P3: Partitioned Path Profiling

Mohammed Afraz
Indian Institute of Science
mohammed.afraz@csa.iisc.ernet.in

Diptikalyan Saha
IBM Research
diptsaha@in.ibm.com

Aditya Kanade
Indian Institute of Science
kanade@csa.iisc.ernet.in

ABSTRACT

Acyclic path profile is an abstraction of dynamic control flow paths of procedures and has been found to be useful in a wide spectrum of activities. Unfortunately, the runtime overhead of obtaining such a profile can be high, limiting its use in practice.

In this paper, we present *partitioned path profiling* (*P3*) which runs K copies of the program *in parallel*, each with the same input but on a separate core, and collects the profile only for a *subset* of intra-procedural paths in each copy, thereby, distributing the overhead of profiling. P3 identifies "profitable" procedures and assigns disjoint subsets of paths of a profitable procedure to different copies for profiling. To obtain exact execution frequencies of a subset of paths, we design a new algorithm, called PSPP. All paths of an unprofitable procedure are assigned to the same copy. P3 uses the classic Ball-Larus algorithm for profiling unprofitable procedures. Further, P3 attempts to evenly distribute the profiling overhead across the copies. To the best of our knowledge, P3 is the first algorithm for parallel path profiling.

We have applied P3 to profile several programs in the SPEC 2006 benchmark. Compared to sequential profiling, P3 substantially reduced the runtime overhead on these programs averaged across all benchmarks. The reduction was 23%, 43% and 56% on average for 2, 4 and 8 cores respectively. P3 also performed better than a coarse-grained approach that treats all procedures as unprofitable and distributes them across available cores. For 2 cores, the profiling overhead of P3 was on average 5% less compared to the coarse-grained approach across these programs. For 4 and 8 cores, it was respectively 18% and 25% less.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics; F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Algorithms, Performance

Keywords

Parallel, Distributed, Path Profiling, Divide and Conquer

1. INTRODUCTION

Collecting execution frequencies of dynamic control flow paths of procedures reveals a wealth of information about the runtime behavior and usage patterns of a program. Acyclic path profile [4] is an abstraction of dynamic control flow paths of procedures and gives execution frequencies of acyclic paths in a procedure. Acyclic path profile has been found to be a useful measure in a wide spectrum of activities ranging from compiler optimizations [10] to testing [17], debugging [8] and maintenance [13].

Unfortunately, the runtime overhead of obtaining such a profile can be high, limiting its use in practice. For example, Vaswani et al. [24] reported an average runtime overhead of 50% with worst case overhead of 132%. Other studies (e.g. [6]) also report similar overheads. We believe that, with the prevalence of multi-core systems and computing clusters, parallelizing acyclic path profiling has become an attractive option to reduce profiling overhead. Surprisingly, to date, there is no algorithm that exploits parallelism for path profiling. In this paper, we present such an algorithm.

We propose to run K *copies* of the program *in parallel*, each with the same input but on a separate core (or cluster node), and collect the profile only for a *subset* of intra-procedural paths in each copy, thereby, distributing the overhead of profiling. A straightforward approach to achieve this is to use the classic Ball-Larus algorithm [4] to instrument only a subset of procedures in each copy, in a way that, every procedure is profiled in exactly one copy. We call this approach *parallel Ball-Larus profiling* (*PBL*) in contrast to *sequential Ball-Larus profiling* (*SBL*) which profiles all the procedures in one copy.

In practice, the number of (acyclic) paths may differ widely across procedures, and consequently, also the profiling overheads. For example, consider a program M with three procedures P, Q and R requiring 100, 10 and 5 instrumentation probes to profile all their paths respectively. An instrumentation *probe* is a statement added by a profiling algorithm to a procedure to track the path-ids and their execution frequencies. The number of probes gives a static estimate of the runtime overhead of profiling. If 3 cores are available, PBL may assign one procedure to each copy (core). The copy profiling the procedure P is likely to be much slower than the others. The benefit of parallelization is limited by the speedup of the *slowest* copy. Thus, PBL may fail to exploit parallelism to the fullest.

We therefore propose a novel approach which attempts to get a more uniform distribution of profiling overhead by sub-dividing the job of profiling all paths of a procedure into sub-jobs of profiling disjoint subsets of paths of the procedure. The subsets are assigned to different static *instances* of a procedure which are then distributed across multiple copies. For example, our approach may obtain three instances, say P_1, P_2 and P_3, of the procedure P above and assign them to separate copies. The subsets of paths of a pro-

cedure are constructed so that they form a partition. Hence, we call our approach *partitioned path profiling* (*P3*). P3 essentially provides more opportunity for load balancing across cores by constructing smaller jobs from bigger jobs.

There are three key challenges that P3 needs to overcome: (1) which procedures to select for partitioning and how to partition their paths, (2) how to instrument an instance of a partitioned procedure so as to obtain exact execution frequencies of the paths profiled in it and (3) how to distribute the instrumented procedures to achieve good load balancing.

We observe that even if the subsets of paths being profiled are disjoint across two instances of a procedure, some instrumentation probes may get *duplicated* between them (see Section 2.2 for an example). We consider profiling of sequential programs. Therefore, once we fix an input, all copies of the program follow the same dynamic control flow path and hence, the duplicated probes along that path get executed in multiple copies. In general, a path which is not profiled in an instance P_i may still go through some probes in P_i. Thus, on one hand, we reduce the number of instrumentation probes per instance. Whereas, on the other hand, we may increase the runtime overhead due to the possibilities mentioned above. P3 therefore only selectively partitions procedures by identifying what we call as *profitable procedures*. The profitable procedures and the partitioning of their paths are identified by a static analysis that uses both intra-procedural and inter-procedural control flow information. This addresses the first challenge.

We now consider the second challenge. An existing approach, selective path profiling (SPP) [1], has been proposed to profile only a *subset* of paths S. We could have used SPP on each instance of a profitable procedure to profile the subset of paths assigned to it. Unfortunately, we noticed that SPP can assign the *same* path-id to a path $p \in S$ and a path $p' \notin S$ (see Section 2.2 for an example). This means that it can over-approximate the execution frequencies of paths, in particular, by counting the execution frequencies of p' as those of p. We therefore design a new algorithm called *precise selective path profiling* (*PSPP*) which overcomes this issue in SPP and use it in P3 for instrumenting the instances of a profitable procedure. The immediate benefit of PSPP is that we can obtain the overall profile of a profitable procedure by merely collating the partial profiles obtained from its instances. For unprofitable procedures, we use the Ball-Larus algorithm. Thus, the exact acyclic path profiles of all procedures can be obtained.

Finally, towards addressing the third challenge, P3 uses the number of instrumentation probes as a cost measure and distributes the instrumented procedures to different copies. The optimal distribution in this setting is an NP-complete problem [14]. P3 therefore uses a round-robin approach that produces a $4/3$th approximation of the optimal distribution [15]. If multiple instances of a profitable procedure are assigned to the same copy, P3 takes the union of the sets of paths being profiled in them and instruments a single instance for all the paths in the union.

Our approach differs from the existing approaches that attempt to lower the profiling overhead. Some approaches attempt to reduce the memory overhead [24, 9], whereas others attempt to reduce the runtime overhead by focusing on a subset of paths that are relevant in specific contexts [1, 24, 5]. In contrast, our goal is to reduce runtime overhead while profiling *all* paths. Our approach is also applicable if only a subset of paths is of interest but we do *not* make this assumption about the usage scenario to speed up profiling.

We have implemented the PSPP and P3 algorithms for sequential C/C++ programs and applied them to profile several programs in the SPEC 2006 benchmark [16]. Compared to SBL, P3 substantially reduced the runtime overhead on these programs averaged

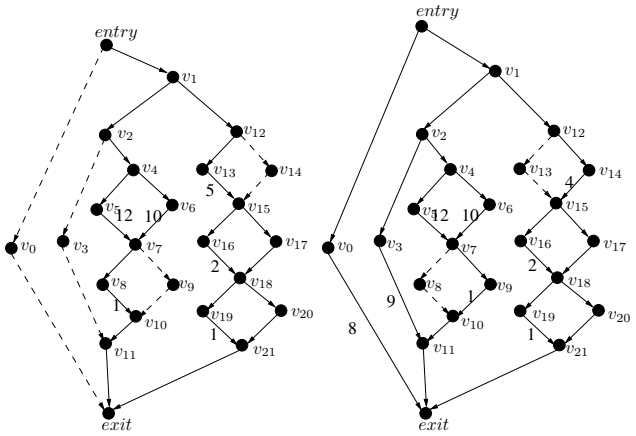


Figure 1: A partition (P_a, P_b) of a procedure P

across multiple inputs. The reduction was 23%, 43% and 56% on average across these programs for 2, 4 and 8 cores respectively. The profiling overhead for an input is taken to be the maximum of the number of times instrumentation probes are executed on the same input across the individual copies. It is essentially the overhead incurred by the slowest copy. P3 also performed better than PBL. We used the round-robin approach of distribution for both P3 and PBL. For 2 cores, the profiling overhead of P3 was on average 5% less compared to PBL across these programs. For 4 and 8 cores, it was respectively 18% and 25% less.

