Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems

Rahul Joshi, Michael D. Bond[†], and Craig Zilles

Dept. of Computer Science, University of Illinois at Urbana-Champaign
[†] Dept. of Computer Sciences, University of Texas at Austin
E-mail: {rujoshi, zilles}@cs.uiuc.edu, mikebond@cs.utexas.edu

Abstract

In this paper, we present a technique for reducing the overhead of collecting path profiles in the context of a dynamic optimizer. The key idea to our approach, called ***Targeted Path Profiling*** *(TPP), is to use an edge profile to simplify the collection of a path profile. This notion of profile-guided profiling is a natural fit for dynamic optimizers, which typically optimize the code in a series of stages.*

TPP is an extension to the Ball-Larus Efficient Path Profiling algorithm. Its increased efficiency comes from two sources: (i) reducing the number of potential paths by not enumerating paths with cold edges, allowing array accesses to be substituted for more expensive hash table lookups, and (ii) not instrumenting regions where paths can be unambiguously derived from an edge profile. Our results suggest that on average the overhead of profile collection can be reduced by half (SPEC95) to almost two-thirds (SPEC2000) relative to the Ball-Larus algorithm with minimal impact on the information collected.

1. Introduction

Dynamic optimization systems such as Crusoe CMS [16], Dynamo [6], and Jalapeño [4] collect data about a program as it runs and use this data to improve performance. Typically, such optimizers optimize the code in a series of stages. Each stage represents a different trade-off between expended effort during optimization and the performance of the resulting code. In addition, some stages may be responsible for collecting profile information about the executable.

Control flow profiles are perhaps the most important form of profile collected. They allow the optimizer to focus its time on frequently executed (*i.e.*, "hot") portions of the program and apply transformations that benefit the hot portions at the expense of "cold" (*i.e.*, less frequently executed) portions of the program. Many optimizers collect control flow "point" profiles (*e.g.*, block or edge profiles) because they can be collected with minimal (0.5 to 3%) overhead via sampling [2, 24] or with hardware support [10, 11, 12, 14].

In this paper, we are concerned with the identification of higher-level control flow patterns in the form of hot paths. Knowledge of these frequently-executed sequences of basic blocks can be used to drive powerful optimizations (*e.g.*, superblock formation [15] and path-profile guided code optimizations [13]) that would typically be found in the final stages of dynamic optimizers. Ball, *et al.* showed that hot paths could be derived from edge profiles where there were few unbiased branches (a process they call *attribution of definite flow*) [9]. They found that they could attribute around 80%—76% for SPEC95 INT and 84% for SPEC95 FP—of the program's execution to acyclic paths. While promising, their technique leaves a sizable fraction of paths unaccounted for.

While a number of techniques have been proposed for collecting path profiles [6, 8, 23], none are as efficient as edge profiling. Whereas edge and block profiles—being "point profiles"—track the frequencies of individual events, path profiles require correlating the execution of multiple events. The complexity of correlation profiles comes from storing the profile in a manner which is space efficient and can be quickly queried[1]. Even if path fragments are collected in hardware (as can be done by the Pentium4 [20]), a challenge remains in aggregating them into a profile.

Ball and Larus [8] proposed an instrumentation-based path profiling algorithm that enables efficient collection and storage of path profiles in two ways: (i) The program is logically broken into independent acyclic regions that are profiled independently, and (ii) a minimal numbering algorithm is proposed that allows paths within these regions to be identified with a single integer. While breaking a program

1 The MRET technique [6] sidesteps this challenge by collecting a single trace. While statistically likely to find the hottest path, MRET doesn't provide the information about the path's relative frequency necessary to decide how aggressively to optimize the path. As a result, Dynamo is overly aggressive in programs with many unbiased branches, thrashes the code cache, and bails out [6].

into acyclic regions prevents identification of control flow correlations that cross region boundaries, Ball and Larus ensure that all overlapping paths begin and end at the same place, enabling the paths to be easily compared and aggregated. By assigning each path a unique number, this aggregation can be performed by simply incrementing a counter associated with the path's number.

