Practical Path Profiling for Dynamic Optimizers*

Michael D. Bond Kathryn S. McKinley
Dept. of Computer Sciences, University of Texas at Austin
E-mail: {mikebond,mckinley}@cs.utexas.edu

Abstract

Modern processors are hungry for instructions. To satisfy them, compilers need to find and optimize execution paths across multiple basic blocks. Path profiles provide this context, but their high overhead has so far limited their use by dynamic compilers. We present new techniques for low overhead online practical path profiling *(PPP). Following targeted path profiling (TPP), PPP uses an edge profile to simplify path profile instrumentation (profile-guided profiling). PPP improves over prior work by (1) reducing the amount of profiling instrumentation on cold paths and paths that the edge profile predicts well and (2) reducing the cost of the remaining instrumentation.*

Experiments in an ahead-of-time compiler perform edge profile-guided inlining and unrolling prior to path profiling instrumentation. These transformations are faithful to staged optimization, and create longer, harder to predict paths. We introduce the branch-flow *metric to measure path flow as a function of branch decisions, rather than weighting all paths equally as in prior work. On SPEC2000, PPP maintains high accuracy and coverage, but has only 5% overhead on average (ranging from -3% to 13%), making it appealing for use by dynamic compilers.*

1. Introduction

To perform well, modern out-of-order processors need long sequences of predictable instructions. To provide these sequences, compilers must look beyond a single basic block to find, analyze, and optimize *hot* (frequently executed) paths. Prior work uses hot paths to drive hyperblock and superblock formation [7, 22, 24] and path-based analyses and optimizations [1, 18, 19, 33]. In an ahead-of-time compiler, the compiler collects a path profile and then uses it to optimize the program. The optimized program thus does not incur profiling overhead. In a dynamic compiler, execution time includes the compile-time overhead of adding path profiling instrumentation and the runtime overhead of executing the instrumentation, which has so far been prohibitively high for dynamic optimizers. This paper focuses on reducing runtime overhead with modest impact on compile-time overhead.

The runtime overhead of Ball-Larus path profiling (PP) is 31% on average, but as high as 97% [10]. Joshi et al. targeted path profiling (TPP) improves over PP with profile-guided profiling and has overhead of 16% on average, but as high as 53% [23]. This high overhead has driven dynamic compilers to use edge profiles [5], which are inexpensive to collect but are often poor predictors of a program's hot paths. We show here that edge profile accuracy at predicting hot paths is on average 73% and as low as 26% for SPEC2000.

This paper introduces *practical path profiling* (PPP), which is efficient enough to use in a dynamic setting, without much loss of accuracy or coverage. PPP builds on PP and TPP. It lowers overhead by (1) instrumenting even fewer paths than TPP, and (2) more efficiently profiling the paths it does instrument. For example, PPP uses a global criterion to exclude more cold paths than TPP; pushes instrumentation further than PP and TPP by ignoring cold edges; places less instrumentation on the hottest paths than PP and TPP; and eliminates TPP's poison check.

This paper also improves the methodology for evaluating path profiling by introducing a new metric and using a more realistic and challenging compilation setting. Previous work uses what we call the *unit-flow* metric to evaluate estimated path profiles [11, 23]. Unit flow weights all paths equally regardless of length, which leads to non-intuitive results and obscures how well profiling predicts longer paths. We introduce the *branch-flow* metric to measure prediction accuracy, which weights paths by their lengths.

To provide more realistic paths than prior profiling work, we first perform edge profile-guided inlining and unrolling. This step produces longer and more complex paths, making accurate path profiling harder. This environment is more faithful to staged optimization, which first finds hot calls and loops, inlines and unrolls, instruments paths, and then applies path optimizations.

We implement and evaluate PP, TPP, and PPP in

*This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

Scale [25], an ahead-of-time compiler, which provides a deterministic platform for the experiments but neglects compilation costs. Our results show that PPP is almost as accurate as TPP: PPP predicts 96% of hot path flow on average, and is within 1% of TPP. PPP performs better than TPP: TPP instrumentation adds 12% on average (from -4 to 50%) to program runtime, whereas PPP adds 5% on average (from -3 to 13%). Furthermore, we show that all of PPP's techniques contribute to its low overhead.

2. Related Work

This section first discusses how well edges predict paths. It then overviews related work for reducing the cost of collecting path profiles. Section 3 reviews the path profiling algorithms on which PPP builds.

If an edge profile accurately predicts the path profile, the optimizer can use it to estimate the path profile. This approach is inexpensive: Edge profiling has negligible overhead (0.5 to 3%) using sampling [2, 34] or hardware support [13, 14, 16, 20]. Ball et al. found that 80% of the paths could be attributed from an edge profile, but noted that the most complex paths are not predictable from an edge profile [11]. We repeat their analysis here with methodology that is more realistic for a dynamic setting and find that just 48% of paths can be attributed from an edge profile (Section 8.1).

One way to reduce path profiling overhead in a dynamic optimizer is to execute instrumented code only part of the time. Previous approaches use code sampling [6, 21] and dynamic instrumentation [26, 28, 31].[1] These approaches lower overhead at the cost of extending the time it takes to collect a given number of samples. PPP lowers overhead directly by making instrumentation less expensive and is thus orthogonal to these approaches. Furthermore, PPP demonstrates overhead comparable to that of code sampling frameworks alone.

Dynamo [7, 17] demonstrates dynamic optimization benefits for real programs already optimized by an ahead-of-time compiler. Dynamo selects likely hot paths using Next Executing Tail (NET) (previously called Most Recently Executed Tail), which collects a single path trace after a backward branch becomes hot. While NET is statistically likely to select the hottest path, it cannot distinguish between the cases of a few dominant hot paths and many "warm" paths. As a result, Dynamo is too aggressive in the latter case, causing it to thrash the code cache and bail out to native execution in some benchmarks. In contrast, PPP can distinguish these cases through wider coverage (Section 8.1).

Selective path profiling (SPP) [3] reduces instrumentation by profiling only a subset of a program's paths, called *paths of interest*. Like PPP, SPP assigns unique path numbers to the profiled paths and non-unique path numbers to the other paths. However, SPP assigns high path numbers to profiled paths and lower path numbers to other paths, while PPP does the opposite. As a result, PPP places less instrumentation on profiled paths, which are also the hot paths.

Another approach uses hardware to collect paths, such as the Pentium 4's path buffer [27], but aggregating path statistics from the buffer remains a challenge. Vaswani et al. present a hardware-based programmable path profiler that maintains a hot path table (HPT) in hardware [29]. Their approach has very low overhead (less than 1% on average), and its accuracy is high (above 90% on average) when the HPT is large enough.

3. Background

We first review Ball-Larus [10] and Joshi et al. [23], since PPP builds on them.

3.1. Ball-Larus Path Profiling (PP)

PP adds instrumentation to a program to count how many times it executes each acyclic, intraprocedural path (i.e., a loop back edge ends the current path and starts a new path; a routine call starts a new path, deferring the current path until the routine returns).[2]

PP first converts the routine's control flow graph (CFG) to a directed acyclic graph (DAG) by removing each back edge (loop tail $\rightarrow$ loop header) and adding two *dummy* edges: one from the routine entry to the loop header and one from the loop tail to the routine exit. Figures 1(a) and 1(b) illustrate this process of "breaking" a back edge. The algorithm in Figure 2 then assigns values to edges such that, for each of the N acyclic paths in the DAG, the sum of the edge values is a unique number in $[0, N-1]$. Figure 1(c) shows a labeled DAG where the sum of the edge values is a unique value in $[0, N-1]$, where $N = 8$.