We summarize the main contributions of our work as follows:

- We present P3 – an algorithm for efficient parallelization of path profiling. To the best of our knowledge, this is the first algorithm for parallel path profiling.
- We present PSPP – an algorithm to obtain *exact* execution frequencies of a subset of paths of a procedure – which is used in P3. PSPP on its own can be used in applications such as residual testing [22, 9, 7].
- We have implemented P3 and show its effectiveness compared to the sequential and parallel Ball-Larus profiling on several SPEC 2006 benchmark programs for 2, 4 and 8 cores.

2. OVERVIEW

We first present the definitions used in the paper and some background. We then illustrate the key steps of P3 through examples.

2.1 Definitions and Background

Consider a directed acyclic graph (DAG) G which represents all acyclic intra-procedural paths of a procedure P. We refer the reader to [4] on how such a DAG is constructed from the control flow graph of a procedure. Formally, $G = (V, E, entry, exit)$ where V is a finite set of vertices representing basic blocks of P, the set of edges $E \subseteq V \times V$ approximates the control flow between the respective basic blocks, and $entry$ and $exit$ are respectively the unique entry and exit vertices of G. For example, Figure 1(a) shows a DAG for some procedure. For a vertex $v \in V$, the set of successors is given by $succ(v) = \{w \in V \mid (v, w) \in E\}$.

Given a *path* p in G, $edges(p)$ gives the set of edges in p. For a path p passing through a vertex v, *suff*(p, v) denotes the suffix of p starting with v. We use N_v to denote the number of paths passing through v. A *labeling function* L associates a natural number, called an *edge label*, to each edge in G. The *path-id* of a path p is the sum of edge labels of edges in $edges(p)$ and is denoted by $pathid(p)$. As a convention, zero-valued edge labels are not

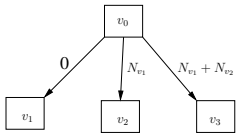


Figure 2: Ball-Larus labeling

shown. The path-id of $p = \langle entry, \ldots, v_4, v_6, v_7, v_9, \ldots, exit\rangle$ is 10. A profiling algorithm assigns edge labels and instruments the edges to compute the path-id at runtime. Thus, each edge with a non-zero label is instrumented with a statement called an instrumentation *probe*. For a procedure P, $overhead(P)$ is the total number of non-zero edges in the DAG of P and is taken as an estimate of the runtime overhead of profiling P. Under the labeling shown in Figure 1(a), the overhead for that procedure is 6.

For a procedure P, if there are multiple instances used for profiling a partition of its paths, we denote the set of paths being profiled in an instance P_i by *interesting*(P_i). We call a labeling function L_i for an instance P_i, a valid labeling, if every path $p \in$ *interesting*(P_i) has a unique path-id (under the labeling L_i) which is distinct from the path-ids of paths which are not in *interesting*(P_i). Formally, a labeling L_i is a *valid labeling* of an instance P_i if:

(a) $\forall p, q \in interesting(P_i) : pathid(p) \neq pathid(q)$

(b) $\forall p \in interesting(P_i), \forall r \notin interesting(P_i) :$
$pathid(p) \neq pathid(r)$

A valid labeling generates the exact execution frequencies of the interesting paths of a procedure from a single copy. This in turn simplifies the job of obtaining exact frequencies of all paths of a procedure spread across multiple copies. We note that a valid labeling may assign the same path-id to two paths not in $interesting(P_i)$.

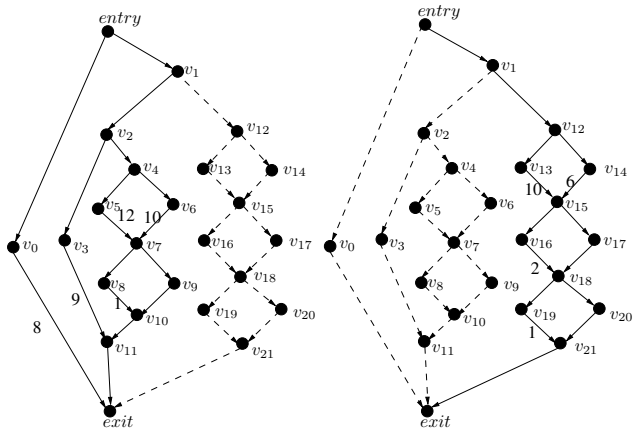
If there is only *one* instance of a procedure P (that is, its set of paths is not sub-divided) then the classic Ball-Larus algorithm [4] already yields a valid labeling L. We give a brief overview of the Ball-Larus algorithm. In the first step, it visits the vertices in the DAG in the reverse topological order and labels a vertex v by the number of paths N_v passing through it. The algorithm considers an arbitrary order among the outgoing edges of a vertex. For the first edge e_1, $L(e_1) = 0$ and for an ith edge e_i, $L(e_i) = L(e_{i-1}) + N_{v_{i-1}}$ where $e_{i-1} = (v, v_{i-1})$. Figure 2 shows how the outgoing edges of a vertex v_0 are labeled. If an edge (v, v_i) is labeled before an edge (v, v_j) then the labeling ensures that all the paths passing though the edge (v, v_j) have greater path-ids than path-ids of paths passing through the edge (v, v_i). In the next step, a maximum spanning tree of the DAG is computed and labels are revised using an event counting algorithm [2] and placed only on the chords (complement of spanning tree edges).

2.2 Examples

Partitioning paths of a profitable procedure. Let a procedure P whose DAG is shown in Figure 1(a) be a profitable procedure. We describe our approach of classifying procedures into profitable and unprofitable in Section 3.2.

Suppose the two instances P_a and P_b of the procedure shown in Figure 1(a) and Figure 1(b) are constructed for profiling disjoint sets of paths of P. A path p is profiled in an instance if all the edges in $edges(p)$ are shown in solid lines in that instance. For example, the path $\langle entry, v_0, exit\rangle$ is profiled in the instance P_b but not in P_a. The edges in both the instances are labeled with valid labeling.

We have, $overhead(P_a) = 6$ and $overhead(P_b) = 8$. Now, consider another partition of the paths of P given by two instances P'_a and P'_b shown in Figure 3(a) and Figure 3(b), also labeled with valid labeling. Here, $overhead(P'_a) = 5$ and $overhead(P'_b) =$



(a) P'_a (b) P'_b

Figure 3: A partition (P'_a, P'_b) of the procedure P which results in less overheads than the partition of Figure 1.

4. Since the maximum overhead of (P'_a, P'_b) is smaller than the maximum overhead of (P_a, P_b), the partition (P'_a, P'_b) is likely to yield better performance than the partition (P_a, P_b).

We now analyze the cause of inefficiency in (P_a, P_b). Consider a path $p_0 = \langle entry, \cdots, v_6, v_7, v_8, \cdots, exit\rangle$. This path encounters two probes, respectively at (v_6, v_7) and at (v_8, v_{10}), in P_a and one probe (v_6, v_7) in P_b. Thus, the runtime overhead due to execution of this path affects both the instances. Similar is the case for the paths passing through $V_2 = \{v_{12}, v_{15}, v_{18}, v_{21}\}$. In general, it may be difficult to avoid such situations, but P3 performs a control flow analysis, whereby, it assigns all the paths passing though a sequence of conditionals, that do not have other conditionals nested within them, to only one instance. This is seen for the partition (P'_a, P'_b) shown in Figure 3. Here, all the paths passing through $V_1 = \{v_4, v_7, v_{10}\}$ are profiled in P'_a, whereas all the paths passing through V_2 are profiled in P'_b. Due to the resultant labeling, the runtime overhead due to execution of those paths (including p_0) will affect only one instance. This partitioning strategy reduces *overlapping profiling overheads*. For this example, P3 can compute (P'_a, P'_b) as the partition of paths of P.

