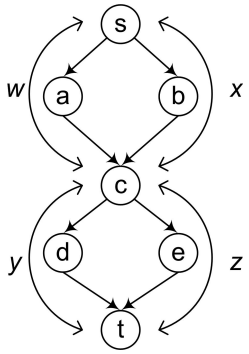

Figure 5. A counterexample for separation of paths.

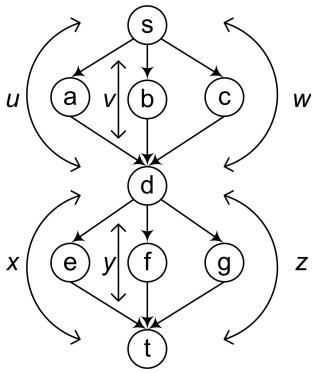

Figure 6. A counterexample for perfect edge assignment.

that *separates* interesting and uninteresting paths using a non-negative integer $\beta \in \mathbb{Z}^{\geq 0}$.

LEMMA 2 (Separation of paths). *Given a* DAG $G = (V, E, s, t)$ *and a set of interesting paths* $I \subseteq paths(G)$*, a map* $W : E \rightarrow \mathbb{Z}$ *that satisfies the following conditions may not always exist.*

1. uniqueness: $\forall p, q \in I, pathid(p) \neq pathid(q)$.
2. separation: $\exists \beta \in \mathbb{Z}^{\geq 0}$ *such that*
   (a) $\forall p \in I, pathid(p) \leq \beta$*, and*
   (b) $\forall p \notin I, pathid(p) > \beta$.

Proof: Consider the simple DAG in Figure 5. Assume that we are interested in profiling paths $sacet$ and $sbcdt$. Assume that there exists an edge assignment W that satisfies all conditions in the lemma. Let w, x, y and z represent the cumulative weights of the sub-paths sac, sbc, cdt and cet respectively. From condition 2(a), we have $w + z \leq \beta$ and $x + y \leq \beta$, and from condition 2(b), it follows that $w + y > \beta$ and $x + z > \beta$. This implies that

$$\begin{aligned} x + y + w + z &\leq 2\beta \\ x + y + w + z &> 2\beta \end{aligned}$$

which is a contradiction and the lemma follows. ■

We find that separating arbitrary sets of interesting and uninteresting paths is almost always infeasible, primarily due to the presence of many shared edges. We therefore simplify the problem by relaxing condition 2(b) in Lemma 2. Edge assignments that cause interesting paths to alias with uninteresting paths are acceptable as long as the interesting paths are assigned minimal unique identifiers. As described in Section 2.3, a second counter that computes the Ball-Larus identifiers of all paths can be used to resolve the aliases. This relaxation allows us to reason about interesting paths only, an aspect critical to the solution we propose. However, it turns out the even this simplified problem may not have a perfect solution as the following lemma indicates.

LEMMA 3 (Perfect edge assignment). *Given a* DAG $G = (V, E, s, t)$ *and a set of interesting paths* $I \subseteq paths(G)$*, a map* $W : E \rightarrow \mathbb{Z}$ *that satisfies the following conditions may not always exist.*

1. uniqueness: $\forall p, q \in I, pathid(p) \neq pathid(q)$,
2. perfect assignment: $\forall p \in I, 0 \leq pathid(p) < |I|$.

Proof: Consider the graph in Figure 6. Say we are interesting in profiling the paths $sadft, sadgt, sbdet, sbdgt, scdet$, and $scdft$. For simplicity, we represent the sum of the edges along the sub-paths sad, sbd, scd, det, dft and dgt as u, v, w, x, y and z. Consider the sum of the identifiers of all these paths.

$$\begin{aligned} sum &= (u + y) + (u + z) + (v + x) \\ &\quad +(v + z) + (w + x) + (w + y) \\ &= 2(u + v + w + x + y + z) \end{aligned}$$

Hence, the sum of the path identifiers of these paths is necessarily even. However, for a perfect assignment, these paths must be allocated identifiers between 0 and 5. Since the sum of numbers from 0 to 5 is odd, we conclude that a perfect edge assignment for this graph and set of interesting paths does not exist. ■

Since a perfect edge assignment for interesting paths may not always exist, even an optimal edge assignment may induce a path assignment with "holes" in the interval of path identifiers. In light of this lemma, we restate our problem as follows.

Problem A (Optimal Edge Assignment): Given a DAG $G = (V, E, s, t)$ and a set of interesting paths $I \subseteq paths(G)$, compute an edge assignment $W : E \rightarrow \mathbb{Z}$ that satisfies the following conditions.

1. uniqueness: $\forall p, q \in I, pathid(p) \neq pathid(q)$,
2. compactness: The compactness measure δ defined by

$$\delta \stackrel{\text{def}}{=} \frac{(\max_{p \in I} pathid(p) - \min_{p \in I} pathid(p)) + 1}{|I|}$$

is minimized.

It is easy to see that $\delta \geq 1$. A perfect edge assignment W induces a $\delta = 1$. Lemma 3 shows that a solution with $\delta = 1$ does not always exist. Hence solutions with lower values of δ are preferred. We find that for arbitrary graphs and arbitrary set of paths, even characterizing the optimal δ seems to be a hard problem. In the next section, we propose an algorithm that computes an edge assignment that attempts to minimize δ, and later prove an optimality result for this algorithm by establishing a connection with arithmetic coding.

3.1 The Preferential Path Profiling Algorithm

The preferential path profiling (PPP) algorithm is a generalization of the Ball-Larus algorithm with the added capability of biasing the edge assignment towards an arbitrary set of interesting paths. Before we describe the algorithm, we introduce some notation and state some of the key observations that the algorithm is based on.