PP then uses Ball's efficient event counting algorithm [8] to reassign values to DAG edges so each path computes the same path number as before, but more efficiently. It estimates edge frequencies with simple static heuristics (e.g., loops execute 10 times and branch directions are 50/50) to select a spanning tree containing the predicted high frequency edges. It reassigns zero to the spanning tree edges and nonzero values to the other edges, such that the path numbers remain the same. Figure 1(d) shows the DAG with new edge values after applying the event counting algorithm.

PP next places instrumentation on edges. A variable `r`, called the *path register*, computes the unique number for a path as the path executes. The instrumentation

1. initializes the path register to zero (`r=0`) on entry;

[1] Except for structural path profiling [31], which improves coverage of dynamic instrumentation for Ball-Larus path profiling, these approaches are not limited to intraprocedural, acyclic paths. For example, bursty tracing [21] can profile cyclic, interprocedural paths, albeit at a lower sampling rate.

[2] Young presents an alternative approach that ignores loop and routine boundaries but limits path length to a constant number of branches [32].

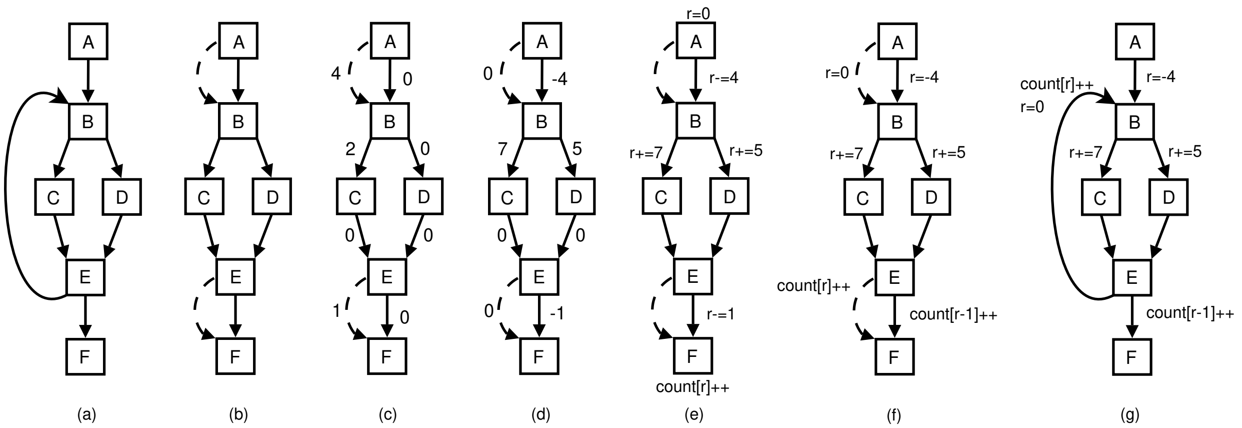

Figure 1. An example routine instrumented by PP. (a) Original routine; (b) after PP converts the CFG to a DAG; (c) after path numbering; (d) after edge value reassignment via the event counting algorithm; (e) after instrumentation placement; (f) after instrumentation pushing and combining; and (g) after conversion back to the CFG.

2. adds values to the path register (`r+=val`) on edges with nonzero values; and
3. updates the frequency for the taken path in a path frequency table (`count[r]++`) on exit.

Figure 1(e) shows the DAG with instrumentation that counts paths. To reduce overhead further, PP pushes instrumentation with the form `r=0` down paths until it reaches instrumentation with the form `r+=val`, and then combines them into `r=val`. Similarly, it pushes `count[r]++` up until it encounters `r+=val` (or even `r=val`) and then combines them into `count[r+val]++` (or `count[val]++`). Figure 1(f) shows the DAG after pushing path initialization instrumentation down and path counting instrumentation up. The final step of the PP algorithm converts the DAG back to a CFG. It removes dummy edges, restores back edges, and moves instrumentation from dummy edges to their corresponding back edges. Figure 1(g) shows the example converted back to a CFG.

Ball and Larus show that PP instrumentation adds runtime overhead of 31% on average to SPEC95, although it is as high as 73% for `perl` and 97% for `gcc`, which are considered most representative of modern programs. The next section describes how targeted path profiling (TPP) lowers PP overhead in a dynamic optimizer.

3.2. Joshi et al. Targeted Path Profiling (TPP)

TPP [23] uses an existing edge profile to simplify instrumentation and thus reduce two sources of overhead in PP: (1) hashing and (2) instrumentation of paths the edge profile predicts well.

```
foreach basic block v in reverse topological order
  if v is the exit block
    NumPaths(v) = 1
  else
    NumPaths(v) = 0
    foreach edge e = v→w in incr. order of NumPaths(w)
      Val(e) = NumPaths(v)
      NumPaths(v) = NumPaths(v) + NumPaths(w)
```

Figure 2. PP path numbering algorithm.

Cold Path Elimination PP uses hashing instead of an array to store path frequencies when the number of possible paths results in a too large array [10]. A large array wastes space and has poor caching and paging performance. Joshi et al. estimate hashing is about five times more expensive than an array [23]. To remove the need for a hash table, TPP eliminates cold paths by identifying and excluding cold edges from profiling. It uses a *local* criterion to identify cold edges: An edge is cold if the ratio of its frequency to its source basic block's frequency is below a threshold. Consider Figure 3(a), which has eight possible paths. After removing the cold edge, it has only four possible paths—the other paths each contain a cold edge. TPP assigns values to the remaining edges (Figure 3(b)) and instruments the routine (Figure 3(c)).

However, programs may still *execute* cold edges, computing incorrect or even invalid path numbers. TPP solves this problem by *poisoning* the path register on cold paths. Poisoning assigns the path register a large negative value. Instrumentation at the end of a path examines whether the path register is poisoned (`r<0`). If so, TPP increments a *cold counter*; otherwise, it counts a hot path (Figure 3(d)). Because poison checks add overhead, TPP only eliminates cold paths from routines that would require a hash table to count paths *without cold path removal*, but can use an array after cold path removal.

Obvious Path Identification TPP also reduces PP overhead by not instrumenting paths that the edge profile predicts well. For example, consider the routine in Figure 4. Each path has at least one edge, a *defining edge*, that is only on that path. Joshi et al. call such a path an *obvious path* because its frequency is equal to the frequency of the defining edge. TPP also identifies loops with obvious bodies (i.e., all paths in the loop body are obvious) and high average trip counts, and does

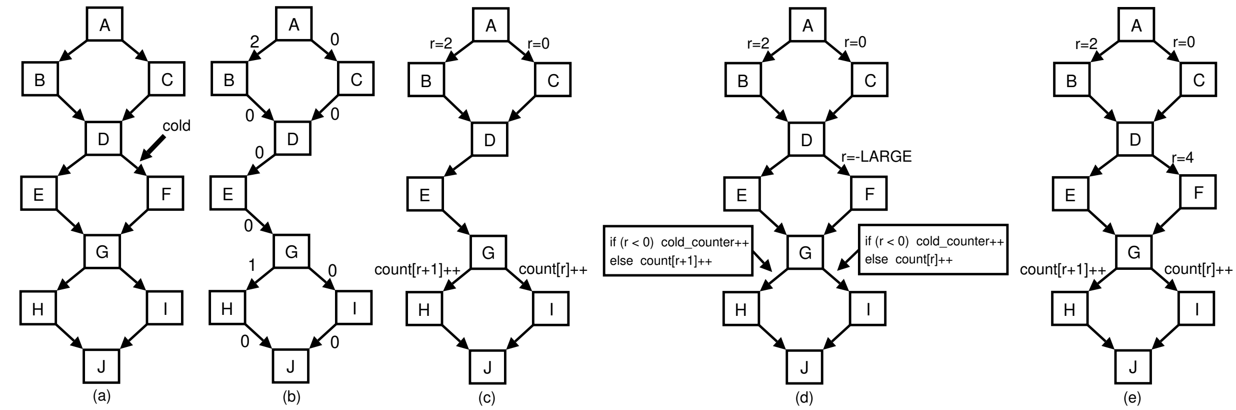

Figure 3. Cold path poisoning. (a) Original routine with a cold edge; (b) after cold edge removal and path numbering; (c) after instrumentation; (d) after poisoning the cold edge and adding poison tests (TPP only); (e) after poisoning the cold edge (PPP only).

not instrument these loops, effectively trading information about paths that enter or exit the loop for low overhead. Joshi et al. found that identifying obvious paths and loops *after* cold path removal significantly increases the number of obvious paths.