Computing valid labeling of profitable procedures. The selective path profiling (SPP) algorithm [1] is designed to compute an edge labeling that assigns unique path-ids to a chosen subset of paths S. However, for a labeling to be valid, we additionally require that the path-ids of paths in S should be *distinct* from those of paths not in S. SPP does not satisfy this requirement as demonstrated below.

Consider the set of interesting paths S profiled in an instance of a procedure P shown in Figure 4(a). An edge that appears in some uninteresting path is shown in dashed lines in Figure 4(a) and Figure 4(b). We refer to such edges as *uninteresting edges*. Similar to the Ball-Larus algorithm, in the first step, SPP proceeds in the reverse topological order except that at each vertex, it processes all uninteresting (outgoing) edges before the interesting ones. Figure 4(a) shows the edge labels thus computed. In the next phase, SPP visits the vertices in the topological order and if (v, w) is the only incoming edge to w then SPP adds its label to the labels of all the outgoing edges of w and sets the label of (v, w) to zero. The labels obtained after this step are shown in Figure 4(b). In the third and final step, it sets the labels of all uninteresting edges to zero. In our example, the non-zero labels of uninteresting edges (v_3, v_{11}) and (v_{14}, v_{15}) are set to zero. The other uninteresting edges are anyway zero after the second step (see Figure 4(b)). The labeling after the third step is not shown due to space constraints.

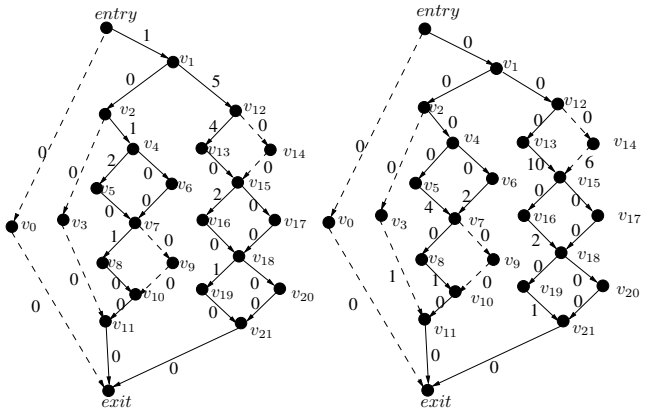


Figure 4: Steps of SPP: (a) labeling after the first step and (b) labeling after the second step.

Consider two paths $p = \langle entry, \cdots, v_6, v_7, v_8, \cdots, exit\rangle$ and $p' = \langle entry, \cdots, v_{14}, v_{15}, v_{16}, v_{18}, v_{19}, \cdots, exit\rangle$ where $p \in S$ and $p' \notin S$. However, under the labeling computed by SPP (in the third step), the path-ids of the two are the *same*, equal to 3. Since the edge (v_{14}, v_{15}) is uninteresting, as stated above, SPP sets its label to zero in the third step, resulting in this situation.

In order to overcome the overlapping path-ids assigned by SPP and to obtain valid labeling for individual instances of a profitable procedure, we design a variant of SPP, called the precise SPP (PSPP) algorithm. Figure 1(a) shows the labeling obtained by PSPP for the same subset of paths as in Figures 4(a) and 4(b). In particular, the paths p and p' identified earlier respectively get distinct path-ids 11 and 3 in Figure 1(a). We explain the computation of the valid labeling by PSPP in the next section.

3. ALGORITHMS

In this section, we first present the PSPP algorithm for profiling a subset of paths, followed by the P3 algorithm.

3.1 Precise Selective Path Profiling

Our precise selective path profiling (PSPP) algorithm is a variant of the SPP algorithm and computes only valid labeling. Before we design the algorithm, we analyze SPP in more depth.

In-depth analysis of SPP. Let us consider our running example in Figure 4(a) and understand the reason why SPP cannot construct a valid labeling. In the first step of SPP, an uninteresting edge may get a zero label but it may become non-zero after the second step which propagates edge labels in the topological order as described in Section 2.2. For example, the uninteresting edge (v_3, v_{11}) has a zero label in Figure 4(a) but gets a non-zero label in Figure 4(b).

In the third step, SPP sets non-zero labels of uninteresting edges to zero. We use the term *absorb* to denote when the non-zero label of an uninteresting edge is made zero in the third step of SPP. A path p *absorbs* when an edge in $edges(p)$ absorbs. Since interesting paths do not contain uninteresting edges, they are always unabsorbable. Note that before the third step, path-ids produced by SPP are unique and only after absorption, path-id of some uninteresting path may overlap with that of some interesting path.

Let $pid(p, v)$, called a *partial identifier*, denote the sum of edge labels of the suffix *suff*(p, v) of a path p from vertex v onwards. Clearly, $pathid(p) = pid(p, entry)$.

Consider two paths p and p' going through a vertex v such that they are respectively uninteresting and interesting paths. Further, let an edge (v, w) be an uninteresting edge that appears in p. Similarly, let an edge (v, w') be an interesting edge that appears in p'. SPP processes the uninteresting edges before interesting edges at v in the first step to try to make sure that $pid(p, v) < pid(p', v)$. $pid(p, v)$ remains less than $pid(p', v)$ after the second step and in particular, after absorption (the third step).

Example 1. At v_2, the uninteresting edge (v_2, v_3) is processed by SPP before the interesting edge (v_2, v_4) in Figure 4(a). Consider an uninteresting path $p_0 = \langle entry, v_1, v_2, v_3, v_{11}, exit\rangle$ going through v_2. In Figure 4(a), $pid(p_0, v_2)$ is less than the pid of the suffix starting at v_2 for any interesting path passing through v_2. After absorption, $pid(p_0, v_2)$ still remains less than pids of suffixes starting at v_2 for the interesting paths going through v_2. ■

Unfortunately, $pid(p, v)$ of an uninteresting path p which goes through an *interesting* outgoing edge of v can be *more* than that of an interesting path going through v. SPP does not ensure that they will not become equal after absorption as shown next.

Example 2. Consider paths $p = \langle entry, \cdots, v_6, v_7, v_8, \cdots, exit\rangle$ and $p' = \langle entry, \cdots, v_{14}, v_{15}, v_{16}, v_{18}, v_{19}, \cdots, exit\rangle$ in Figure 4(b) which get the same path-id (equal to 3) after the edge (v_{14}, v_{15}) absorbs, as discussed in Section 2.2. The key observation from Figure 4(a) is the following. In the first step of SPP, while processing v_1's outgoing interesting edges, (v_1, v_2) is processed *before* (v_1, v_{12}). This assigns the value 5 to the edge (v_1, v_{12}) as there are 5 paths passing through v_2. This labeling is faulty, as even though it makes $pid(p', v_1) > pid(p, v_1)$, they become equal after the absorption. The central problem of SPP is that it does not assign any order in processing of the interesting outgoing edges. ■

The PSPP algorithm. We now present the PSPP algorithm to remedy the faulty labeling arising in SPP. While SPP picks interesting outgoing edges of a vertex v during the first step in an *arbitrary* order, PSPP enforces a specific order among those edges. For a vertex v, PSPP processes the outgoing edges (v, w), in *the decreasing order of* $w.min$ where $w.min$ is defined as the minimum pid of the *interesting paths* starting from w. This, together with the processing of uninteresting outgoing edges before the interesting outgoing edges ensures valid labeling. Specifically, it ensures that an uninteresting path p passing through v after absorption results in $pid(p, v) < v.min$ and therefore will not have same path-ids with the interesting paths passing through v. This holds *irrespective* of whether p starts with an uninteresting or an interesting edge at v. This is in contrast with SPP, since such a claim is valid for SPP only if p starts with an uninteresting edge at v as discussed earlier.

With this intuition, we next describe the PSPP algorithm in detail (see Algorithm 1). PSPP takes a DAG G and an edge set EI of edges appearing in interesting paths as input and produces a *valid* labeling of edges in G. The label of an edge e is denoted by $e.val$.