For the SPEC95 suite of programs, Ball and Larus found that Path Profiler (PP, a tool based on their efficient algorithm) added 31% overhead on average, with overheads as high as 97% for `gcc` [8]. While this instrumentation would only need to be run temporarily (*e.g.*, during one stage of optimization) to identify hot paths, a less expensive means of path profiling remains desirable. Sampling techniques for instrumentation (*e.g.*, the Arnold-Ryder framework [5] and dynamic instrumentation [19, 22]) can be used to reduce the overhead rate, but do so by extending the time it takes to collect a given number of samples.

As we will show in Section 2.2, a sizable fraction of the overhead of PP can be attributed to either (i) paths that can be identified unambiguously from an edge profile or (ii) path counters maintained in a hash table. Because the number of potential paths can grow exponentially with the size of the acyclic regions, PP resorts to using hash tables to count paths when the number of potential paths makes allocating an array of counters impractical. This space-time trade-off is feasible because in practice only a small fraction of potential paths are exercised by programs, but the hashed regions consume a disproportionate amount of execution time.

In an attempt to reduce the overhead of collecting each path sample, we propose a variant of the Ball-Larus algorithm that we call *Targeted Path Profiling* (TPP), which is especially suited for staged dynamic optimizers. In such an optimizer, generally a rudimentary edge profile is available by the time that the optimizer decides that a routine is sufficiently hot to warrant collection of a path profile for path-based optimizations. TPP uses this edge profile to more efficiently collect the path profile. TPP is an example of a more general notion of *profile-guided profiling techniques*, which naturally fit in the context of dynamic optimizers, as discussed in Section 3.

Specifically, TPP uses the edge profile to (i) avoid instrumenting regions where the edge profile is sufficient to unambiguously identify paths and (ii) narrow the set of potential paths, allowing regions to be converted from using hash tables to using (faster) arrays of counters. Many potential paths (often as many as 99.99% of them) can be identified as definitely cold from an edge profile because the execution frequency of a path cannot exceed the frequency of any of its component edges. Removing these paths from consideration means that their execution frequencies will not be counted, but such information for cold paths is of little use to an optimizer. To ensure that executions of these cold paths do not incorrectly increment a valid path's counter, our instrumentation *poisons* the path counter register by setting upper bits not set by any valid path. Using the poison bits, our instrumentation can aggregate samples of all cold paths to a single counter, allowing TPP to track the fraction of paths that are not being counted. If this fraction is too high, a dynamic optimizer is free to re-instrument the region using a more recent edge profile. TPP's techniques are described in Section 4 and the instrumentation itself is described in Appendix A.

In an effort to isolate the performance-accuracy trade-off, our evaluation of TPP (Section 5) was not done in the context of a dynamic optimizer, because we felt the results might be clouded by the complexity of such systems. Instead, we compare to the Ball-Larus implementation of their algorithm, which is a highly-tuned and well-understood system, on full benchmark runs. We find that, when the edge profile is not unrepresentative of the profiling run, TPP has overhead that is half (SPEC95) or two-thirds (SPEC2000) that of the Ball-Larus algorithm, with minimal impact on the information collected (attribution of definite flow exceeding 98%).

While this reduction in overhead does come at the cost of some additional analysis time, we feel that this is a good trade-off given the trends in semiconductor technology. While it is increasingly difficult to improve the performance of a sequential thread, adding additional hardware contexts to support thread-level parallelism is straightforward. Targeted path profiling enables moving work out of the application thread and into the run-time optimizer, which can potentially execute in parallel with the application.

2. Ball-Larus Path Profiling

This section explains briefly the Ball-Larus path profiling algorithm and explores its major sources of overhead. More details can be found in [8].