4. Practical Path Profiling

This section describes the six techniques that PPP uses in addition to TPP's techniques to reduce profiling overhead further. The first four techniques reduce the amount of instrumentation, while the other two reduce the cost of the remaining instrumentation.

4.1. Instrument Routines with Low Coverage

TPP identifies and does not instrument obvious paths for which the edge profile always provides perfect coverage of the path profile (Section 3.2). Coverage is the fraction of the path profile that the edge profile measures (Section 6.2). Many non-obvious paths still have high coverage from an edge profile, and profiling them costs as much as profiling paths with low coverage. PPP thus does not instrument routines with coverage exceeding a threshold.

4.2. Global Edge Criterion

TPP eliminates many cold paths by removing cold edges (Section 3.2). It marks an edge cold if its bias is above some threshold percentage. This *local* criterion trades loss of accuracy on cold paths for simpler instrumentation on hotter paths in the same routine. PPP adds a *global* criterion: An edge is cold if its frequency, as a percentage of *total program flow*, falls below a threshold. PPP marks an edge cold if either criterion applies.

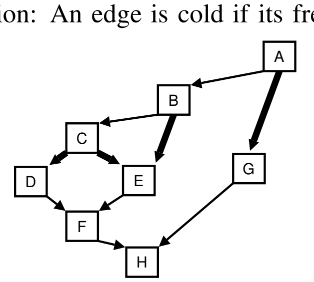

Figure 4. Example routine with all obvious paths. Each path has at least one defining edge (marked in bold).

4.3. Self-Adjusting Criterion

Even after eliminating cold and obvious paths, some routines have so many possible paths that they still require hashing, which is expensive (Section 3.2). To eliminate hashing, PPP increases the global cold edge threshold and re-executes path numbering, trading profile accuracy for lower overhead. PPP repeats this process until the number of possible paths falls below the hashing threshold. We find PPP rarely needs to adjust the global edge criterion. In our experiments, PPP adjusts it for only two routines, one in `vpr` and one in `mesa`, and it needs at most four iterations to drop below the hashing threshold.

4.4. Pushing Instrumentation Further

PP, TPP, and PPP assign instrumentation to edges and then push down path register initialization and push up path counting (Section 3.1). For correctness, they stop pushing along a path when the current edge and another merge at the same basic block. TPP stops pushing even if the other edge is cold. Unlike TPP, PPP ignores cold edges when pushing instrumentation and thus finds additional opportunities to combine instrumentation and detect obvious paths. Consider the example routine in Figure 5(a), where the edge MO is cold. After cold edge removal, TPP places instrumentation as shown in Figure 5(b). TPP does not push the counting instrumentation (`count[r]++`) above M because it has two outgoing edges. PPP instead ignores the cold edge MO, so M effectively has only one outgoing edge (MN), and PPP pushes instrumentation above block M as shown in Figure 5(c). As a result, PPP (1) removes instrumentation altogether from the obvious paths $AIJLMNO$ and $AIKLMNO$ and (2) combines the path counter increment on MN with the path register increment on EF.

This PPP optimization causes some cold paths to record hot paths. For example, in Figure 5 if the cold path $ABCEGHMO$ executes, the instrumenta-

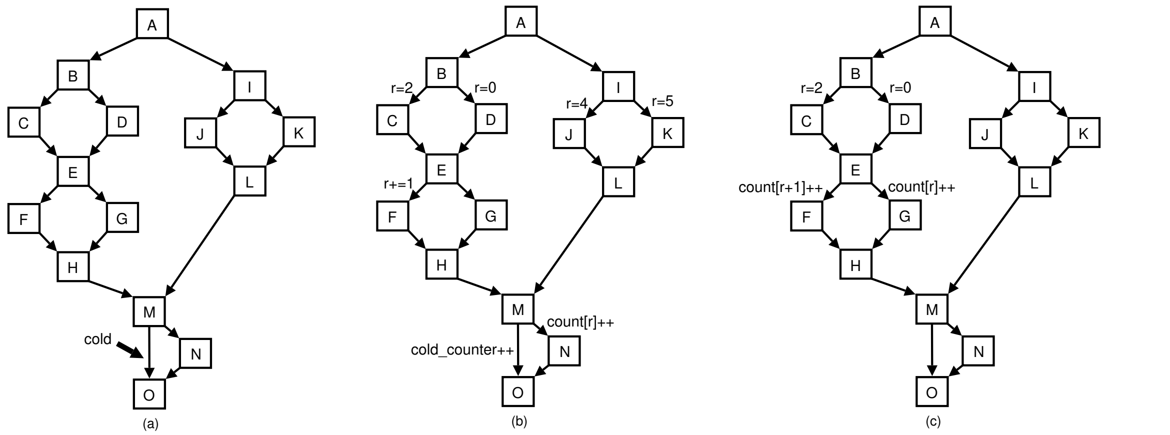

Figure 5. TPP and PPP push instrumentation. (a) Routine with cold edge; (b) TPP instrumentation; and (c) PPP instrumentation.

tion counts path number 2 ($ABCEGHMNO$). Since cold paths execute infrequently, the overcount tends to be low. Section 6.2 describes how to account for overcount when measuring the coverage of PPP.

4.5. Smart Path Numbering

PPP numbers paths and places instrumentation so the hottest edges have the lowest runtime overhead. It modifies two PP algorithms to use edge frequencies rather than static heuristics: path numbering and event counting (Section 3.1). PP path numbering numbers a block's outgoing edges in order of increasing number of possible paths in each edge's target's subgraph, which decreases the range of edge increments [10]. PPP instead numbers each edge in decreasing order of execution frequency (Figure 6). Thus, it assigns zero to the hottest outgoing edge, adding no instrumentation to the most frequently executed edge (unless it begins or ends a path). While the PP event counting algorithm uses static heuristics to build a spanning tree, the PPP version uses an edge profile. Because an edge profile is generally a better predictor of future edge frequencies than static heuristics, PPP event counting moves instrumentation from relatively hot edges to relatively cold edges more successfully than PP.

4.6. Free Poisoning

TPP *poisons* the path register with a large negative value on cold edges and checks for a poisoned path register at the end of a path (Section 3.2). This test adds overhead. PPP instead poisons cold edges so that each cold path computes a non-unique path number in approximately $[N, 2N-1]$, eliminating the need to check for a poisoned path. Conceptually, PPP maps cold paths to this range by setting the path register to N on cold edges; the increments on any remaining edges on the path add at most $N-1$ to the path register. In practice, event counting may assign negative values to edges, so PPP compensates. It first computes the range of possible incoming path register values for the cold edge's target block with a single reverse topological traversal. It then adds appropriate instrumentation on the cold edge to map the path register to a range that is at most $[N, 3N-1]$. Figure 3(e) shows how PPP instruments an example routine with free poisoning. The cold edge poisons the path register by setting it to 4. The four cold paths compute path numbers in $[4, 5]$ and thus do not interfere with the hot path numbers $[0, 3]$.