Algorithm 1 initializes $v.min$ to ∞ for all vertices of G except the $exit$ vertex (line 2). For the $exit$ vertex, it initializes $exit.min$ to zero and N_{exit} to one (line 3). In the loop at lines 4–13, PSPP iterates over the vertices of G (excluding the $exit$ vertex) in the reverse topological order. This is similar to the first step of SPP but differs from SPP in the order in which interesting outgoing edges are processed, as explained below. For each vertex v, it initializes N_v to zero (line 5). In the loop at lines 6–8, it first iterates over the uninteresting outgoing edges of v and for each uninteresting edge $e = (v, w)$, it sets $e.val$ to zero. It also accumulates the value of N_w in N_v (line 7). Then, in the loop at lines 9–12, it iterates over the interesting outgoing edges in the decreasing order of $w.min$. For each such edge $e = (v, w)$, it assigns the current value of N_v to $e.val$ (line 10), updates N_v by adding N_w (line 10) and updates the value of $v.min$ by taking the minimum of the current

Algorithm 1: PSPP(G,EI)

Input: A DAG $G = (V, E, entry, exit)$ and a set of interesting edges $EI \subseteq E$
Output: A valid edge labeling of G

```
1  begin
2    foreach v ∈ V \ {exit} do v.min ← ∞
3    exit.min ← 0; N_exit ← 1
4    foreach non-exit node v in reverse topological order do
5      N_v ← 0
6      foreach uninteresting edge e = (v, w) do
7        e.val ← 0; N_v ← N_v + N_w
8      end
9      foreach interesting edge e = (v, w) in decreasing order
       of w.min do
10       e.val ← N_v; N_v ← N_v + N_w
11       v.min ← Minimum(v.min, w.min + e.val)
12     end
13   end
14   foreach non-exit node v in topological order do
15     if v has only one edge e_i = (u, v) and e_i.val > 0 then
16       foreach e_o = (v, w) do
17         e_o.val ← e_o.val + e_i.val
18       end
19       e_i.val ← 0
20     end
21   end
22   foreach edge e not in EI do e.val ← 0
23 end
```

value of $v.min$ and $w.min + e.val$. The loops at lines 14–21 and line 22 implement the second and third steps of SPP respectively. In particular, the loop at line 22 performs *absorption*.

Example 3. We now revisit the scenario of faulty labeling of the outgoing edges of v_1 by SPP discussed in Example 2 and show how PSPP remedies it. The processing of the first step of PSPP will yield the same edge labeling as SPP for the subgraphs rooted at v_2 and v_{12} as shown in Figure 4(a).

When v_1 is processed, $v_2.min = 2$ (corresponding to the path $\langle entry, \cdots, v_4, v_6, v_7, v_8, \cdots, exit\rangle$) and $v_{12}.min = 4$ (corresponding to the path $\langle entry, \cdots, v_{13}, v_{15}, v_{17}, v_{18}, v_{20}, \cdots, exit\rangle$). Therefore, (v_1, v_2) is processed *later* than (v_1, v_{12}) which makes $e_1.val = 0$ and $e_2.val = 8$ where $e_1 = (v_1, v_{12})$ and $e_2 = (v_1, v_2)$. The final labeling obtained after propagation and absorption of labels by PSPP is shown in Figure 1(a). The paths

$$p = \langle entry, \cdots, v_6, v_7, v_8, \cdots, exit\rangle$$
$$p' = \langle entry, \cdots, v_{14}, v_{15}, v_{16}, v_{18}, v_{19}, \cdots, exit\rangle$$

will have path-ids 11 and 3, respectively, after the absorption step absorbs the label 1 on (v_{14}, v_{15}), propagated from $(entry, v_1)$.

Consider another path p_1=$\langle entry, v_1, v_2, v_3, v_{11}, exit\rangle$ in Figure 1(a). Since it passes through an outgoing uninteresting edge from v_2, $pid(p_1, v_2)$ (same as $pathid(p_1)$ due to absorption) remains less than $v_2.min$ and consequently less than $v_{12}.min$. This prohibits any chance of colliding with interesting paths passing through v_{12}. Besides, it can be seen that any uninteresting path p_2 having $pid(p_2, v_2)$ higher than $v_2.min$ is *unabsorbable* and hence, its path-id will not be changed by PSPP. ■

Proof of correctness. We show that the final labeling produced by PSPP is a valid labeling (as defined in Section 2.1). A detailed proof of this claim is included in Appendix A.

3.2 Partitioned Path Profiling

We now present the partitioned path profiling (P3) algorithm. P3 uses the PSPP algorithm presented in Section 3.1 as a sub-routine.

Data structures and helper functions. A program M is a set of procedures. For a DAG $G = (V, E, entry, exit)$ of a procedure P, $G.E$ and $G.V$ respectively denote the set of edges and vertices of G. For a vertex $v \in V$, $v.pathcount$ gives the number of paths from the $entry$ vertex to v. $\mathcal{T}$ denotes a set of tasks where each task, denoted by T, corresponds to a subset of paths (say T_p) in the program. For each $T \in \mathcal{T}$, we maintain two fields: $T.E$ and $T.Cost$ where $T.E$ is the union of $edges(p)$ for all $p \in T_p$ and $T.Cost$ gives the number of instrumentation probes required by PSPP for profiling the paths in T_p if $T.E \subset G.E$. If $T.E = G.E$ then $T.Cost$ is the number of probes required by the Ball-Larus algorithm for profiling the paths in T_p.

The function BL_DAG(P) returns the DAG based on the DAG construction algorithm [4] for a procedure P and BL(G) returns the edge labeling for profiling all paths in a DAG G using the Ball-Larus algorithm. The function $size(X)$ returns the number of elements in the set X. The function $caller\_count(P)$ gives the number of call-sites of P in the program M. For a DAG G, a set of four vertices $\{a, b, c, d\}$ forms a *diamond* if they have the following edges among them $\{(a, b), (a, c), (b, d), (c, d)\}$. For example, $\{v_4, v_5, v_6, v_7\}$ in Figure 3(a) forms a diamond. A *triangle* is a set of three vertices $\{a, b, d\}$ such that they have the following edges among them $\{(a, b), (a, d), (b, d)\}$. We call the vertices a and d as *begin* and *end* vertices. These control flow structures respectively represent if-else and if statements containing only straightline code within the branches (equivalently, they do not contain nested conditionals). The function $reduced\_graph(G)$ returns a new DAG, called a *reduced DAG*, where all diamonds and triangles in G are replaced by new vertices, called *dummy vertices*. The incoming edges to the begin vertex of a diamond (or a triangle) in G are added as incoming edges to the dummy vertex it is replaced with. The case of outgoing edges of the end vertex is analogous.

Let es be a set of edges in the reduced DAG G' obtained from a DAG G. For an edge $e = (v, d)$ or $e = (d, w)$ such that d is a dummy vertex, let $original(e)$ contain the set of edges in G belonging to the diamond or triangle that d replaced while obtaining G'. If x and y are the begin and end vertices of the diamond (or triangle) corresponding to d then we also add $\{(v, x) \mid (v, d) \in es\} \cup \{(y, w) \mid (d, w) \in es\}$ to $original(e)$. The function $get\_original\_edges(es, G, G')$ returns the set of edges $\{(v, w) \in es \mid v, w$ are not dummy vertices$\} \cup \{e \in original(e') \mid e' = (v, d)$ or $e' = (d, w)$ for some dummy vertex d in G' and $e' \in es\}$. Finally, the function $reachable\_edges(v, G')$ returns all the edges reachable from v in G'.

The P3 algorithm. The P3 algorithm is presented in Algorithm 2. It takes a program M and a finite set C of identical cores. For each core $C_i \in C$, we maintain the following fields: (1) $C_i.E$: set of edges of the paths profiled in C_i, (2) $C_i.load$: computed as the number of edges in $C_i.E$ and (3) $C_i.Probes$: set of probes in C_i (output of P3).