2.1. Algorithm and Definitions

One of the important contributions of the algorithm is efficient enumeration of all acyclic paths in the control flow graph (CFG). The basic algorithm assigns an integer value to each edge in a directed acyclic graph (DAG) such that (i) the sum of the values of the edges on a path is unique for each path in the DAG, and (ii) the numbering of paths induced by the edge values is minimal (*i.e.*, if there are N acyclic paths, then the path numbers range from 0 to $N-1$). Once these edge values are computed, the algorithm uses the efficient event counting algorithm [7] to place instrumentation code on selected edges so that at the EXIT of the DAG, the path number is available in a register, which Ball and Larus call the *path register*. The instrumentation at the EXIT node then counts the path.

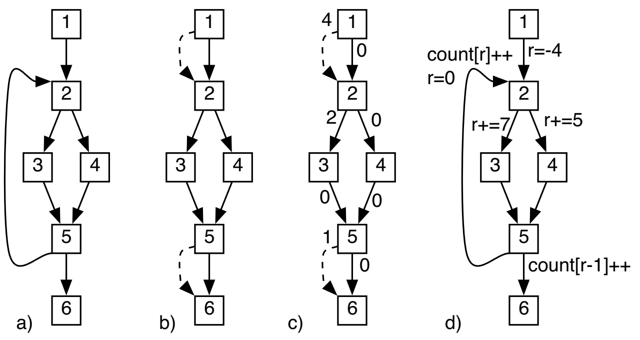


Figure 1. The Ball-Larus path profiling algorithm "breaks" back edges to create a directed acyclic graph (DAG): a) original control flow graph (CFG), b) CFG converted to a DAG, c) DAG with edge value assignments, d) CFG with final instrumentation.

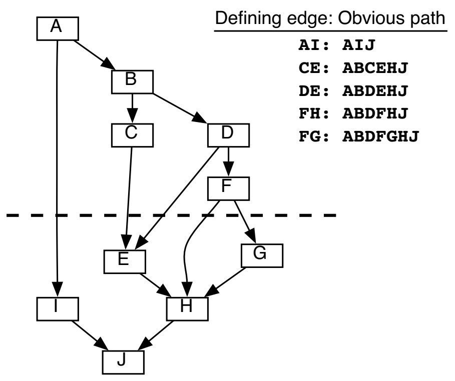


Figure 2. Example CFG where all paths are obvious. For each path in this routine, there is at least one *defining edge* (an edge included in only one path). This example is the routine `simplify_unary_operation` taken from `gcc`, after the cold paths have been removed.

Ball and Larus extend this basic algorithm to work for arbitrary CFGs with back edges. Each back edge is converted into two *dummy* edges: an edge from the ENTRY to the target of the back edge (which we call *entry dummy edge*) and the other from the source of the back edge to the EXIT (*exit dummy edge*). In effect, the target of each back edge starts a new acyclic path, and the source terminates an acyclic path. This process of "breaking" a back edge is shown in Figures 1a and 1b.

The Ball-Larus algorithm instrmuents each back edge so that it increments a path counter and resets the path register to zero. Therefore, the Ball-Larus algorithm counts acyclic paths that start at the ENTRY basic block or the target of a back edge and end at the EXIT basic block or the source of a back edge. When the number of enumerated paths grows too large, the algorithm selects an edge for *truncation*; entry and exit dummy edges are substituted for the truncated edge, reducing the number of paths through the DAG. Figure 1c shows the assignments of values to edges; the sum of the edge weights is the path number. Figure 1d shows the final instrumentation.