Because TPP adds checks for poisoned paths, Joshi et al. remove cold edges only when as a result, TPP will use an array instead of a hash table (Section 3.2). PPP removes cold edges from all routines since its poisoning trades extra space to eliminate the poison check.

foreach basic block v in reverse topological order
  if v is the exit block
    $\mathit{NumPaths}(v) = 1$
  else
    $\mathit{NumPaths}(v) = 0$
    foreach edge $e = v \rightarrow w$ <u>in decr. order of exec. freq.</u>
      $\mathit{Val}(e) = \mathit{NumPaths}(v)$
      $\mathit{NumPaths}(v) = \mathit{NumPaths}(v) + \mathit{NumPaths}(w)$

Figure 6. PPP path numbering. Changes from the original Ball-Larus algorithm (Figure 2) are underlined.

4.7. PPP Analysis Time

PPP trades lower profiling overhead for additional compile-time analysis. PPP's techniques are linear, except for identifying routines with high edge profile coverage (Section 4.1), which takes $O(|V| \times \mathit{width}(G))$ time in the worst case [11]. $|V|$ is the number of basic blocks, and $\mathit{width}(G)$ is the number of edges in a maximal cut of DAG G. In practice, this computation is linear. However, Section 8.3 shows this technique adds little improvement to the others in PPP, so a dynamic optimizer could omit it to guarantee a linear-time analysis. We do not present quantitative results for PP, TPP, and PPP analysis time because none of these implementations are designed for efficiency.

5. Constructing Estimated Path Profiles

So far we have discussed how PPP instruments programs. This instrumentation produces an estimated path profile for P_{instr}, the set of paths instrumented by PPP.

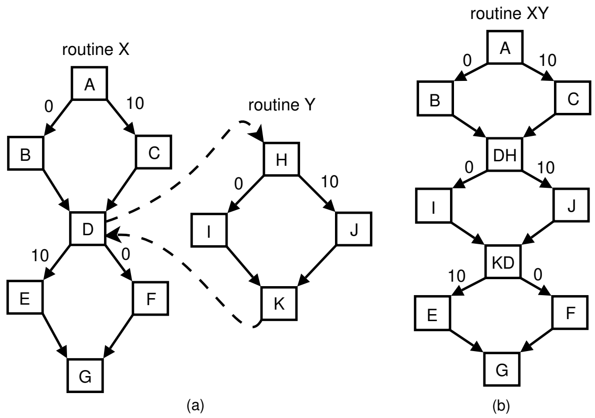

Figure 7. Example edge profile that motivates branch flow. (a) Routine `X` calls `Y` and (b) `Y` inlined into `X`.

The set of all paths P equals $P_{instr} \cup P_{uninstr}$. PPP estimates the profile for $P_{uninstr}$ from the *definite flow* profile. This section explains definite flow and how PPP computes the definite flow profile. It first defines *flow* and how we measure it.

5.1. Flow, Unit Flow, and Branch Flow

Flow is a measure of the amount of execution on a path or paths. Prior work uses the *unit-flow* metric, which defines flow along a path p as its execution frequency:

$$F(p) = \mathit{freq}(p)$$

The flow on a set of paths P is the sum of the paths' flows:

$$F(P) = \sum_{p \in P} \mathit{freq}(p)$$

Unit flow weights all paths equally, regardless of path length. This metric produces non-intuitive flows. For example, consider the edge profile for routines `X` and `Y` in Figure 7(a). The path $ACDEG$ has flow 10, and the path HJK also has flow 10, since the call to `Y` starts a new path (Section 3.1). The total unit flow through `X` and `Y` is thus 10 + 10 = 20. However, if the compiler inlines `Y` into `X` (Figure 7(b)), then the path $ACDHJKDEG$ has flow 10, and total unit flow is only 10.

We introduce the *branch-flow* metric, which instead measures flow as a function of the number of branches b_p in a path:

$$F(p) = \mathit{freq}(p) \times b_p$$

The flow on a set of paths P is again the sum of the flows:

$$F(P) = \sum_{p \in P} (\mathit{freq}(p) \times b_p)$$

We define a branch as an edge whose source block has at least one other outgoing edge. If we measure flow using branch flow, then in Figure 7(a), path $ACDEG$ has flow 20 because it has two branches and frequency 10, and path HJK has flow 10 because it has one branch and frequency 10. Total flow is 20 + 10 = 30. In Figure 7(b), the path $ACDHJKDEG$ has flow 30 because it has three branches and frequency 10. Total flow is thus 30 in both the original and inlined code.

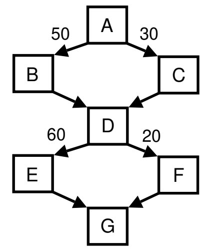

Figure 8. Example edge profile.

5.2. Definite Flow

Ball et al. introduce *definite* flow, which is the minimum flow that an edge profile guarantees [11]. For example, consider the edge profile in Figure 8. The routine's total actual flow *using the branch-flow metric* is 50 + 30 + 60 + 20 = 160 (i.e., the sum of branch edge frequencies). The actual flow on any of the four paths is unknowable from the edge profile. We can, however, compute the minimum flow of each path. For example, consider path $ABDEG$. The other three paths may have flow at most 100 (if path $ACDEG$ has frequency 30 and thus flow 60, path $ABDFG$ has flow 40, and path $ACDFG$ has flow 0). Thus the minimum, or definite, flow of path $ABDEG$ is 160 - 100 = 60. By similar reasoning, the definite flows of paths $ACDEG$, $ABDFG$, and $ACDFG$ are 20, 0, and 0, respectively. The routine's definite flow is thus 60 + 20 + 0 + 0 = 80.

PPP constructs the definite flow path profile using Ball et al.'s algorithm [11], but modified to use the branch-flow metric. The appendix presents our modified algorithms for definite flow, potential flow [11], and reconstructing paths from definite and potential flow. The reconstruction algorithm also includes a minor fix to the original not related to branch flow.

6. Evaluating an Estimated Path Profile

This section describes *accuracy* and *coverage*, two measures of how closely an estimated path profile imitates the actual path profile.

6.1. Accuracy of an Estimated Path Profile

We define *accuracy* as the ability of an estimated path profile to predict the hot paths in a program. To measure accuracy, we use Wall's weight matching scheme [30], following Ball et al. [11]. The scheme first identifies a program's actual hot paths, H_{actual}, from the path profile collected by PP. A path is hot if its flow is above a threshold percentage of total program flow. The scheme then constructs the set of estimated hot paths, $H_{estimated}$, by selecting the $|H_{actual}|$ hottest paths from the estimated path profile. Accuracy is the fraction of actual hot path flow that the estimated path profile predicts correctly:

$$\mathit{Accuracy} = \frac{F(H_{estimated} \cap H_{actual})}{F(H_{actual})}$$

For edge profiling, we select $H_{estimated}$ from a *potential* flow profile because Ball et al. found it predicts hot paths better than definite flow and a greedy algorithm [11].

For PPP, we select $H_{estimated}$ from the estimated path profile PPP constructs (Section 5). An exception is when PPP does not add any instrumentation to a program (`swim` and `mgrid` in our experiments). In this case, we select $H_{estimated}$ from a potential flow profile in order to match edge profile accuracy.

6.2. Coverage of an Estimated Path Profile

We define *coverage* as the fraction of actual program flow that a profiling method (e.g., edge profiling, TPP, or PPP) definitely measures. The coverage of an edge profile is the ratio of definite flow to actual flow:

$$Coverage = \frac{DF(P)}{F(P)}$$

Ball et al. call this ratio *attribution of definite flow* [11]. In Figure 8, the coverage of the edge profile is 80 / 160 = 50% (Section 5.2 has the intermediate computations). Intuitively, 50% of the routine's flow can be attributed from the edge profile. The other 50% cannot be definitely attributed to any path.