Lines 3–10. If the paths in the DAG of a procedure are partitioned and assigned to different copies, there may be overlap of probes across the copies. This can cause increase in runtime for both the copies. Since we cannot completely eliminate such overlap, we try to limit its impact by considering only those procedures that may get called at most once in any execution. P3 therefore applies partitioning to procedures (by calling find_partition at line 5) which have no more than one caller. For the rest of the procedures, it

Algorithm 2: P3(M, C)

Input: A program M and a finite set C of identical cores
Output: An assignment of instrumented copies of M to the cores

```
1  T ← ∅ // a global variable to store tasks
2  begin
3    foreach procedure P ∈ M do
4      DAG G ← BL_DAG(P)
5      if call_count(P) ≤ 1 then find_partition(G)
6      else
7        probes ← BL(G)
8        T.E ← G.E;  T.Cost ← size(probes)
9        Add T to T
10     end
11   end
12   foreach task T ∈ T in the decreasing order of T.Cost do
13     Let C_i be the core with the minimum load
14     C_i.E ← C_i.E ∪ T.E
15     C_i.load ← C_i.load + T.Cost
16   end
17   foreach C_i ∈ C and P ∈ M do
18     G ← BL_DAG(P)
19     E_P ← G.E ∩ C_i.E
20     if E_P = G.E then Add BL(G) to C_i.Probes
21     else Add PSPP(G,E_P) to C_i.Probes end
22   end
23 end

24 Function find_partition(G)
25 begin
26   Let G = (V, E, entry, exit)
27   G' ← reduced_graph(G)
28   foreach v ∈ G'.V \ {entry} do v.pathcount ← 0, v.SE ← ∅
29   entry.SE ← {∅}; entry.pathcount ← 1
30   totalpathcount ← 1; S ← ∅
31   foreach vertex v ∈ G'.V in topological order do
32     totalpathcount ← totalpathcount − v.pathcount
33     foreach (v, w) ∈ G'.E do
34       add (v, w) to S
35       foreach es ∈ v.SE do
36         Add es ∪ {(v, w)} to w.SE
37         Increment w.pathcount and totalpathcount by 1 each
38         if totalpathcount = Δ then goto LBL
39       end
40     end
41   end
42   LBL: foreach v s.t. ∃(u, v) ∈ S ∧ ∃(v, w) ∉ S do
43     foreach es ∈ v.SE do
44       foreach (v, w) ∈ G'.E ∧ (v, w) ∉ S do
45         add {(v, w)} ∪ reachable_edges(w, G') to es;
46       end
47       T.E ← get_original_edges(es, G, G')
48       probes ← PSPP(G,T.E)
49       T.Cost ← size(probes)
50       Add T to T
51     end
52   end
53 end
```

applies the classic Ball-Larus algorithm to determine the probes (lines 7-9) and creates a task for each such procedure.

Lines 24–52. The find_partition function combines the paths in G that pass through some diamonds or triangles into one task. It does so by first creating a reduced DAG as explained earlier. These paths will share lots of overlapping instrumentation probes and are therefore more suited for profiling in the same core.

Example 4. Consider the example in Figure 3. It contains five diamonds. The corresponding reduced DAG is formed by replacing the five diamonds with dummy vertices. There will be four paths in the reduced DAG and corresponding to each such path, P3 creates one task. Note that the resultant four tasks in the above example can be profiled without any overlapping probes among them. ■

For large procedures, the number of paths in the reduced graph can be large. Therefore, we employ a threshold Δ on the number of tasks generated from a procedure. The tasks, in the presence of a threshold, are computed on the reduced DAG as follows. For a vertex v in the reduced DAG, each element of the set $v.SE$, denoted by es, refers to the set of edges corresponding to a path from $entry$ to v. Starting from $entry$, the vertices of the reduced DAG are traversed in topological order (line 31). While processing v, $w.SE$ is updated (line 36) for each successor w of v. The process terminates when the number of paths found ($totalpathcount$) is equal to the threshold Δ. S accumulates all the edges *covered* during this process. In the loop beginning at line 42, a vertex v is selected which is either $exit$ or not all of its successors are processed in the previous iteration. For v, all edges reachable from v to $exit$ passing through uncovered edges are added to each edge-set in $v.SE$ (line 45). Each such edge-set is mapped to the edges in G using the function $get\_original\_edges$ (line 47) and a task is created using the edge-set defined over G.

For some procedures (e.g., those without diamonds or triangles), even if find_partition is invoked at line 5, it will return only a single task. A procedure on which find_partition is invoked and it returns multiple tasks is called a *profitable procedure*.

Lines 12–16. For getting optimal benefit out of the distribution, one has to minimize the maximum time taken across all cores. It turns out that the optimal distribution to minimize the maximum cost across all cores is an NP-complete problem. The hardness can be shown by reduction from multiprocessor scheduling problem [14]. In the multiprocessor scheduling problem (MSP), we are given m identical machines $M_1, \ldots, M_m$ and n jobs $J_1, \ldots, J_n$. Job J_i has a processing time $p_i > 0$ and the goal is to assign jobs to the machines so as to minimize the maximum load. The load of a machine is defined as the sum of the processing times of jobs that are assigned to that machine. In our context, a task (T) is considered as equivalent to a job in MSP and $T.Cost$ is considered as the processing time p_i for a job J_i. We use a known 4/3th approximation algorithm [15] for multiprocess scheduling for distribution in P3. Here, in a loop, the highest-cost task among the remaining non-distributed tasks is assigned to the core with least load.

Example 5. In the example in Figure 3, the four tasks have cost 1 ($\langle entry, \cdots, v_0, \cdots, exit\rangle$), 1 ($\langle entry, \cdots, v_3, \cdots, exit\rangle$), 3 (for the 4 paths passing through v_4), and 7 (for the 8 paths passing through v_{12}). If only two cores are available, the 4th task is assigned to the first core and first three tasks are assigned to the second core. This results into the distribution shown in Figure 3. ■

Lines 17–21. Finally, for each core, P3 collects all the interesting edges of the same procedure in the core and calls PSPP or BL to get the final set of probes for the procedure on those edges.

4. EXPERIMENTAL EVALUATION

In this section, we explain our implementation and experimental setup. We then report the experimental results on several programs from the SPEC 2006 benchmark [16].

4.1 Implementation

We have implemented the PSPP and P3 algorithms for sequential C/C++ programs using the LLVM 3.3 infrastructure [19]. LLVM has an implementation of the Ball-Larus algorithm and we use it as a sub-routine in P3's implementation. For the experiments, for each

Table 1: Benchmark characteristics

ID	Program name	LOC	#Procedures	#Profitable procedures
1	473.astar	4694	167	12 (7%)
2	403.gcc	1738852	4347	540 (12%)
3	445.gobmk	158600	2476	556 (22%)
4	456.hmmer	22049	471	163 (35%)
5	464.h264ref	37694	518	97 (19%)
6	470.lbm	1159	17	1 (6%)
7	462.libquantum	3353	115	14 (12%)
8	429.mcf	2225	24	5 (21%)
9	433.milc	10560	235	35 (15%)
10	998.rand	339	3	1 (33%)
11	999.rand	339	3	1 (33%)
12	458.sjeng	10896	144	14 (10%)
13	482.sphinx3	17585	319	69 (22%)

procedure, we choose the threshold on the number of partitions that P3 creates as the maximum of the out-degree of the $entry$ vertex of its DAG and the number of cores. Note that, BL_DAG can create a DAG having vertices with out-degree more than two.

We specialized our P3 implementation to derive an implementation of the *parallel Ball-Larus strategy* (PBL) outlined in the Introduction. More specifically, in our PBL implementation, all procedures are considered as *unprofitable* and are instrumented using the Ball-Larus algorithm to be distributed subsequently. The *sequential Ball-Larus strategy* (SBL) is same as the Ball-Larus profiling on a single core. We compare the profiling overheads of *three* different techniques: P3, PBL and SBL.

4.2 Experimental Setup

Setup. We use programs from the SPEC 2006 benchmark [16] for experimental evaluation. SPEC benchmarks are popular evaluation targets in the profiling literature. We could run the Ball-Larus implementation supplied with LLVM on 13 C/C++ programs from this benchmark. We evaluate the profiling algorithms on all these programs (see Table 1). These comprise both large programs such as 403.gcc and some small programs such as 999.rand. Most programs have several thousand lines of code and a few hundred procedures. The SPEC benchmark also provides a few tests per program. To evaluate the runtime overhead of profiling, we run each of the programs in Table 1 on all the tests available for it.

The experiments were conducted on Ubuntu Linux 12.04 on an Intel Xeon W3520 2.67GHz machine with 4 cores and 8 GB RAM. We simulate different cores by running the distinct copies generated by the profiling algorithms separately on a single core of this machine. We present results of the parallel path profiling techniques on 2, 4 and 8 cores.

Quantifying profiling overhead. We quantify the runtime overhead of profiling using a metric, called hit count. The *hit count* of a procedure P on a test X is the number of times *instrumentation probes* inserted into P got executed while running the test X. The hit count of a copy C on a test X is the summation of the hit counts of all the procedures in that copy on X. The time overhead for profiling P under X on a copy C is proportionate to its hit count. Measuring real-time can have some inaccuracies based on the processor load and other environmental factors. Further, we have to accurately distinguish between the actual execution time and the time taken by instrumentation probes. Hit count directly quantifies the profiling overhead independent of these issues.