Let $G = (V, E, s, t)$ be a DAG, and let $I \subseteq paths(G)$ be a set of interesting paths. Consider a node $v \in V$ and an edge $e \in out(v)$. Let $paths_I(e)$ represents the set of interesting paths

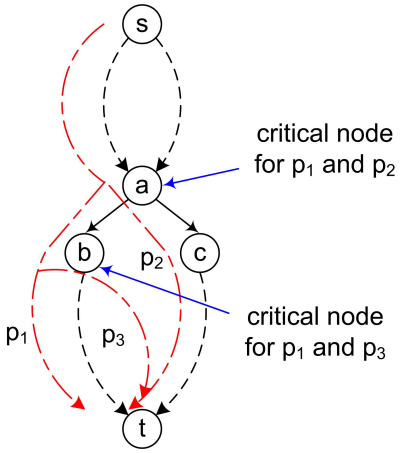

Figure 7. Critical nodes for pairs of paths.

that contain the edge e. Let $prefix(p, e)$ denote the sequence of nodes from s to $src(e)$ along the path p. Then $prefix_I(e)$ denotes the set of all prefixes $\{prefix(p, e)\}_{p \in I}$. Given a pair of interesting paths $p, p' \in I$, $lcp(p, p')$ represents the longest common prefix of p and p' in G. Note that $lcp(p, p')$ is trivially the start node s if p and p' are edge disjoint. We define the *critical node* for a pair of interesting paths p and p' as follows.

$$critical(p, p') \stackrel{\text{def}}{=} \textit{the last node in } lcp(p, p')$$

For any pair of paths, the critical node is unique since the longest common prefix is uniquely defined. For example, in Figure 7, node a is the critical node for paths p_1 and p_2, whereas node b is the critical node for paths p_1 and p_3. Again, the critical node for a set of paths is trivially the start node s if the set of paths are edge disjoint in G.

We also define a map $pid : I \rightarrow \mathbb{Z}$ to track *partial identifiers* allocated to paths during the execution of an edge assignment algorithm. Assuming that all edges are initialized with a weight $\perp$ (this denotes the undefined value), the partial identifier of a path is defined as follows.

$$pid(p) = \sum_{e \in edges(p) \wedge W(e) \neq \perp} W(e)$$

From the statement of Problem A, it is evident that an edge assignment must simultaneously satisfy two constraints, uniqueness and compactness of path identifiers. We now describe how PPP satisfies these constraints.

Uniqueness. We first define an invariant that *any* algorithm computing an edge assignment by processing nodes in reverse topological order must satisfy, in order to ensure that all interesting paths are allocated unique identifiers[1].

LEMMA 4 (Invariant for uniqueness). *Consider a node v being processed by the algorithm. If v is the critical node for any pair of interesting paths p and p, then p and p' should be assigned different partial identifiers after the node v has been processed.*

Proof: Follows from the definition of critical nodes, and the fact that the algorithm assigns weights to edges in reverse topological order. ■

[1] A similar invariant based on suffixes can be defined if the algorithm were to perform a top-down traversal, processing nodes in topological order.

The PPP algorithm computes an edge assignment that attempts to achieve the most compact path numbering (low δ) that maintains this invariant at every node. However, to make the algorithm simpler and more amenable for analysis (Section 4), PPP makes the following approximation. Instead of explicitly checking the partial identifiers of paths at a critical node, PPP works over *intervals* of path identifiers. Given an edge e, the interval $int_{e,q}$ represents the range of partial identifiers allocated to all interesting paths through e that have a prefix q. Formally,

$$int_{e,q} = [\min_{p \in paths_I(e) \wedge prefix(p,e)=q} pid(p), \max_{p \in paths_I(e) \wedge prefix(p,e)=q} pid(p)]$$

At every node, PPP computes an interval $int_{e,q}$ for every (edge, prefix) pair, and assigns weights to the edges to maintain the following invariant.

LEMMA 5 (Invariant for uniqueness over intervals). *Consider a node v being processed by* PPP. *Assume that a prefix q induces a set of intervals $S_{v,q} = \{int_{e,q} \mid e \in out(v)\}$ on the outgoing edges of v. To ensure uniqueness, the intervals in $S_{v,q}$ should not overlap after v has been processed. Furthermore, this condition must hold for every prefix $q \in \bigcup_{e \in out(v)} prefix_I(e)$.*

Proof: It is easy to see that if a prefix q induces an interval at two or more outgoing edges of a node v, then v is the critical node for all paths p with the prefix q. By preventing overlap between all such intervals for a given prefix, PPP automatically ends up separating all paths with prefix q for which node v is critical. If this condition is satisfied for all prefixes, all interesting paths for which node v is critical are distinguished, and hence the Lemma 4 holds. ■

Compactness. We will now describe how PPP ensures compactness. At any node $v \in V$, consider the set of intervals $S_{v,q} = \{[min_i, max_i]_{i \in [1,|out(v)|]}\}$ induced on edges $e_1, e_2 \ldots e_{|out(v)|}$ by a prefix q. If q is the only valid prefix at v, PPP uses *compaction* to compute a *minimal* edge assignment $W(e_i)_{i \in [1,|out(v)|]}$ which ensures that these intervals do not overlap. To achieve compaction, each $W(e_i)$ is computed as follows:

$$W(e_i) = \sum_{j \in [1,(i-1)]} (max_j - min_j + 1) - min_i \quad (1)$$

$$= cis_{i-1} - min_i \quad (2)$$

where cis_{i-1} represents the cumulative interval size of all intervals induced on previous edges. However, this simple compaction method cannot be used if multiple prefixes induce intervals on the outgoing edges of a node. In such situations, PPP performs a *join* operation over the intervals at all edges with two or more intervals. The *join* operation computes the weights induced by different prefixes on the edge, and conservatively assigns a weight equal to the maximum among all these weights. Due to the *join*, interesting paths associated with all but one of the prefixes will be assigned a weight higher than what is required to separate its intervals, creating holes in the path numbering. However, it is easy to see that this choice of weight leads to the most compact numbering that is feasible.

Figure 8 illustrates the scenarios that PPP deals with. In Figure 8(a), all interesting paths through the node a have the same prefix q_1 and traverse the edge e_2 (represented by the shaded region). Since the interval int_{e_2,q_1} is the only interval in the set S_{a,q_1}, no overlap between intervals exist and the invariant for uniqueness (Lemma 5) is trivially satisfied. A similar situation occurs in Figure 8(b), where paths through the edges do not share any prefixes. The interval sets S_{a,q_1} and S_{a,q_2} are singletons and no conflicts occur.