PPP's estimated path profile is a combination of *measured* flow (*MF*) for P_{instr} and *computed* definite flow (*DF*) for $P_{uninstr}$ (Section 5). PPP may overcount some paths in P_{instr} (Section 4.4), making $MF(P_{instr})$ an overestimate. Thus, we use *actual* flow $F(P_{instr})$ when computing coverage, and subtract out the overcounted flow $F_{overcount} = MF(P_{instr}) - F(P_{instr})$ as a penalty:

$$Coverage = \frac{F(P_{instr}) + DF(P_{uninstr}) - F_{overcount}}{F(P)}$$

7. Methodology

This section describes our compiler framework, architecture platform, the path characteristics of our benchmarks, and our path profiling implementations.

7.1. Compiler and Platform

We implement path profiling in Scale, a retargetable ahead-of-time optimizing compiler for C and Fortran [25]. Scale uses static single assignment and performs many classic compiler optimizations (e.g., constant propagation, value numbering, register allocation, and alias analysis). Although it is a research compiler, it achieves competitive performance for the Alpha and SPARC architectures. While this evaluation differs from a dynamic optimization system evaluation, it eliminates non-determinism, yielding stable and understandable results. Scale generates optimized Alpha binaries, which we execute on an AlphaServer 4100 with four 21164 processors running at 600 MHz, each with 8 KB direct-mapped L1 split caches, 96 KB shared on-chip secondary cache, 8 MB off-chip secondary cache, and 2 GB of memory running OSF1.

7.2. Benchmarks

We use the C and Fortran 77 SPEC2000 benchmarks. We omit (1) `gzip` and `vortex` because Scale would not generate correct binaries for these benchmarks and (2) `gcc` because our (admittedly space inefficient) code for computing profiling accuracy (Section 6.1) runs out of memory. We use the `ref` inputs to collect profiles and to measure profiling overhead. Because we are simulating a dynamic optimizer, which uses information about execution behavior from the same run, this *self* advice is a realistic choice. For benchmarks with a `ref` input with multiple runs, we combine profiles from all runs, and report combined runtimes. Scale generates an executable for `perlbmk` that will not execute `ref` inputs 1, 4, 5, 6, and 7 correctly, so we use 2 and 3 only.

7.3. More Realistic Paths

We first perform edge profile-guided inlining and unrolling to approximate optimized code in a staged dynamic optimizer. This section shows the runtime and path length effects of profile-guided inlining and unrolling. These optimizations provide only modest execution time improvements. However, they increase the average number of branches and instructions in a path, sometimes significantly. We use these versions for the remaining experiments because they provide a realistic and challenging setting for path profiling. Previous work does not include these optimizations.

Table 1 compares the dynamic paths of SPEC2000 benchmarks with and without the effects of inlining and unrolling. For the *original code*, we perform standard scalar optimizations without inlining or unrolling. We perform the same optimizations on *inlined and unrolled code*. Table 1 shows the number of dynamic paths; average number of branches and instructions (statements in Scale's low-level internal representation (IR)) per path for the original and expanded code; percentage of dynamic calls inlined; unroll factor (averaged over dynamic loop executions); and speedup (or slowdown) from inlining and unrolling.

Profile-guided inlining uses a cost-benefit analysis similar to Arnold et al. [4]. The inliner assigns each call site a priority based on expected benefit (i.e., hotness of the call site) and cost (i.e., size of the callee). Scale inlines call sites in order of decreasing priority until total program size increases by an amount called *code bloat*. It does not inline large callees (more than 200 IR statements). We use a code bloat of 5% following prior work [4]. This modest code bloat balances compilation time with runtime gains for their dynamic optimizer.

Our experiments with larger code bloat increase the percentage of inlined dynamic calls by more than 1% in `parser` and `bzip2` only. Increasing the code bloat does not induce more inlining in other benchmarks because either (1) there are no additional calls with nonzero execution frequency; (2) the candidate callees are too large; (3) the inliner cannot safely perform inlin-

	Original code			Inlined and unrolled code					
Benchmark	Dyn. paths (in billions)	Avg. branches	Avg. instrs.	Dyn. paths (in billions)	Avg. branches	Avg. instrs.	% calls inlined	Avg. unroll factor	Speedup
vpr	6.2	2.73	17.88	3.8	4.19	24.99	71%	1.65	0.97
mcf	6.1	1.80	8.92	4.0	2.70	16.41	98%	1.00	1.01
crafty*	3.7	3.50	14.74	3.7	3.50	14.74	0%	1.00	1.00
parser	19.4	2.45	11.77	16.0	2.91	13.76	29%	1.46	1.03
perlbmk*	1.0	2.64	13.82	0.9	2.75	14.28	14%	1.00	1.02
gap	8.8	2.97	14.94	7.9	3.26	16.33	59%	1.22	1.02
bzip2	19.3	1.88	18.98	13.4	2.43	24.46	49%	1.99	1.07
twolf	16.4	1.86	13.93	11.2	2.40	17.30	23%	2.19	0.96
INT Avg	10.1	2.48	14.37	7.6	3.02	17.78	43%	1.44	1.01
wupwise	14.0	2.06	14.34	10.0	2.72	17.40	0%	1.90	0.98
swim	5.7	1.00	35.47	1.4	1.01	82.60	0%	4.00	1.02
mgrid	10.4	1.03	13.54	2.6	1.23	57.12	10%	4.00	0.96
applu	16.8	1.47	17.90	7.6	1.75	25.61	0%	1.31	1.14
mesa*	5.3	2.62	21.08	4.2	3.09	24.99	0%	2.31	1.00
art	11.5	1.67	11.70	3.4	3.67	21.40	100%	4.00	1.06
equake	3.7	1.20	16.53	1.0	2.37	41.54	100%	2.97	1.03
ammp	13.9	2.12	16.23	11.6	2.36	18.11	98%	1.81	1.02
sixtrack	16.7	1.19	25.62	5.1	2.12	62.48	57%	3.35	1.29
apsi	55.9	0.44	8.63	5.4	2.04	64.91	100%	3.90	1.02
FP Avg	15.4	1.48	18.10	5.2	2.24	41.62	46%	2.96	1.05
Overall Avg	13.0	1.92	16.44	6.3	2.58	31.02	45%	2.28	1.03

Table 1. Dynamic path characteristics with and without inlining and unrolling. *No cross-module inlining.

ing (e.g., because of aliased reference variables in Fortran); or (4) cross-module inlining (i.e., the caller and callee are in different source files) is disabled, which is the case for `crafty`, `perlbmk`, and `mesa` due to limitations in Scale.

Scale unrolls hot inner loops by a factor of four. The Alpha compiler [12] and Jikes RVM [5] use the same factor. If a loop has a low average trip count (less than eight), or unrolling would make the loop larger than 256 IR statements, Scale unrolls less or not at all. Scale does not unroll most `while` loops, so unrolling applicability is limited in the integer C programs.

Inlining and unrolling improve performance very little. Our performance numbers are comparable to profile-guided inlining and unrolling results for the Alpha compiler [12] on SPEC95. These transformations do however increase the average numbers of instructions and branches in dynamic paths, and thus provide a challenging setting for path profiling.

7.4. Path Profiling Implementations

We implement PP, TPP, and PPP. The profilers use 64-bit rather than 32-bit path numbers as in previous work, increasing the maximum number of possible paths from 2^{31} to 2^{63}, making path truncation rare [10]. We also use 64-bit path counters because some paths in SPEC2000 execute more than 2^{31} times. In routines with more than 4000 possible paths, the profiler uses a hash table with 701 slots and three tries of secondary hashing [15]. If a path still conflicts, the instrumentation increments a “lost path” counter. Less than 0.1% of all dynamic paths are lost except in `crafty`, which loses 7% of flow.