If an algorithm A generates K copies $C_1, \ldots, C_K$ for a program M then the profiling overhead of A for M on a test X is taken as the maximum of the profiling overheads of C_i for M on X, for $1 \leq i \leq K$. For a program M and an algorithm A, we consider

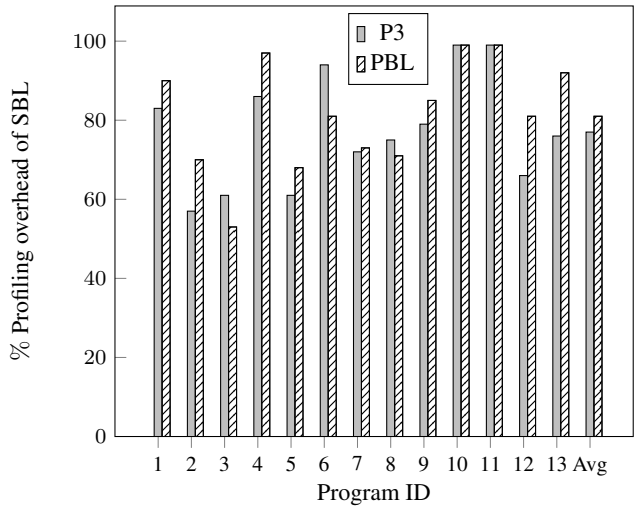


Figure 5: Profiling overhead of P3 and PBL relative to SBL on 2 cores (lower is better): Avg is the average over the programs.

the average of the profiling overhead of A across all the tests of M as the *profiling overhead*. Average across all programs is simply referred to as average and identified by Avg in the figures.

4.3 Experimental Results

RQ1. How many procedures were deemed to be profitable by P3?

As discussed in Section 3.2, P3 automatically identifies profitable procedures by a static control flow analysis. Table 1 shows the number and percentage of procedures that P3 deemed profitable for each of the programs. For each of them, P3 could generate at least 2 disjoint subsets of paths. The profitable procedures range from 6–35% of all procedures across the programs. Thus, in each of the programs, P3 could identify opportunities for load balancing across cores through profitable procedures.

RQ2. Does P3 reduce profiling overhead compared to SBL?

The key test of effectiveness of P3 is whether and how much reduction in profiling overhead (defined in Section 4.2) does P3 achieve compared to the sequential Ball-Larus (SBL) strategy.

In Figure 5, we plot the profiling overhead of P3 relative to SBL on 2 cores. On the X-axis, we represent different program IDs assigned to the programs in Table 1. The Y-axis is labeled with the percentage of the profiling overhead of SBL at the intervals of 20%. A filled gray bar shows the percentage of P3's profiling overhead relative to that of SBL for the same program. Lower the value of a Y-coordinate, the more reduction in profiling overhead P3 achieved. It can be seen that with only 2 cores, P3 could reduce the profiling overhead for most of the programs. For 8 programs, the reduction is more than or equal to 20%, whereas for 3 programs (IDs 6, 10 and 11), it is only marginal. The average reduction across all the programs is 23% as shown in the last bar of Figure 5.

In Figure 6 and Figure 7, we plot the profiling overhead of P3 relative to SBL for 4 and 8 cores respectively. For 4 cores, the reduction is more than or equal to 47% for 8 programs, whereas for 5 programs (IDs 4, 6, 7, 10 and 11), it is less than or equal to 30%. The average reduction for 4 cores is 43%. Finally, for 8 cores, the reduction is more than or equal to 50% for 11 programs, whereas it is less than or equal to 25% for the remaining two programs. The average reduction for 8 cores is 56%.

Overall, P3 achieved substantial parallelization benefits for most of the programs across 2, 4 as well as 8 cores.

RQ3. Does P3 reduce profiling overhead compared to PBL?

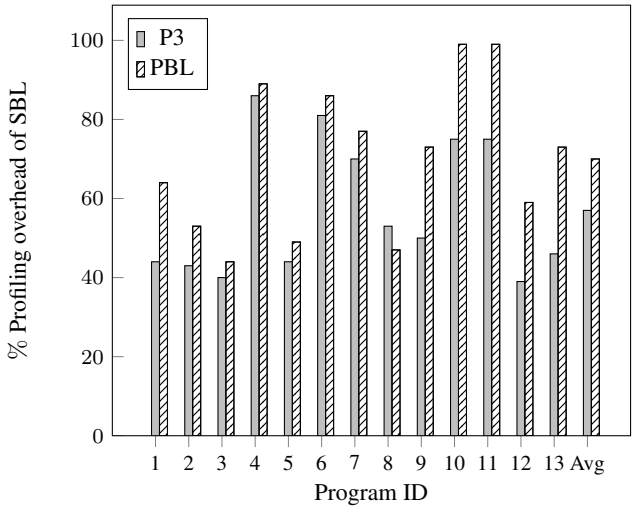


Figure 6: Profiling overhead of P3 and PBL relative to SBL on 4 cores (lower is better): Avg is the average over the programs.

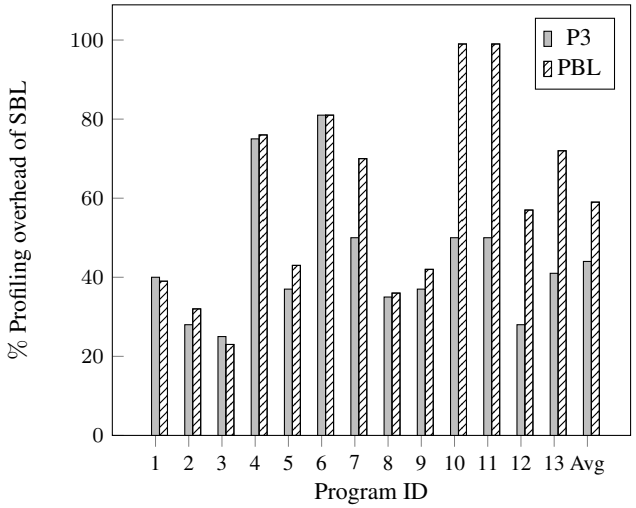


Figure 7: Profiling overhead of P3 and PBL relative to SBL on 8 cores (lower is better): Avg is the average over the programs.

We now compare P3 and PBL. Similar to P3, for each of the programs, we plot the profiling overhead of PBL relative to SBL for 2, 4 and 8 cores in Figures 5, 6 and 7 respectively. The bars with cross lines correspond to PBL.

For 2 cores (Figure 5), for 8 programs P3 shows more reduction compared to that of PBL. On an average across all the programs, the profiling overhead of P3 is 5% less compared to PBL. For 4 cores (Figure 6), P3 gives more reduction than PBL on all but 1 programs. On an average across all the programs, the profiling overhead of P3 is 18% less compared to PBL. Finally, for 8 cores (Figure 7), P3 outperforms PBL in all except 2 cases. On an average, it has 25% less overhead than PBL. We see two reasons behind the cases where P3 could not perform better than PBL: (1) overlapping of probes among the copies and (2) selection of profitable procedures did not have any effect on the result.

In summary, P3 is more effective in exploiting parallelism in path profiling compared to PBL. P3's strategy of automatically identifying profitable procedures and sub-dividing the task of profiling their paths helps reduce profiling overhead compared to PBL's coarse-grained strategy of distributing entire procedures.

RQ4. How much time does P3 take to construct different instrumented copies and assign them to different cores?

P3 constructs different instrumented copies of a program M and assigns them to different cores by a static analysis of M. P3 took a maximum of 48m for the largest program in Table 1. On an average, it took 5m across all the programs with 9 programs taking less than 2 minutes. We highlight that this is only a one-time cost for any program for a given number of cores. The program can then be profiled in parallel for any number of inputs.

4.4 Discussion

Trade-off between overlapping probes and load balancing. Overlapping probes are those probes that may get executed in an instance P_i for a path profiled in another instance P_j. An important factor in achieving reduction in profiling overhead through P3 is to achieve a trade-off between overlapping probes (which can increase the profiling overhead) and load balancing that can be achieved through profiling only a subset of paths in each core. Our experimental results show that this is indeed possible in practice. In particular, we observed that in several cases, multiple acyclic paths of a procedure were exercised on the same core while using PBL but they were profiled on *different* copies when P3 was used. In addition, the overlapping probes between those copies for the procedure did not overshadow the benefit of distribution of the paths.

Threats to validity. There are some threats to validity for our experimental results. The main among them being the limited number of programs and test inputs for them. We attempt to mitigate it by considering the SPEC benchmarks which are widely used in the profiling literature and generally, in performance analysis. Nevertheless, in future, we wish to run our experiments on other programs. The second threat is due to possible non-determinism in the paths being explored in different copies. However, we consider only sequential programs and evaluate them on the same machine (see Section 4.2). Thus, once we fix an input, all copies of the program follow the same dynamic control flow path. We give a detailed proof (see Appendix A) of correctness of PSPP to eliminate the possibility of a theoretical glitch. Finally, we reduced the possibility of bugs in our implementation by manual inspection and repeated experiments on both smaller, hand-written examples and the SPEC benchmarks.