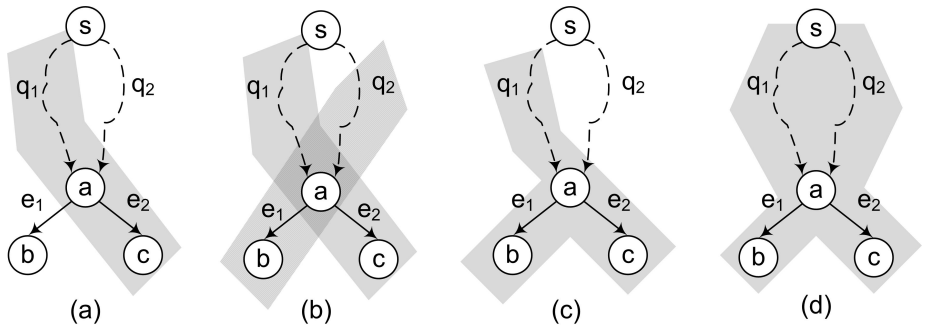

Figure 8. The assignment of weights to edges under four scenarios

Figure 8(c) represents a scenario where the interesting paths induce two intervals for the prefix q_1. PPP uses Equation 1 to compute weights and ensures that these intervals do not overlap. Finally, Figure 8(d) illustrates the scenario where the interesting paths through edges e_1 and e_2 share prefixes q_1 and q_2. Figure 9 illustrates the effect of a join on two sets of intervals induced by prefixes q_1 and q_2. Since we need a larger weight (say w computed by Equation 1) to separate intervals induced by prefix q_1 (as compared to the weight w' required to separate intervals induced by prefix q_2), PPP assigns w to e_2, leading to a hole in the interval for the prefix q_2.

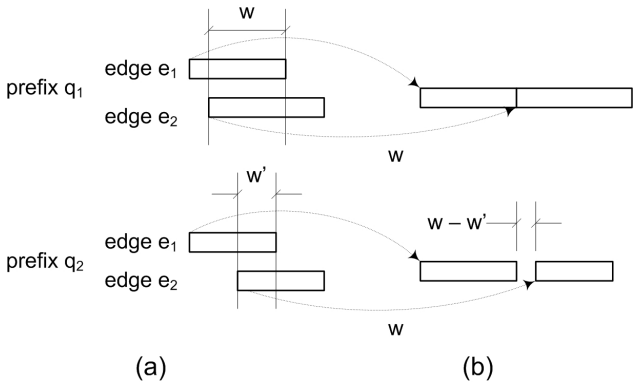

Figure 9. The effect of using the *join* operator to conservatively assign weights to edges. (a) Intervals induced by the prefixes before the join, and (b) effective intervals after the join.

Figure 10 describes the PPP algorithm in detail. For each node $v \in V$ and each outgoing edge $e \in out(v)$, the algorithm iterates over all prefixes and computes the beginning of the interval $int_{e,q}$ ($min_{e,q}$ at Line 5). It uses an auxiliary map cis to determine the cumulative interval size of intervals through previously processed outgoing edges of v with the prefix q (as per Equation 1) and computes the weight induced by q on the edge (Line 7). Finally, the join operation (Lines 10 and 11) selects the maximum over the weights induced by each prefix and assigns this weight to the edge. After the edge is assigned a weight, PPP updates the partial identifiers of all paths through the edge and also computes the new $cis(q)$ for the next iteration on this edge.

In summary, at every node $v \in V$, *computePPPIncrements* recursively merges intervals of each prefix q into the most compact single interval $int_{v,q}$. At the start node s, this interval defines the range of identifiers allocated to the interesting paths. The time complexity of PPP is $\mathcal{O}(|E| \times |I|)$, where E is the set of edges in G, and I is the set of interesting paths.

procedure *ComputePPPIncrements* (G)
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) a map $paths_I : E \rightarrow 2^P$.
(c) $pid : P \rightarrow \mathbb{Z}^{\geq 0}$ initialized to 0 for all interesting paths.
(d) $cis : prefix \rightarrow \mathbb{Z}^{\geq 0}$, initialized to 0 for all prefixes of interesting paths.
Returns: A edge assignment $W : E \rightarrow \mathbb{Z}$.

```
1:  for all nodes v ∈ V in reverse topological order do
2:    for all edges e ∈ out(v) s.t. e ∈ edges(p) for some p ∈ I
      do
3:      for all prefix q ∈ prefix(e) do
4:        // compute the beginning of the interval int_{e,q}
5:        min_{e,q} := min_{p∈paths_I(e)∧prefix(p,e)=q} pid(p);
6:        // compute weight induced by prefix q
7:        weight_q := cis(q) − min_{e,q};
8:        // the join: compute the maximum weight
9:        if ((W(e) = ⊥) ∨ (W(e) < weight_q)) then
10:         W(e) := weight_q;
11:       end if
12:     end for
13:     // update partial identifers of all paths through e
14:     for all paths p ∈ paths_I(e) do
15:       pid(p) := pid(p) + W(e);
16:     end for
17:     for all prefixes q ∈ prefix(e) do
18:       // determine new cumulative interval size for prefix q
19:       cis(q) := max_{p∈paths_I(e)∧prefix(p,e)=q} pid(p) + 1;
20:     end for
21:   end for
22: end for
```

Figure 10. The preferential path profiling (PPP) algorithm for computing an edge assignment for a set of interesting paths.

3.2 Example

We now walk-through an example illustrating how the PPP algorithm works. Let us assume that we are interested in profiling the paths $sacdt$, $sact$ and $sbct$ in the DAG from Figure 3. The following steps trace the manner in which PPP assigns weights to edges of the DAG. Step n_j denotes that at step i, PPP processes node n.

Step 1_t Initialize the partial identifiers of all paths to 0 and cumulative interval sizes of all prefixes to 0.

Step 2_d Node d is not a critical node for any pair of paths since it has only one outgoing edge. The prefix $sacd$ induces an interval $[0, 0]$ on the edge (d, t). Since $cis(sacd) = 0$, $W((d, t)) = 0 - 0 = 0$.

Step 3_c Node c is a critical node for paths $sacdt$ and $sact$. Say the edge (c, d) is processed first. Both prefixes sac and sbc induce an interval $[0, 0]$ on this edge. Hence PPP assigns a weight $W((c, d)) = 0 - 0 = 0$ to this edge. PPP updates the map cis as follows $\rightarrow cis(sac) = 1$ and $cis(sbc) = 1$. Next PPP processes the edge (c, t). The prefix sbc induces an interval $[0, 0]$ on this edge. Since $cis(sbc) = 1$, PPP assigns a weight $W((c, t)) = 1 - 0 = 1$ The partial identifiers of paths $sact$ and $sbct$ are also updated to 1.

Step 4_b Node b has one outgoing edge (b, c). The prefix sb induces an interval $[1, 1]$ on this edge. PPP assigns a weight $W((b, c)) = 0 - 1 = -1$ to this edge since $cis(sb) = 0$. The partial identifier of the path $sbct$ is now updated to $1 + -1 = 0$.