Our TPP implementation is faithful to Joshi et al. [23] with a few changes. (1) We could not easily reproduce TPP’s efficient poisoning, which uses inline assembly and a conditional move. We instead use PPP’s free poisoning (Section 4.6). (2) The original TPP truncates paths at disconnected loop entrances and exits. PPP instead marks such entrances and exits as cold. (3) Joshi et al. use edge frequencies to estimate frequencies of uninstrumented paths. We use definite flow to estimate paths TPP does not instrument. The former is an overcount, while the latter is an undercount. These changes should not affect accuracy and coverage much, and should slightly improve performance.

We use the following parameters for TPP and PPP. The TPP parameters are the same as Joshi et al. [23].

- TPP/PPP marks an edge cold if its frequency is less than 5% of the frequency of its source (Section 3.2).
- PPP marks an edge cold if its frequency is less than 0.1% of total program flow in terms of unit flow (Section 4.2).
- TPP/PPP disconnects obvious loops that have average trip counts of at least 10 (Section 3.2).
- PPP only instruments routines with less than 75% coverage from an edge profile (Section 4.1).
- PPP increases the global edge criterion by 50% until the number of paths drops below the hashing threshold (Section 4.3).

8. Results

This section first characterizes hot paths and how well PPP finds them compared with edge profiling and TPP. It also compares the overheads of PP, TPP, and PPP. PPP delivers much higher accuracy and coverage than an edge profile, although it has slightly less accuracy

Benchmark	Distinct paths	No. hot paths and % program flow ≥0.125% flow		≥1% flow	
vpr	3395	89	85.6%	24	66.1%
mcf	279	39	98.0%	16	90.9%
crafty	4559	133	72.3%	16	37.4%
parser	5627	133	76.6%	19	37.2%
perlbmk	2276	127	86.0%	21	54.0%
gap	3972	79	88.3%	19	67.5%
bzip2	2124	106	91.0%	18	61.7%
twolf	2039	100	95.3%	32	66.7%
INT Avg			86.6%		60.2%
wupwise	128	46	99.5%	26	92.5%
swim	75	16	99.8%	4	97.4%
mgrid	224	41	98.7%	17	85.8%
applu	242	44	97.9%	30	90.5%
mesa	412	60	95.3%	22	79.0%
art	463	43	97.7%	21	88.0%
equake	169	20	97.9%	12	96.2%
ammp	603	35	96.6%	12	89.7%
sixtrack	948	33	98.1%	11	89.5%
apsi	576	135	94.8%	28	43.9%
FP Avg			97.6%		85.2%
Overall Avg			92.7%		74.1%

Table 2. Hot paths in SPEC2000. Divided into all distinct paths, hot paths and their percent total program flow (hot is defined as 0.125% or 1% total program flow).

and coverage than TPP. However, PPP is significantly cheaper than TPP (one-third the overhead on the integer benchmarks).

8.1. Accuracy and Coverage

Table 2 shows the number of distinct paths the program takes at runtime, how many of these paths are hot, and the fraction of total program flow that hot paths represent. We use the same thresholds as Ball et al., 0.125% and 1% [11]. If we consider only hot paths with at least 1% of total program flow, the hot path flow is just 37% in two cases. We use a threshold of 0.125% in the rest of the evaluation because it includes hot paths that represent more program flow, but still effectively winnows the hot path candidates.

For these hot paths, Figure 9 presents the accuracy of edge profiling, TPP, and PPP, i.e., the fraction of hot path flow that each profiling method predicts (Section 6.1). Clearly, edge profiles are unable to predict hot paths well. PPP predicts 96% of hot path flow on average, and is within 1% of TPP. We believe PPP's accuracy, which never falls below 90%, is acceptable for many optimizations, but some very aggressive optimizations may require essentially perfect accuracy.

Figure 10 plots the coverage, i.e., the portion of the actual path profile definitely measured (Section 6.2). Edge profiling has poor coverage; it captures only about half of the actual path profile. Compared with PPP, TPP has consistently better coverage on the integer benchmarks because it is less aggressive at eliminating paths. However, as Figure 9 shows, PPP sacrifices some coverage but without a corresponding drop in accuracy.

Figure 11 plots the fraction of dynamic paths instrumented. Although TPP and PPP instrument only about half of all dynamic paths on average, they are still able to predict hot paths well (Figure 9).

8.2. Overhead

Figure 12 compares the overheads of PP, TPP, and PPP. Edge profiling has negligible overhead (Section 2). PP increases program execution time substantially in a number of cases, ranging from 39% up to 119% for eight benchmarks. TPP substantially reduces path profiling overhead, on average by 67%, although overhead is still sometimes high, ranging from 22% to 50% for four benchmarks. PPP reduces path profiling overhead further to 5% on average, compared with 12% for TPP. PPP overhead grows beyond 10% only in `crafty`, `twolf`, and `wupwise`. These benchmarks are among those for which the edge profile has the worst coverage (Figure 10). Further reductions will be difficult to achieve.

TPP reduces PP's overhead by 86% on the floating-point (FP) benchmarks, which tend to have less path complexity. TPP eliminates hashing from all FP benchmarks except `mesa`, and it instruments just 55% of dynamic paths on average (Figure 11). PPP offers only modest improvements over TPP (37% on average) on FP programs.

TPP eliminates hashing from just four of the eight integer programs, leaving room for PPP to improve (Figure 11). TPP reduces PP's overhead by 45%, but PPP's techniques reduce overhead by 67% over TPP (82% over PP).

Although PPP's instrumentation is at least as simple as TPP's, PPP occasionally has greater overhead than TPP (e.g., `mcf`). And although PPP can only add to execution time, PPP has a few negative overheads (e.g., `sixtrack`). These reproducible anomalies must be due to architectural sensitivities to factors such as code and data placement in the caches.

These results show that the techniques in PPP reduce the overhead of profile-guided profiling to levels acceptable for use in staged optimization.

8.3. Effects of Individual Techniques

Figure 13 shows how PPP's six techniques perform individually (Section 4). We present results for the benchmarks for which PPP improves performance significantly: more than 5% over TPP. We use a *leave-one-out* methodology, evaluating each technique by turning it off. Because the global edge criterion and self-adjusting are dependent, we evaluate them as one technique.

Each technique is essential to attaining the best performance on one or more benchmarks. The most important techniques are free cold path poisoning (FP), the self-adjusting global edge criterion (SAC), and to a lesser extent, aggressive instrumentation pushing (Push). Removing a technique sometimes reduces overhead (e.g., SPN for `vpr`). These results are another performance anomaly. SPN permutes the mapping of paths

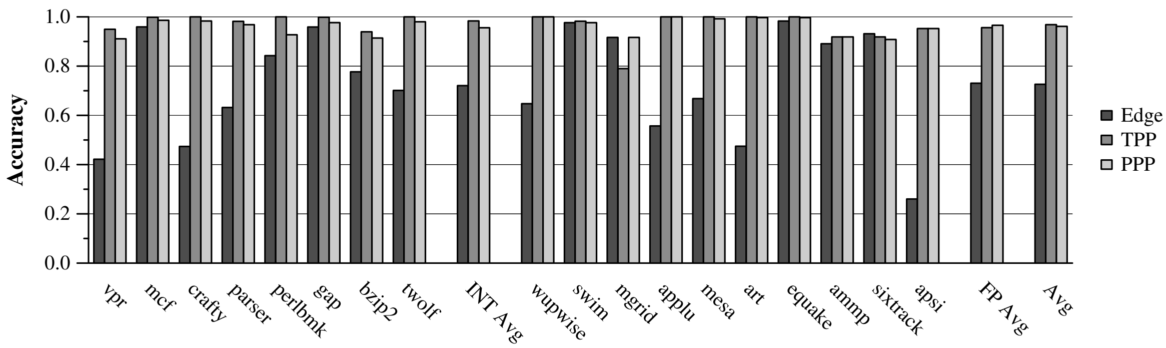

Figure 9. The accuracy (fraction of hot path flow predicted) of edge profiling, TPP, and PPP.