5. RELATED WORK

Program profiling. Ball and Larus [4] introduced the notion of acyclic intra-procedural path profiling and provided an algorithm to compute it. The Ball-Larus algorithm has been extended to profile inter-procedural paths by Melski et al. [21] and to cyclic paths by D'Elia et al. [11]. We believe that P3 can be extended to cover these extensions. Extending P3 to profile inter-procedural paths is an immediate future work for us.

Recently, Li et al. [20] presented an algorithm to overcome the impreciseness of SPP [1]. Their algorithm, called Modified SPP (MSPP), deletes a label from an uninteresting edge only if it does not result in an invalid labeling. MSPP is an exponential algorithm in worst-case whereas PSPP is linear in the size of the DAG.

Vaswani et al. [24] introduced preferential path profiling (PPP) which reduces the overhead of path profiling by profiling a given set of paths with an objective of compact numbering. The compact numbering facilitates the use of arrays for updating the frequency for each acyclic path, thereby reducing the overhead caused by the use of hashtable. It can be shown with an example that their algorithm does not ensure computation of valid labeling. In contrast, the PSPP algorithm computes only valid labeling. As a workaround, PPP uses Ball-Larus' labeling for distinguishing interesting and un-

interesting paths. This workaround cannot be used in our scenario as the overhead will be same as overhead for Ball-Larus' labeling irrespective of the interesting paths. Chilimbi et al. [9] extended PPP for inter-procedural paths and used it for residual path profiling. We plan to extend PSPP for efficient residual path profiling.

Pertinent path profiling [5] introduced a new control-flow entity, namely, pertinent paths that pass though a given set of nodes called pertinent nodes. It generates a unique numbering for pertinent paths and generates compact numbering of path-ids. They do not try to reduce the number of probes, instead try to reduce the path-table size. Targeted path profiling [18] addressed the profiling overhead problem by leveraging edge profiling information in the context of staged dynamic optimization systems. Parallelizing edge-profile in itself can be an interesting research problem. In P3, the task cost can take into account such profiling information for better distribution of tasks into cores.

One program, many copies. Closest to our work is distributed program tracing [23]. It collects a single program trace corresponding to a given input by distributing the witnesses across multiple copies of the program and run them parallel on the same input to collect the partial traces. The partial traces are then merged to produce the whole trace. Though the same code-replication based divide-and-conquer strategy is applied in the context of a different problem, the challenge there was to devise the necessary and sufficient condition which guarantees that the original *order* of basic blocks can be constructed by merging. The distribution of witnesses and merging algorithm are therefore crucial for soundness of the algorithm. In contrast, here, the distribution is addressing the efficiency of the technique and the PSPP algorithm, applied on each procedure locally on each core, is ensuring the soundness. In both the works, the distribution strategies further address the efficiency of profiling or tracing. In tracing, a sequence of diamonds is put together in one copy as they can be *covered* by less number of witnesses, whereas P3 uses the notion of profitable procedures to optimize.

Software tomography [7] splits monitoring tasks across many instances of the software, so that partial information may be collected from users by means of light-weight instrumentation and merged to gather the overall monitoring information. Although sounds similar, the main difference is that they do not try to obtain accurate profiling information for a given set of paths. There technique distributes the monitoring tasks to different users who can use the software at will with different inputs. Their goal is to gather enough information for each sub-task. For example, for path coverage, they discover whether each path is executed in a given set of executions, whereas our goal is to obtain precise execution frequencies in a given task. Thus, their framework is more suitable towards efficient distributed profiling (estimation based) whereas our algorithm is more suitable towards accurate parallel path profiling. Additionally, their algorithm is not accurate as it is as based on SPP [1].

Diep et al. [12] consider distribution of probes to multiple program variants, where each variant contains a subset of probes, where the subset size can be bounded to meet the overhead requirements. However, the aim there is to profile a set of events and not paths.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an algorithm called P3 for parallel path profiling. To the best of our knowledge, this is the first algorithm for parallel path profiling. P3 profiles the path of a program by distributing all acyclic paths into multiple cores, running on the same input. P3 judiciously performs partitioning of paths of some selected (profitable) procedures to reduce the common overhead caused by the execution of a path in multiple cores. It uses an approximation algorithm to evenly distribute the overhead based on number of probes. To precisely estimate execution frequencies for each subset of paths in a profitable procedure, we have developed an algorithm called PSPP which we use in P3.

This paper opens up some interesting research problems: how to extend P3 for inter-procedural or cyclic paths which poses the challenge of extending PSPP for such paths. As the Ball-Larus algorithm can benefit by previously available profiling information for determining low frequency chords to place the probes, P3's distribution algorithm can be extended to get benefit from such information. In the absence of dynamic profiling information, it is possible to do static estimation [3] based on program's inter-procedural CFG. This paper also opens up possibility of optimizations based on better selection strategy of profitable procedures and threshold.

APPENDIX A. CORRECTNESS OF PSPP

We now show that the final labeling produced by PSPP is a valid labeling (as defined in Section 2.1). We first define local validity of a labeling L at a vertex v. Let I_v and $\bar{I}_v$ denote the set of interesting and uninteresting paths passing through the vertex v respectively. The labeling L is *locally valid* at v if the following conditions hold for pids obtained using L:

(A) $\forall p, q \in I_v : pid(p, v) \neq pid(q, v)$
(B) $\forall p \in I_v, \forall r \in \bar{I}_v : pid(p, v) \neq pid(r, v)$

This definition is similar to the definition of valid labeling but uses partial identifiers pids instead of $pathid$s.

Since $pathid(p) = pid(p, entry)$, a locally valid labeling for $v = entry$ is same as a valid labeling. Thus, by proving local validity for each vertex, we can prove the validity of the labeling produced by PSPP. We now give names to the different steps of PSPP for simplicity. The loop at lines 4–13 is called the *initial step*. The loop at lines 14–21 is called the *propagation step* and finally, the loop at line 22 is called the *absorption step*.

It is easy to see that local validity holds at every vertex for the (edge) labeling obtained after the initial step. In PSPP, the propagation and absorption steps are performed after the initial step is over for all the vertices. To show that local validity holds for each vertex after the absorption step, we define a variant of the PSPP algorithm called *PSPP'* in which propagation and absorption steps are *interleaved* with the loop of the initial step. In particular, the propagation and absorption loops are run within the loop of the initial step, immediately after a vertex v is processed in each iteration of the loop at lines 4–13, by treating v as the entry vertex of the subgraph of G rooted at v. We first show equivalence of PSPP and PSPP', and then prove correctness of PSPP'.

LEMMA 1. *The edge labeling computed by PSPP and PSPP' are identical.*

PROOF. This follows from the fact that PSPP' also performs the propagation and absorption at the $entry$ vertex, same as that in PSPP. Since the propagation step is iterative and it considers each vertex in the topological order for propagation of edge labels to the outgoing edges of vertices with in-degree 1, the final edge labeling of the interesting edges is the same for both PSPP and PSPP'. All the uninteresting edges are labeled 0 in both PSPP and PSPP'. □

Let $succ_i(v) = \{w \mid (v, w) \in EI\}$. Henceforth, we refer to visit of a vertex v by PSPP' as the visit of v in its outermost loop.

LEMMA 2. *For a vertex v, the value $v.min$ is only updated (line 11 of Algorithm 1) for the first interesting edge (v, w_1) PSPP (and also PSPP') visits in the initial step. The value $v.min$ is at least $w_1.min$. Also, $\forall w_i \in succ_i(v)$, $w_i.min \leq v.min$.*

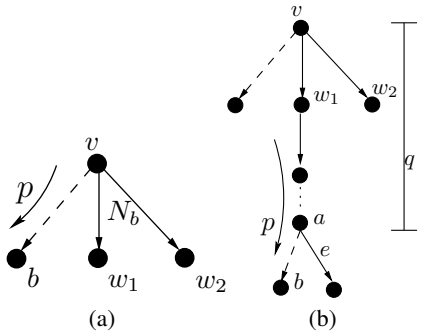


Figure 8: (a) Case 1: First non-interesting edge of p starts from v; (b) Case 2: First non-interesting edge of p starts after prefix q from v

PROOF. PSPP essentially follows Ball-Larus' edge labeling for interesting edges and the above lemma follows from the fact that Ball-Larus' edge labeling always gives higher *pid* to the paths passing through the edges that are visited later. Thus, the interesting path from v with minimum path-id always passes through the first interesting edge visited by PSPP. Since the value assigned to an interesting edge is at least 0, the minimum path-id of an interesting path through v is atleast $w_1.min$. Also, $\forall\ w_i \in succ_i(v)$, as $w_i.min \leq w_1.min$, we deduce $w_i.min \leq v.min$. □

LEMMA 3. *After PSPP' visits a vertex v, it ensures that an uninteresting path p having $pid(p, v)$ higher than $pid(p', v)$ of any interesting path p' is always unabsorbable.*

PROOF. We prove this by contradiction. Suppose there exists such an uninteresting path p which is absorbable. Let $w_1, w_2...., w_n$ be the successors of v. Let (a, b) be the first uninteresting edge of p reachable from v. We consider two cases for p.