Step 5_a Node a has two outgoing edges, but only the edge (a, c) has interesting paths through it. The prefix sa induces an interval $[0, 1]$ at this edge. PPP assigns a weight $W((a, c)) = 0 - 0 = 0$ to the edge.

Step 6_s Node s has two outgoing edges with three paths, all sharing a common prefix s. This prefix induces an interval [0,1] at the edge (s, a), and the interval $[0, 0]$ at the edge (s, b). PPP processes the edge (s, a) first and assigns a weight $W((s, a)) = 0 - 0 = 0$ to the edge. Then PPP updates $cis(s) = 1 + 1 = 2$, and processes the edge (s, b). Since $cis(s) = 2$, the edge (s, b) is assigned a weight $W((s, b)) = 2 - 0 = 2$. The partial identifier of the path $sbct$ is also updated to 2.

On termination, PPP assigns the identifiers 0, 1 and 2 to the interesting paths $sacdt$, $sact$ and $sbct$ respectively.

3.3 Discussion

In summary, PPP attempts to achieve a compact path numbering by (1) only numbering the edges required to distinguish interesting paths, and (2) computing the smallest weights such that the interesting paths are assigned unique identifiers. Our experiments suggest that PPP achieves good compactness measures for a vast majority of the procedures, even when a large number of interesting paths are specified. However, despite its best efforts, PPP does not always achieve the best possible compactness measure. Figure 11 illustrates one scenario in which PPP fails to assign an optimal numbering although such a numbering clearly exists. Consider the graph G_1 and assume that PPP has assigned identifiers to a set of interesting paths. Also assume that the interval of allocated identifiers contains two holes of size k_1 and k_2 respectively. As shown in the figure, we can construct a new graph G and select a set of interesting paths such that there exists an edge assignment which achieves the $\delta = 1$. Here G is obtained via a parallel composition of three graphs G_1, G_2 and G_3 such that $|paths(G_2)| = k_1$ and $|paths(G_3)| = k_2$. Furthermore, the set of interesting paths now includes all paths through G_1 and G_2. An optimal numbering for this set of interesting paths is obtained by assigning a weight I_1 to the edge (s, s_2) and I_2 to the edge (s, s_3), filling up the holes in the interval of graph G_1. However, PPP fails to compute such assignments since we restrict ourselves to the class of solutions where edge intervals do not overlap. As we show in the next section, this constraint allows us to establish an optimality condition by drawing a connection with arithmetic coding.

4. Path Profiling - an information theoretic perspective

In this section, we give an information theoretic characterization of the preferential path profiling algorithm. We begin by introducing the notion of arithmetic coding and context modeling. We then show that the Ball-Larus path profiling algorithm is a special instance of arithmetic coding. After drawing this connection, we reformulate Problem A described in Section 3 so that it is more amenable to analysis, and provide a theoretical analysis of our algorithm for preferential path profiling.

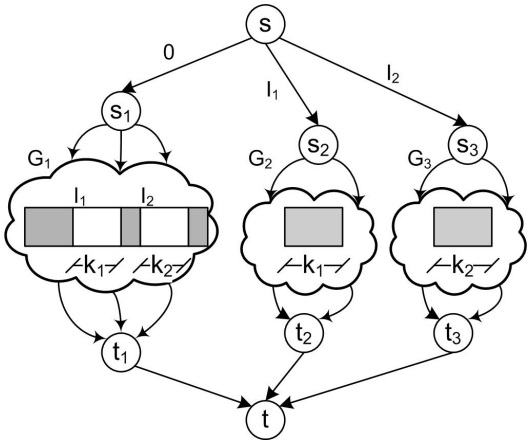

Figure 11. Figure illustrating a scenario in which PPP assigns a sub-optimal numbering but an optimal numbering clearly exists.

4.1 Arithmetic Coding

Arithmetic coding [13, 15, 7] is a well-known universal, lossless compression technique that achieves close-to-optimal compression rates. Much like other compression schemes, arithmetic coding relies on the observation that in any given input stream, a small fraction of characters/substrings are likely to occur frequently. Arithmetic coding achieves compression by encoding these frequently occurring characters/substrings using a smaller number of bits. An arithmetic coder uses a *probability model* to identify frequent characters. In the simplest of cases, the probability model D is an assignment of probabilities to characters of the input alphabet Σ, and is easily obtained from the frequency counts of characters in a representative string.

An arithmetic coder encodes strings into a single positive number less than 1. To compute this number, the arithmetic coder maintains a *range* or an *interval*, which is set of $[0, 1)$ at the beginning of the coding process. As each symbol of the input string is processed, the coder iteratively narrows the range based on the probability of the symbol. After the last symbol is read, any number within the resulting range uniquely represents the input string. Moreover, this number can be uniquely decoded to create the exact stream of symbols that went into its construction. We illustrate the encoding an decoding process by way of an example (adapted from [15]).

Symbol	Probability	Range
a	0.2	$[0, 0.2)$
b	0.3	$[0.2, 0.5)$
c	0.1	$[0.5, 0.6)$
d	0.2	$[0.6, 0.8)$
e	0.1	$[0.8, 0.9)$
!	0.1	$[0.9, 1)$

Table 1. A sample probability model for the alphabet $\Sigma = \{a, b, c, d, e, !\}$.

Example. Let $\Sigma = \{a, b, c, d, e, !\}$ be the finite alphabet, and let a fixed model that assigns probabilities to symbols from Σ be as

shown in Table 1. Suppose we wish to send the message $bacc!$. Initially, both the encoder and the decoder know that the range is $[0, 1)$. After seeing the first symbol b the encoder narrows it down to $[0.2, 0.5)$ (this is the range that the model allocates to symbol b). For the second symbol a, the interval is further narrowed to one-fifth of itself, since a has been allocated $[0, 0.2)$. Thus the new interval is $[0.2, 0.26)$. After seeing the first c the narrowed interval is $[0.23, 0.236)$, and after seeing the second c the new interval is $[0.233, 0.2336)$. Finally on seeing !, the interval is $[0.23354, 0.2336)$; knowing this to be the final range, the decoder can immediately deduce that the first character was b. Now the decoder simulates the action of the encoder, since the decoder knows that the interval $[0.2, 0.5)$ belonged to b, the range is expanded to $[0.2, 0.26)$. Continuing this way, the decoder can completely decode the transmitted message. It is not really necessary for the decoder to know both ends of the range produced by the encoder. Instead, a single number in the range (say 0.23355 in our example) will suffice.