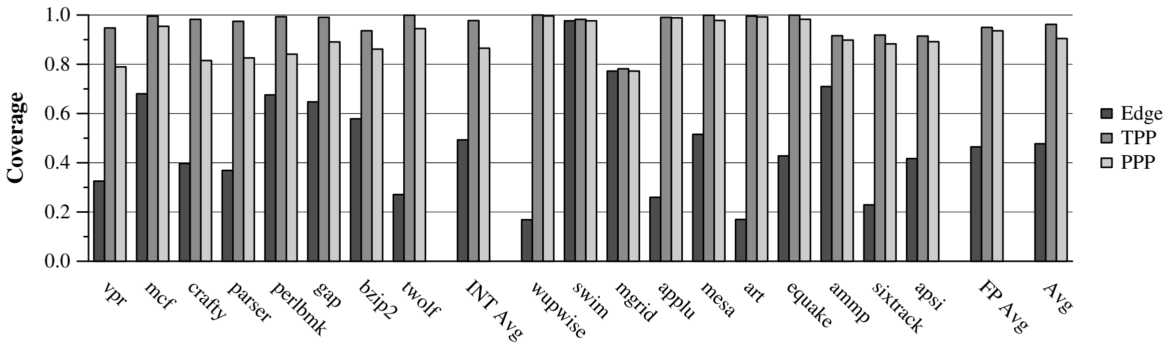

Figure 10. The coverage (fraction of actual path profile measured) of edge profiling, TPP, and PPP.

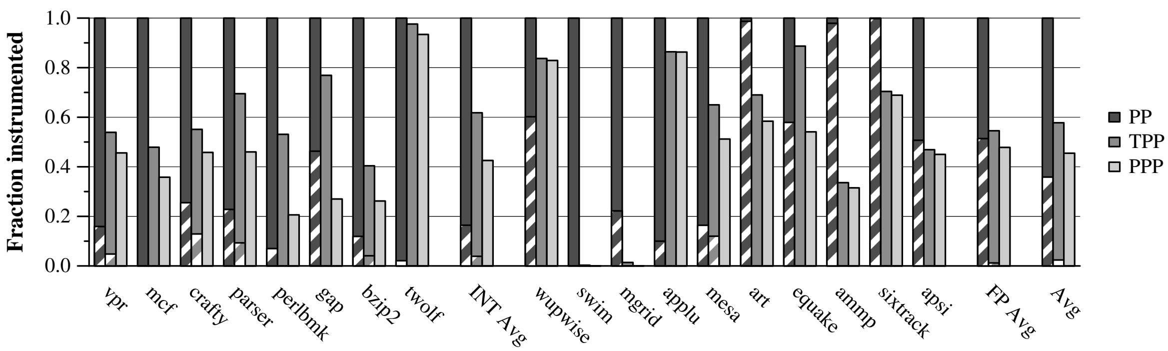

Figure 11. The fraction of dynamic paths instrumented by PP, TPP, and PPP. The stripes represent hashed paths.

to path numbers, which changes data cache accesses significantly. These effects seem to outweigh the benefits of SPN (i.e., fewer increments on hot edges). Removing SPN reduces overhead for four benchmarks and increases overhead for four.

The leave-one-out methodology suggests that instrumenting routines with low coverage only (LC) and SPN are not very beneficial. If we instead use a *one-at-a-time* methodology (results omitted for lack of space), we find LC and SPN are beneficial, lowering TPP's overhead by 27% and 16%, respectively, for the benchmarks in Figure 13.

9. Summary

This paper demonstrates a series of optimizations to prior path profiling work. It also introduces the branch-flow metric that fairly evaluates path profiling accuracy. The appendix corrects a small algorithmic error in prior work. The evaluation shows PPP adds 5% average program runtime overhead, yet still identifies the hot paths. This result makes it feasible for future staged dynamic compilation systems to collect path profiles continuously and use them to drive path-based optimizations.

10. Acknowledgements

We thank James Burrill for his generous help with Scale. We thank Steve Blackburn, Samuel Guyer, Xianglong Huang, Maria Jump, and the anonymous reviewers for their helpful suggestions.

References

[1] G. Ammons and J. R. Larus. Improving Data-flow Analysis with Path Profiles. In *Conference on Programming Language Design and Implementation*, pp 72–84, 1998.

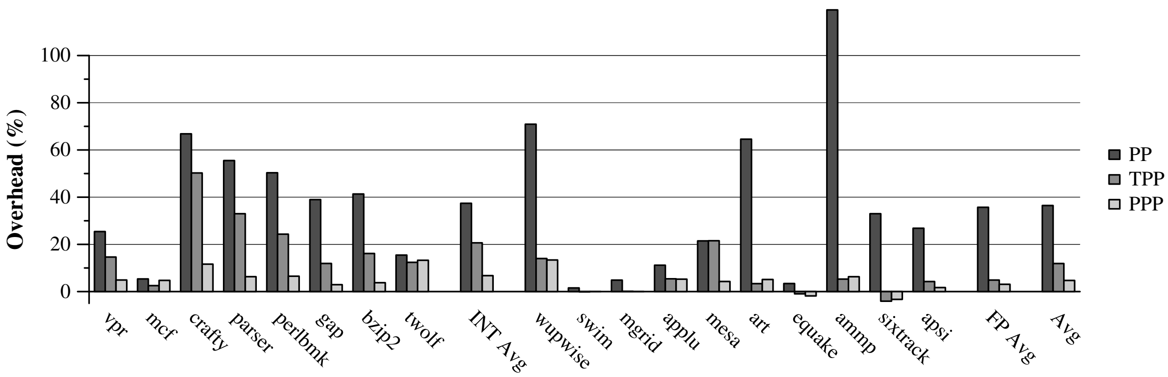

Figure 12. The overheads of three types of path profiling.

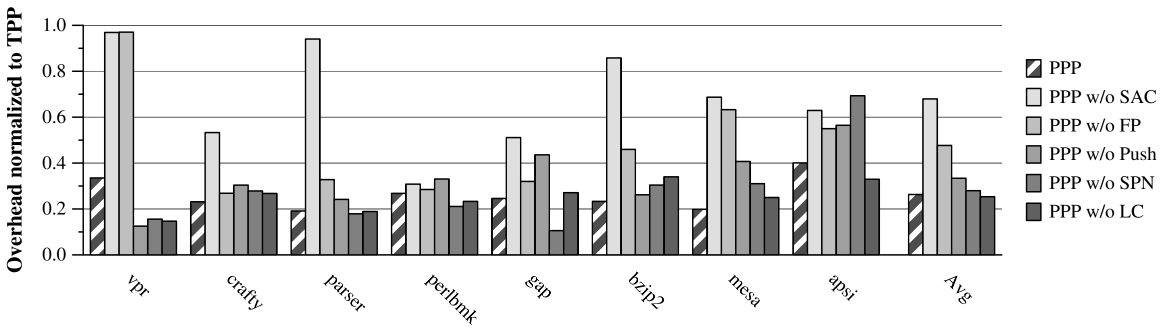

Figure 13. PPP leave-one-out methodology normalized to TPP. SAC = Self-adjusting cold edge criterion; FP = Free cold path poisoning; Push = Pushing instrumentation further; SPN = Smart path numbering; LC = Instrument routines with low coverage only.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Symposium on Operating Systems Principles*, pp 1–14, 1997.