First case: (a, b) is the leading edge of the subgraph rooted at v (means $a = v$). Refer to Figure 8(a) for an illustration. All the interesting paths will be assigned higher *pid*s than the *pid* of the uninteresting path p since interesting edges are processed after uninteresting edges. This contradicts our assumption.

Second case: (a, b) is not the leading edge (see Figure 8(b)). That means there exists a prefix q consisting of all interesting edges (without more than one incoming edge) till the edge (a, b) of this path which makes it absorbable. Now, consider a path q equal to $\langle entry, \ldots, v, w_1, x_1, x_2, \ldots, x_n, a, \ldots, exit\rangle$. For an interesting edge e passing through the vertex a, $e.val$ will get higher value than the *pid* of the uninteresting path with respect to a because interesting edges are processed later than uninteresting edges. Thus, the value of $a.min$ will be higher than the $pid(p, v)$ as after PSPP' completes processing of v, all labels of edges in q and (a, b) are zero due to propagation and absorption from v. Using Lemma 2, $x_n.min \geq a.min$. Lemma 2 can be used to show the inequality $X.min \geq a.min$ for X ranging over all the vertices $x_{n-1}, \ldots, x_1, w_1, v$, since each of the vertices is the successor of the next node. Hence, $v.min \geq a.min$. This means that the minimum *pid* of the interesting path passing from v is at least $a.min$. But $a.min > pid(p, v)$. This contradicts our assumption. □

THEOREM 1. *Given a DAG and a set of interesting edges, after PSPP' visits a vertex v, the labeling obtained is locally valid at v.*

PROOF. The proof is by induction on the *height* of a vertex in the DAG. The height of a vertex v is the smallest number of edges between v and the $exit$ vertex of the DAG.

Base case: v has height equal to zero (that is, $v = exit$). The theorem trivially holds.

Induction step: We show that the theorem holds for any vertex v of height $H > 0$. Since we are considering a DAG, all successors $w_1, w_2...., w_n$ of v have height less than H, so by induction hypothesis the local validity holds on all w_i. The algorithm assigns Ball-Larus' edge labeling to the interesting edges and after computing propagation and absorption steps, the *pid* of all the interesting paths remains intact since there are no uninteresting edges in interesting paths. So the *pid*s of the interesting paths satisfy part (A) of the definition of locally valid labeling. For proving part (B) of the local validity definition, partition the uninteresting paths passing through v in two groups N_1 and N_2. N_1 consists of all the uninteresting paths which pass through leading uninteresting edges of v, whereas N_2 consists of all the uninteresting paths passing through leading interesting edges of v. Again, its trivial to see that: $\forall p \in N_1$ and $\forall q \in I_v$, $pid(p, v) < pid(q, v)$, since all the interesting edges are processed later than the uninteresting edges and interesting edges are assigned in increasing order.

For proving the distinctness of *pid*s of paths in I_v and those in N_2, we consider following two cases:

Case a = Paths passing through the same interesting edge from v: We first prove the distinctness of the path-ids of interesting paths I_v and path-ids of the paths in N_2 which pass through the same outgoing edge $e = (v, w_i)$ of v. Consider two such paths $p_1 \in N_2$ and $p_2 \in I_v$. By induction hypothesis, $pid(\mathit{suff}(p_1, w_i), w_i) \neq pid(\mathit{suff}(p_2, w_i), w_i)$. We need to prove that after propagation and absorption of e's edge label (say α) their *pid*s (w.r.t. v) are disjoint. By Lemma 3 on w_i, p_1 is absorbable if $pid(\mathit{suff}(p_1, w_i), w_i) < pid(\mathit{suff}(p_2, w_i), w_i)$. As no absorption happens to interesting paths, $pid(p_2, v) = \alpha + pid(\mathit{suff}(p_2, w_i), w_i)$. For such a p_1, $pid(p_1, v) < \alpha + pid(\mathit{suff}(p_1, w_i), w_i)$ as some absorption happens in p_1. Therefore $pid(p_1, v) < \alpha + pid(\mathit{suff}(p_2, w_i), w_i) = pid(p_2, v)$. In the other case, p_1 is not absorbable therefore its $pid(p_1, v)$ will not change after absorption and remain different than $pid(p_2, v)$.

Case b = Paths passing through different outgoing edges from v: We now prove the distinctness of the *pid*s of the interesting paths and the path-ids of uninteresting paths in N_2 passing through different edges (v, w_i) and (v, w_j). Let, without loss of generality $w_i.min >= w_j.min$.

PSPP' processes edge (v, w_i) before edge (v, w_j) and assigns $(v, w_j).val$ a value which is greater than all the *pid*s of the paths passing through (v, w_i). Thus, all the interesting paths passing through v, w_j have higher pids than the uninteresting paths passing through (v, w_i).

Consider an uninteresting path p_1 passing through (v, w_j) and an interesting path passing through (v, w_i). Say $(v, w_j).val = \alpha$ and $(v, w_i).val = \beta$. If p_1 is unabsorbable, then $pid(p_1, v) > \alpha > pid(p_2, v)$. If p_1 is absorbable, then by Lemma 3 $pid(p_1, v) = pid(\mathit{suff}(p_1, w_j), w_j) < w_j.min$. Since $w_j.min \leq w_i.min$, $pid(p_1, v) < w_i.min$. Since by definition, $pid(p_2, v) \geq w_i.min$, therefore $pid(p_1, v) < pid(p_2, v)$. □

Theorem 1 proves the correctness of PSPP'. If local validity of the labeling L computed by PSPP' holds at the $entry$ vertex then validity of L also holds. The output of both PSPP' and PSPP are same by Lemma 1. We therefore have the following theorem.

THEOREM 2. *Given a DAG and a set of interesting edges, PSPP computes a valid labeling.*

Note that PSPP is same as the Ball-Larus algorithm if all paths in the DAG are marked as interesting and additionally, if all edges are interesting (even though some paths are uninteresting) the PSPP algorithm is same as the Ball-Larus algorithm. Finally, for two sets of paths S and S' where $S \subset S'$, the number of zero edges obtained by PSPP for S is more than or equals to that of S'.

7. REFERENCES

[1] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 35–42, New York, NY, USA, 2002. ACM.

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

[3] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16:1319–1360, July 1994.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[5] S. Baswana, S. Roy, and R. Chouhan. Pertinent path profiling: Tracking interactions among relevant statements. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[6] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society.

[7] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 2–9, New York, NY, USA, 2002. ACM.

[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[9] T. M. Chilimbi, A. V. Nori, and K. Vaswani. Quantifying the effectiveness of testing via efficient residual path profiling. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 545–548, New York, NY, USA, 2007. ACM.

[10] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 95–105, New York, NY, USA, 2002. ACM.

[11] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 373–390, New York, NY, USA, 2013. ACM.

[12] M. Diep, M. Cohen, and S. Elbaum. Probe distribution techniques to profile events in deployed software. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 331–342, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[17] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[18] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 239–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[20] B. Li, L. Wang, and H. Leung. Profiling selected paths with loops. *SCIENCE CHINA Information Sciences*, 57(7):1–15, 2014.

[21] D. Melski and T. W. Reps. Interprocedural path profiling. In *Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 47–62, London, UK, UK, 1999. Springer-Verlag.

[22] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 277–284, New York, NY, USA, 1999. ACM.

[23] D. Saha, P. Dhoolia, and G. Paul. Distributed program tracing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 180–190, New York, NY, USA, 2013. ACM.

[24] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: Compactly numbering interesting paths. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 351–362, New York, NY, USA, 2007. ACM.