Note that *the ranges for any probability model (for example, the one in Table 1) are non-intersecting; this condition is critical for arithmetic coding to work. If the ranges associated with the input symbols were intersecting, two or more strings could map to the same interval and the decoder has no way of distinguishing these strings*.

From the discussion, it should be clear that for arithmetic coding to be effective, the frequency of occurrence of characters in the input string must be skewed and the skew must be accurately reflected in the probability model. In other words, better compression rates are achieved if the model makes accurate predictions about the nature of the input string. We now describe a technique known as finite context modeling, which is commonly used to obtain more accurate probability models.

Finite context modeling. In a finite context scheme, the probabilities of each symbol are calculated based on the *context* the symbol appears in. In its traditional setting, the context is just the symbols that have been previously encountered. The *order* of the model refers to the number of previous symbols that make up the context. One way of compressing data is to make a single pass over the symbols to be compressed (to gather statistics), and then encode the data in a second pass. The statistics collected are the relative frequencies of occurrences of the respective symbols. These relative frequencies are then used to encode/decode the symbol as explained in earlier. Essentially, the model in this setting consists of a set of tables for every possible context up to size k for any order k model. Each context is a state, and each entry corresponding to a symbol frequency is indicative of its probability of occurrence in that context. An *optimal* model is one that represents the best possible statistics for the actual data that is to be compressed. Unfortunately, computing an optimal model in general is undecidable [9].

It can be shown that arithmetic coding achieves optimal compression[2] for a given probability model D [7]. Specifically, if X is a random variable representing events over a set $\mathcal{X}$ with a probability distribution D, then the average number of bits required to encode any event from $\mathcal{X}$ using arithmetic coding is equal to the *entropy* [7] of D which is defined as follows.

$$\mathcal{H}(D) \stackrel{\text{def}}{=} -\sum_{x \in \mathcal{X}} \mathsf{P}(X = x) \log_2 |\mathsf{P}(X = x)|$$

Note: $\mathcal{H}(D)$ is the *binary entropy function*, and $\mathsf{P}(X = x)$ is the probability of occurrence of the event $X = x$.

[2] In order to achieve optimal overall compression of data, the model must be an optimal model for that data.

procedure *computeBLModel* (G)
Assume:
(a) $G = (V, E, s, t)$ is a DAG.
(b) $\forall v \in V$, a map $D : E \to [0, 1]$ that is initially undefined.
Returns: a model D such that $\forall v \in V, \sum_{e \in out(v)} D(e) = 1$.

```
1: N_t := 1;
2: for all nodes v ∈ V in reverse topological order do
3:     N_v := Σ_{e∈out(v)} N_{dest(e)};
4:     for all edges e ∈ out(v) do
5:         D(e) := N_{dest(e)} / N_v;
6:     end for
7: end for
```

Figure 12. The Ball-Larus algorithm as a model computation process.

Example. For the model D described in Table 1, the entropy $\mathcal{H}(D) = -\mathsf{P}(a) \log_2 |\mathsf{P}(a)| - \mathsf{P}(b) \log_2 |\mathsf{P}(b)| - \mathsf{P}(c) \log_2 |\mathsf{P}(c)| - \mathsf{P}(d) \log_2 |\mathsf{P}(d)| - \mathsf{P}(e) \log_2 |\mathsf{P}(e)| - \mathsf{P}(!) \log_2 |\mathsf{P}(!)| = -(0.2 \log_2 |0.2| + 0.3 \log_2 |0.3| + 0.1 \log_2 |0.1| + 0.2 \log_2 |0.2| + 0.1 \log_2 |0.1| + 0.1 \log_2 |0.1|) = 2.45$ bits.

4.2 The Ball-Larus algorithm and Arithmetic coding

We will now show that the Ball-Larus profiling algorithm is in fact an instance of arithmetic coding for paths in a DAG $G = (V, E, s, t)$. We first observe that both path numbering and arithmetic coding have similar objectives, i.e., to *compactly and uniquely* encode strings from an input alphabet. In path numbering, the input alphabet is the set of edges through a DAG, and the input strings are paths through the DAG. We also find that the process of assigning weights to the edges of the DAG corresponds to the process of computing a probability model. However, unlike arithmetic coding where computing an optimal model is undecidable in general, an optimal model for paths through a DAG can in fact be computed for the following reasons: (a) the set of strings that can occur is known *a priori*; this is precisely the set of all paths through the DAG, and (b) the order in which edges can occur in paths is determined by the structure of the graph, which is also known *a priori*. Based on these observations, we derive a model computation procedure for paths through a DAG that is equivalent to the Ball-Larus algorithm. The procedure *computeBLModel*, shown in Figure 12, takes a DAG G as an input and assigns a probability $D(e)$ to every edge e in the graph. The resulting model is a finite context model, where the context of an edge is its source node, and the probability assigned to an edge is the probability of the edge being traversed given that the source node has been reached. One can easily verify that $\forall v \in V$,

$$\sum_{e \in out(v)} D(e) = 1$$

For every path $p \in paths(G)$, the probability $\mathsf{P}(p)$ induced by the model D is defined as follows.

$$\mathsf{P}(p) \stackrel{\text{def}}{=} \prod_{e \in edges(p)} \mathsf{P}(e)$$

It also follows by a simple counting argument that for every $p \in paths(G)$, $\mathsf{P}(p) = 1/|paths(G)|$ and $\sum_{p \in paths(G)} \mathsf{P}(p) = 1$. Denote by D_G, the probability distribution over the set $paths(G)$ – it follows immediately that the entropy $\mathcal{H}(D_G)$ is equal to $\log_2 |N|$.

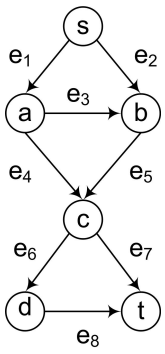

Figure 13. Example for the procedure *computeBLModel*.