[3] T. Apiwattanapong and M. J. Harrold. Selective Path Profiling. In *Workshop on Program Analysis for Software Tools and Engineering*, pp 35–42, 2002.

[4] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Workshop on Dynamic and Adaptive Compilation and Optimization*, pp 52–64, 2000.

[5] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp 47–65, 2000.

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

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[8] T. Ball. Efficiently Counting Program Events with Support for On-line Queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, 1994.

[9] T. Ball. Personal communication, September 2004.

[10] T. Ball and J. R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, pp 46–57, 1996.

[11] T. Ball, P. Mataga, and M. Sagiv. Edge Profiling versus Path Profiling: The Showdown. In *Symposium on Principles of Programming Languages*, pp 134–148, 1998.

[12] R. Cohn and P. Lowney. Feedback Directed Optimization in Compaq's Compilation Tools for Alpha. In *Workshop on Feedback Directed Optimization*, pp 3–12, 1999.

[13] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer. In *International Symposium on Microarchitecture*, pp 36–45, 1996.

[14] T. M. Conte, B. A. Patel, and J. S. Cox. Using Branch Handling Hardware to Support Profile-Driven Optimization. In *International Symposium on Microarchitecture*, pp 12–21, 1994.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[16] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *International Symposium on Microarchitecture*, pp 292–302, 1997.

[17] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 202–211, 2000.

[18] R. Gupta, D. A. Berson, and J. Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *International Conference on Computer Languages*, pp 230–239, 1998.

[19] R. Gupta, E. Mehofer, and Y. Zhang. Profile Guided Compiler Optimizations. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[20] T. H. Heil and J. E. Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *International Symposium on Microarchitecture*, pp 281–290, 2000.

[21] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *Workshop on Feedback-Directed and Dynamic Optimization*, pp 117–126, 2001.

[22] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, pp 229–248, 1993.

[23] R. Joshi, M. D. Bond, and C. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *International Symposium on Code Generation and Optimization*, pp 239–250, 2004.

[24] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *International Symposium on Microarchitecture*, pp 45–54, 1992.

[25] 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.

[26] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, 1995.

[27] B. Sprunt. Pentium 4 Performance Monitoring Features. *IEEE Micro*, 22(4):72–82, 2002.

[28] O. Traub, S. Schechter, and M. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Harvard University, 2000.

[29] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A Programmable Hardware Path Profiler. In *International Symposium on Code Generation and Optimization*, 2005.

[30] D. W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Conference on Programming Language Design and Implementation*, pp 59–70, 1991.

[31] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers. *The Journal of Instruction-Level Parallelism*, 6:1–24, 2004.

[32] C. Young. *Path-based Compilation*. PhD thesis, Harvard University, 1998.

[33] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *International Symposium on Microarchitecture*, pp 115–123, 1998.

[34] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen, and M. D. Smith. System Support for Automated Profiling and Optimization. In *Symposium on Operating System Principles*, pp 15–26, 1997.

Appendix

Figures 14 and 15 show the algorithms we use to efficiently compute definite and potential flow profiles from an edge profile. These dynamic programming algorithms follow Ball et al. [11], but use branch flow instead of unit flow (Section 5.1). Figure 16 shows the algorithm for efficiently selecting the hottest paths from a definite flow profile. This algorithm follows Ball et al. [11], but uses the branch-flow metric and corrects a minor error that we confirmed with Ball [9]. To select the hottest paths from a *potential* flow profile, we make two changes to the algorithm: (1) $\underline{f + \mathit{freq}(\mathit{tgt}(e)) - \mathit{freq}(e)}$ to $\underline{g}$ and (2) $\underline{g = f}$ to $\underline{g \text{ is minimal}, g \geq f}$. The algorithms use flow values $[(f, b) \longmapsto \Delta]$, where f is frequency, b is the number of branches on the path(s), and Δ is the number of paths with frequency f and b branches. The $\uplus$ operator combines flow values with the same f and b.

$M_{\hat{D}}[\mathit{exit}] := [(F, 0) \longmapsto 1]$
for $v \in V - \{\mathit{exit}\}$ in reverse topological order **do**
  for $e \in \mathit{out}(v)$ **do**
    $f_s := \mathit{freq}(\mathit{tgt}(e)) - \mathit{freq}(e)$
    $M_{\hat{D}}[e] := \biguplus_{((f,b)\longmapsto\Delta)\in M_{\hat{D}}[\mathit{tgt}(e)]}$
      $(f > f_s \ ? \ [(f - f_s, b) \longmapsto \Delta] : [])$
  $M_{\hat{D}}[v] := \biguplus_{e\in \mathit{out}(v)}(e$ is branch edge ?
    $[((f, b+1) \longmapsto \Delta) \mid ((f, b) \longmapsto \Delta) \in M_{\hat{D}}[e]] : M_{\hat{D}}[e]$

Figure 14. An algorithm for computing definite flow using the branch-flow metric.

$M_{\hat{P}}[\mathit{exit}] := [(F, 0) \longmapsto 1]$
for $v \in V - \{\mathit{exit}\}$ in reverse topological order **do**
  for $e \in \mathit{out}(v)$ **do**
    $M_{\hat{P}}[e] := \biguplus_{((f,b)\longmapsto\Delta)\in M_{\hat{P}}[\mathit{tgt}(e)]}$
      $[\mathit{min}(f, \mathit{freq}(e)) \longmapsto \Delta]$
  $M_{\hat{P}}[v] := \biguplus_{e\in \mathit{out}(v)}(e$ is branch edge ?
    $[((f, b+1) \longmapsto \Delta) \mid ((f, b) \longmapsto \Delta) \in M_{\hat{P}}[e]] : M_{\hat{P}}[e]$

Figure 15. An algorithm for computing potential flow using the branch-flow metric.

procedure main(M : map; *cutoff* : int)
  var *Paths* $:= \phi$
  for each $((f, b) \longmapsto \Delta) \in M[\mathit{entry}]$ s.t. $f{\cdot}b \geq \mathit{cutoff}$,
    in decreasing order of $f{\cdot}b$ **do**
      enumerate(*entry*, [], f, b, $\underline{f}$, Δ)
  return *Paths*

procedure enumerate(v : vertex; p : path; f, b, $\underline{f'}$, Δ : int)
  var $\Delta' := \Delta$
    used $:= \phi$
  if $v = \mathit{exit}$ **then**
    Paths $:= \mathit{Paths} \cup \{(p, \underline{f'}, b)\}$
  else
    while $\Delta' > 0$ **do**
      let $e \in \mathit{out}(v)$ and $((g, c) \longmapsto \Delta_g) \in M[e]$ s.t.
          $\underline{g = f}$, $b = c$, and $(e, g, c) \notin \mathit{used}$
        $\mathit{debit} = \mathit{min}(\Delta', \Delta_g)$
      in
        enumerate($\mathit{tgt}(e)$, append(p, e),
          $\underline{f + \mathit{freq}(\mathit{tgt}(e)) - \mathit{freq}(e)}$,
          (e is branch edge ? $b - 1 : b$), $\underline{f'}$, *debit*)
        $\mathit{used} := \mathit{used} \cup \{(e, g, c)\}$
        $\Delta' := \Delta' - \mathit{debit}$

Figure 16. An algorithm for selecting hot paths from a definite flow profile using the branch-flow metric. Fixes to the original algorithm, which are unrelated to our modifications for branch-flow metric, are underlined.