Symbol	Probability	Range
e_1	$2/3$	$[0, 2/3)$
e_2	$1/3$	$[2/3, 1)$
e_3	$1/2$	$[0, 1/2)$
e_4	$1/2$	$[1/2, 1)$
e_5	1	$[0, 1)$
e_6	$1/2$	$[0, 1/2)$
e_7	$1/2$	$[1/2, 1)$
e_8	1	$[0, 1)$

Table 2. The Ball-Larus probability model D_G for the graph G in Figure 13 computed by *computeBLModel*.

procedure *pathEncoder* (p, D, N)
Assume:
$G = (V, E, s, t)$ is a DAG and $p = (v_1, \ldots, v_k) \in paths(G)$, $\{v_i \in V\}_{1 \le i \le k}$.
Returns: path identifier for p.

```
1: in := [0, N);
2: for all i = 1 to k − 1 do
3:     e := (v_i, v_{i+1});
4:     [x, y) := in;
5:     n := y − x;
6:     let [r, r') be the range for e defined by D;
7:     in := [x + ⌊rn⌋, x + ⌈r'n⌉);
8: end for
9: [x, _) := in;
10: return x;
```

Figure 14. The coding algorithm that takes a model D and path $p \in paths(G)$ as input, and returns the path identifier or the encoding for p.

Example. Consider the graph G shown in Figure 13. The model D computed by the procedure *computeBLModel* is given in Table 2.

We will now describe the procedure *pathEncoder* that takes a path $p \in paths(G)$, the model D computed by *computeBLModel* and $N = |paths(G)|$ as input, and computes its Ball-Larus identifier. This is the analogous to the procedure *computePathIdentifier* in Section 2.2.

Since arithmetic coding is optimal [7], that is, it achieves the entropy of the input model, *pathEncoder* (which is an arithmetic coder) is also optimal. We will make this connection explicit in the following example.

Example. Consider the graph G shown in Figure 13. Let D be the model computed by the procedure *computeBLModel* as given in Table 2. For an input path $sbcdt$, *pathEncoder*($sbcdt$, D, N) works as follows. We have $N = 6$, and the algorithm starts by assigning $in := [0, 6)$. The first edge encountered along this path is e_2, and therefore the interval in is set to $[4, 6)$. After seeing the next edge e_5, *pathEncoder* chooses the same interval $in = [4, 6)$. For the next edge e_6, the interval is narrowed down to $in = [4, 5)$, and finally for the last edge e_8, the interval is set to $in = [4, 5)$. Therefore, *pathEncoder* returns 4 as the path identifier for the path $sbcdt$. Note that this is precisely the Ball-Larus identifier for this path as is evident from Figure 3.

4.3 The PPP algorithm and arithmetic coding

In Section 3.1, we described an algorithm that compactly numbers a subset of interesting paths $I \subseteq paths(G)$ in a DAG $G = (V, E, s, t)$. We now show that the PPP algorithm is equivalent to an arithmetic coding scheme that uses a *maximal* context model for encoding paths in G. As described in Section 3.1, the procedure *computePPPIncrements* computes for every pair $(e, q) \in E \times prefix(p, e)$, an interval $int_{e,q}$ that represents the range of partial identifiers of interesting paths through e. At every node $v \in V$, these intervals are used to compute the weights associated with edges emanating from v. It can be shown that this procedure is equivalent to computing a finite context model with prefixes as the context. Consider the simple case of a node v with two outgoing edges e_1 and e_2. Assume that a single prefix q induces intervals $int_{e_1,q}$ and $int_{e_2,q}$ on the edges e_1 and e_2 respectively. Define $cis_{v,q} = int_{e_1,q} + int_{e_2,q}$, and $p = \frac{int_{e_1,q}}{cis_{v,q}}$. Then the model $D_{v,q}$ at node v is defined as follows.

Symbol	Probability	Range
e_1	p	$[0, p)$
e_2	$1 - p$	$[p, 1)$

Computing the model is more involved when multiple prefixes induce intervals on the outgoing edges of a node. The problem arises because each outgoing edge may be associated with multiple probabilities, one for each valid prefix at node v. However, unlike traditional context models, an edge in the DAG cannot be associated with multiple probabilities. We overcome this problem by using the *join* operator (defined in Section 3.1) to compute a conservative approximation of the individual models (which we refer to as D_v). Due to the *join*, certain edges may be assigned smaller probabilites than required. Consequently, the number of bits required to encode interesting paths through those edges may increase. The final model D_G is a combination of all models $D_v, v \in V$.

Finally, the process of computing the PPP identifier for an interesting path $p \in I$ corresponds to calling the procedure pathEncoder with parameters p, D_G and $N = |int_{s,s}|$ (assigned to the start node $s \in V$).

Example. Consider the graph G shown in Figure 13. Let the set of interesting paths be $I = \{sacdt, sact, sbct\}$. Then the probability model D_G computed by *computePPPIncrements* is shown in Table 3. For the input path $sact$, *pathEncoder*($sact$, D_G, N) works as follows. We have $N = 3$, and the algorithm starts by assigning $in = [0, 3)$. The first edge encountered along this path is e_1 (model = $D_{s,s}$), and therefore the interval in is set to $[0, 2)$. After seeing the next edge e_4 (model = $D_{a,sa}$), *pathEncoder* chooses the same interval $in = [0, 2)$. For the next edge e_7 (model = $D_{c,sac}$), the interval is narrowed down to $in = [1, 2)$, and therefore *pathEncoder* returns 1 as the path identifier for the path $sact$. Note that

Model	Symbol	Probability	Range
$D_{s,s}$	e_1	$2/3$	$[0, 2/3)$
	e_2	$1/3$	$[2/3, 1)$
$D_{a,sa}$	e_3	0	$empty$
	e_4	1	$[0, 1)$
$D_{b,sb}$	e_5	1	$[0, 1)$
$D_{c,sac}$	e_6	$1/3$	$[0, 1/3)$
	e_7	$2/3$	$[1/3, 1)$
$D_{c,sbc}$	e_6	$1/3$	$[0, 1/3)$
	e_7	$2/3$	$[1/3, 1)$
$D_{d,sacd}$	e_8	1	$[0, 1)$

Table 3. The PPP probability model D_G for the graph G in Figure 13 computed by *computePPPIncrements*.

this is precisely the PPP identifier for this path as is evident from Figure 3.

This characterization of PPP as a model computer and encoder of paths works due to the fundamental invariant that the intervals in PPP do not overlap (this follows from Lemma 5).

4.4 Analysis of the PPP algorithm

The Ball-Larus algorithm computes an edge weight assignment such that δ (the objective function for Problem A) is equal to 1. Therefore, it is an optimal algorithm for those problem A instances for which the interesting paths are all paths, that is, $I = paths(G)$. Information theoretically, this corresponds to saying that all paths in the graph are equally likely (and there is no bias towards any set of paths) – from the previous section, the entropy for such a distribution (say D) is equal to $\log_2 |paths(G)| \Rightarrow paths(G) = 2^{\mathcal{H}(D)}$.

In the previous section, we also saw that the procedure *computePPPIncrements* computes a probability model for a DAG G and a set of interesting paths $I \subseteq paths(G)$. Intuitively, this corresponds to computing a probability distribution D that is biased towards the interesting paths over the uninteresting ones. Since PPP essentially mimics an arithmetic coder, the total number of bits required to represent the set of paths distributed according to the model D is equal to the entropy $\mathcal{H}(D) \Rightarrow$ the interval size that PPP computes is equal to $\lceil 2^{\mathcal{H}(D)} \rceil$. Therefore, the compactness that PPP achieves is $\frac{2^{\mathcal{H}(D)}}{|I|}$, and this is parameterized over how "precise" the model D is. In Theorem 1, we show that this model D computed by *computePPPIncrements* is indeed optimal. We now state a variant of Problem A and prove that PPP computes the optimal solution to this problem.

Problem B (Optimal Edge Assignment): Given a DAG $G = (V, E, s, t)$ and a set of interesting paths $I \subseteq paths(G)$, compute an edge assignment $W : E \rightarrow \mathbb{Z}$ that satisfies the following conditions.

1. uniqueness: $\forall p, q \in I, pathid(p) \neq pathid(q)$,
2. compactness: The compactness measure γ defined by

$$\gamma \stackrel{\text{def}}{=} \frac{2^{\mathcal{H}(D)}}{|I|}$$

is minimized, where D is any probability distribution on $paths(G)$ induced by a model D_G.

We now state and prove the main result in our analysis.

THEOREM 1. *Given a* DAG $G = (V, E, s, t)$ *and a set* $I \subseteq paths(G)$*, the procedure computePPPIncrements computes the optimal solution to* Problem B.

Proof: It follows from Section 4.3 that the PPP algorithm computes a *maximal* context model D_G for a given set of interesting paths in G. The model is a precise context model because it uses the largest context possible, which is the entire prefix. Since the model D_G is optimal, $\gamma = \frac{2^{\mathcal{H}(D_{paths(G)})}}{|I|}$ (where $D_{paths(G)}$ is the probability model over $paths(G)$ induced by D_G) is also optimal (this follows from the optimality of the *pathEncoder* procedure, which essentially mimics an arithmetic coder), and the theorem follows. ■

From Section 3.3, it is clear that any algorithm (such as PPP) that maintains the invariant stated in Lemma 5 will not be able compute an optimal δ. On the other hand, our experiments described in Section 5 also indicate that the objective function γ minimized by our algorithm is close to the minimal δ (the objective function for Problem A) for most graphs and their associated interesting paths, and the interval size is small enough for the path profiler to use an array to track interesting paths.

5. Experimental evaluation

We have implemented the preferential path profiling algorithm using the Scale compiler infrastructure [10]. A few key features of our implementation are listed below.

- *Representing paths and prefixes.* While a user is free to specify the set of interesting paths in several ways, we choose to represent the interesting paths using their Ball-Larus identifiers. Similarly, we represent a prefix using the cumulative sum of the Ball-Larus weights along the edges of the prefix. It is easy to see that this sum is unique for each prefix leading to a given node.
- *Register usage.* Unlike traditional path profiling, preferential path profiling requires two registers, one for PPP counts and one for Ball-Larus counts. Our experiments suggests that the use of two registers instead of one does not add to the overheads of profiling.
- *Counter optimizations.* All counter placement optimizations [3] used in the Ball-Larus algorithm also apply to the PPP counter. These include reducing the number of initialization and increment operations by placing weights only on the edges that do not belong to a maximal spanning tree of the DAG, pushing counter initialization downwards along the edges of the DAG and merging the initializations with the first increments. In our implementation, we ensure that both PPP and Ball-Larus counter updates occur on the same edges.
- *Hash table usage policy.* The default Ball-Larus profiler is configured to use a hash table instead of an array when the total number of paths through the procedure exceeds a threshold. However, the policy for hash table use in preferential path profiling depends on the specific scenario in which the profiler is used. For instance, in residual path profiling, where the goal is to detect the occurrence of untested (uninteresting) paths, a hash table may never be used, even when the PPP identifers allocated to the tested (interesting) paths are large. Here, the profiler implementation may decide to ignore all tested paths with PPP identifiers greater than a threshold, in essence treating them as untested paths. As a result, a few of the tested paths may appear as untested during program execution. Such false positives may be acceptable since PPP ensures that interesting paths are assigned compact numbers. In addition, they can be

easily weeded out off-line. A policy that switches to using hash tables when the PPP identifiers are large may also be used in other applications. Although our implementation supports both modes, we report our results using the former policy.

- *Additional checks.* Before indexing the path array using PPP identifiers, our profiler must check for an underflow/overflow, which can result when an uninteresting path occurs. Our experiments suggest that these additional checks do not add to the cost of preferential profiling since they are highly biased and easy to predict.

We evaluated our profiler implementation using benchmarks from the SPEC CPU2000 suite. We simulated a realistic residual profiling scenario. We first collected a path profile of the benchmarks using the Ball-Larus profiler for the standard reference input. We then assumed that all paths exercised during the reference run were interesting (including procedures where several hundred paths were exercised). These were fed to the preferential profiler, which generated a new instrumented binary. All binaries were run to completion using the reference input on an Alpha 21264 processor running Digital OSF 4.0. Each binary was executed 5 times and the minimum of the execution times (measured using hardware cycle counts) was used for comparison.

Figure 15 shows the percentage overheads of the two schemes relative to execution time of the un-instrumented binary. We find that Ball-Larus profiling incurs an average overhead of 50% with a maximum of 132%. On the other hand, the preferential path profiler incurs an average overhead of 15%, with a maximum of 26%. We attribute the low profiling overheads of PPP to (a) elimination of expensive hash operations, and (b) judicious allocation of counters for profiling (the size of the counter array is proportional to the number of interesting paths and not the number of potential paths).

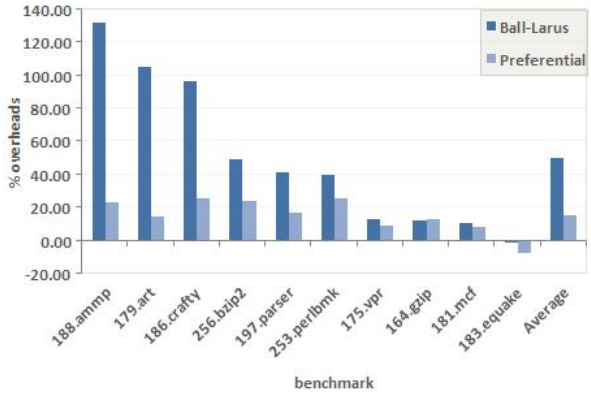

Figure 15. Overheads of preferential path profiling.

Although reflected in the overheads, the real efficacy of the preferential profiling algorithm lies in the compactness measure δ that it achieves. We illustrate the compactness measure achieved by our algorithm in Figure 16, which plots the size of the interval allocated to interesting paths vs. the number of interesting paths for procedures from programs in the SPEC CPU2000 benchmark suite. As aforementioned, all paths exercised during one reference run were selected as interesting paths. The figure suggests that our profiling scheme achieves a δ close to 1 for a vast majority of the procedures, although the value tends to increase for procedures with a large number of paths (100-300). We also found a very small number of cases with $\delta > 10$ (not shown in this figure), most of them in the benchmark *crafty*, a chess program known to have complex control flow.

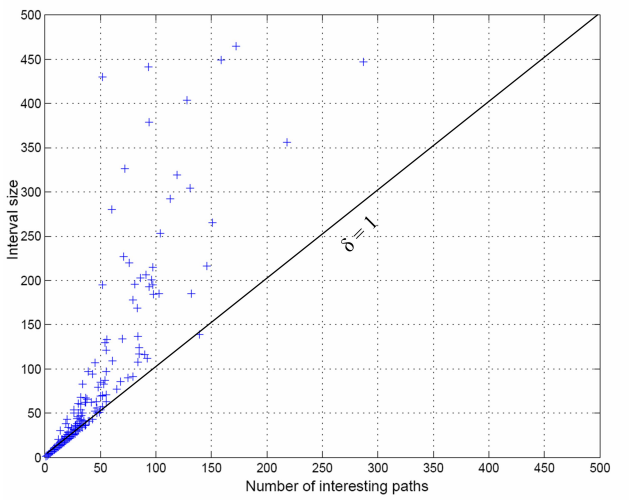

Figure 16. δ values achieved by preferential path profiling.

6. Related work

Several researchers have proposed a variety of techniques to reduce the overhead of Ball-Larus style path profiling [2, 8, 6]. Selective path profiling uses a variation of Ball-Larus numbering where edges are visited in a specific order to ensure that interesting paths are assigned a unique number that is higher than the non-unique numbers assigned to other paths, while minimizing the number of counter updates needed to compute the path number. However, they found that once the number of interesting paths was five or larger, their edges covered most of the DAG and their technique offered little advantage over Ball-Larus numbering. In addition, they made no attempt to ensure that the interesting paths are compactly numbered. Instead of minimizing the number of counter updates needed to compute a path number, we optimize the compactness of numbers assigned to interesting paths. This reduces overhead by enabling the use of a path array in place of a hash table. Our compact numbering scheme is effective even when the number of interesting paths is large.

Both *targeted path profiling* and *practical path profiling* attempt to efficiently profile hot program paths starting from an edge profile by eliminating unneeded instrumentation. Targeted path profiling eliminates profiling cold paths by excluding cold edges and not instrumenting paths that the edge profile predicts well. It uses Ball-Larus numbering for labelling the remaining paths. Practical path profiling attempts to improve over targeted path profiling using a variety of techniques to eliminate a larger number of paths. It also performs intelligent instrumentation placement to further reduce overhead. To minimize overhead, practical path profiling may need to classify warm edges as cold and consequently could compromise the quality of the path profile. It also uses Ball-Larus numbers to uniquely identify the remaining paths. Our technique is orthogonal to both as it proposes a new dense numbering scheme for interesting paths that minimizes the overhead of profiling these paths. It is also more general as it can be applied to scenarios such as residual path profiling (detecting paths not exercised by a test suite), where the techniques that targeted/practical path profiling use to reduce instrumentation overheads do not apply.

Other work in path profiling has focused on collecting richer path profiles. Interprocedural path profiling extends Ball-Larus profiling beyond intraprocedural paths [11]. Tallam et al. proposed a technique to profile overlapping path fragments from which interprocedural and cyclic paths can be estimated [14]. Both these tech-

niques have considerably higher overhead than the Ball-Larus technique for profiling intraprocedural, acyclic paths and our scheme can potentially help reduce this overhead.

7. Conclusion

This paper presents preferential path profiling, a new technique that profiles a specified subset of all program paths with very low overhead. Preferential path profiling labels the paths of interest compactly using a novel numbering scheme. By drawing parallels between arithmetic coding and path numbering we establish an optimality result for our compact path numbering scheme. This compact path numbering allows our implementation to use array-based counters instead of hash table-based counters for gathering path profiles and significantly reduces execution time overhead.

Acknowledgments

We thank Sriram Rajamani, Stefan Schwoon and Aditya Thakur for helpful comments on this work. Special thanks are due to Stefan for proving Lemma 3.

References

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 72–84, 1998.

[2] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Workshop. on Program Analysis for Software Tools and Engineering (PASTE)*, pages 35–42, 2002.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1996.

[4] T. Ball and J. R. Larus. Programs follow paths. Technical Report MSR-TR-99-01, Microsoft Research, 1999.

[5] T. Ball, P. Mataga, and S. Sagiv. Edge profiling versus path profiling: The showdown. In *ACM SIGPLAN Symposium on Principles of Programming Languages(POPL)*, pages 134–148, 1998.

[6] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *International Symposium on Code Generation and Optimization (CGO)*, pages 205–216, 2005.

[7] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., N. Y., 1991.

[8] R. Joshi, M. D. Bond, and C. B. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *International Symposium on Code Generation and Optimization (CGO)*, pages 239–250, 2004.

[9] A. Kolmogorov. Three approaches to the quantitative definition of information. *Prob. Peredach Inform*, 1(1):3–11, 1965.

[10] K. S. McKinley, J. Burrill, M. D. Bond, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z.Wang, and C. Weems. The Scale compiler. *http://ali-www.cs.umass.edu/Scale*, 2005.

[11] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction (CC)*, pages 47–62, 1999.

[12] E. Perelman, T. M. Chilimbi, and B. Calder. Variational path profiling. In *Parallel Architectures and Compilation Techniques '05 (PACT)*, pages 7–16, 2005.

[13] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Res. Develop.*, 23(2):149–162, 1979.

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

[15] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.