Bench mark	Overhead PP	Overhead Array	Overhead Hashing	DynPath Hashed	Hash Cost
compress	23.7%	23.8%	0.0%	0.0%	-
gcc	95.6%	16.4%	77.6%	61.3%	3.0
go	67.6%	14.3%	47.2%	34.9%	6.2
ijpeg	12.2%	11.6%	0.4%	2.2%	-
li	18.3%	19.2%	1.8%	0.1%	-
m88ksim	44.0%	33.4%	5.8%	4.1%	4.1
perl	34.2%	3.6%	28.3%	49.4%	8.0
vortex	34.6%	6.8%	33.0%	40.2%	7.2
INT Avg:	41.3%	16.1%	24.3%	24.0%	5.7
apsi	8.6%	7.6%	1.2%	4.0%	3.8
applu	4.8%	4.6%	0.0%	0.7%	-
hydro2d	29.7%	2.8%	25.4%	66.3%	4.6
mgrid	0.8%	0.8%	0.1%	0.0%	-
su2cor	7.3%	2.6%	4.6%	26.9%	4.8
tomcatv	12.2%	0.0%	12.6%	$>$ 99.9%	-
turb3d	20.2%	20.7%	0.3%	$<$ 0.1%	-
FP Avg:	11.9%	5.6%	6.3%	28.3%	4.4
Average:	27.6%	11.2%	15.9%	26.0%	5.2

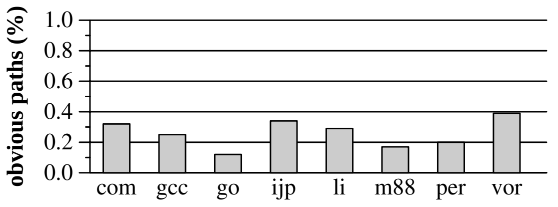
Table 1. Comparison of hash calls vs. array counter updates in PP. The *Overhead* columns shows the percentage increase in execution time due to profiling for the original PP (*PP*), for only routines with index- and address-based counters instrumented (*Array*), and for only routines using hash table-based counters instrumented (*Hashing*). The second to last column shows the fraction of dynamic paths counted that were in routines that used hash tables. *Hash cost* is the overhead per dynamic path increment of a hash-table based routine relative to a routine that uses arrays of counters. Data shown for the working subset of SPEC95.

2.2. Overhead

Ball and Larus found that their path profiling technique has an average overhead of 31% for the SPEC95 benchmarks. The overhead is substantially higher for the integer programs (average 45%) than it is for the floating point (FP) programs (average 20%), and it is as high as 97% for `gcc` and 73% for `perl`, two of the benchmarks most representative of modern programs.

Many of the programs with above average overheads record a sizable fraction of their paths using hash tables. PP relies on hash tables when the number of possible paths would make it infeasible to statically allocate an array of counters for the routine. While updating an array-based counter can be done with a load-add-store instruction sequence, the hash table-based counters require a function call that involves control flow and multiple memory operations to find the appropriate hash bucket to update. As shown in Table 1, we find that the overhead of counting a path in a hash table-based routine is a factor of 5.2 greater than in non-hashed routines. Thus, overhead can be reduced if the number of necessary counters can be brought below the hashing threshold.

Overhead can likewise be reduced by reducing the *number* of counter increments. We can avoid counting paths if the path frequencies can be computed directly from the edge



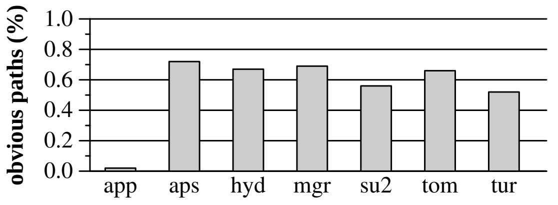


Figure 3. Fraction of total path counts due to obvious paths. Data collected with cold path and loop disconnection thresholds of 5% (see Section 5). If instrumentation for obvious paths can be removed, we would expect overhead to go down around this percentage. Data shown for the working subset of SPEC95.

profile. For example, Figure 2 shows a routine where every path includes at least one edge (the *defining edge*) that is part of only that path, which we call an *obvious path*. Instrumenting this routine provides no additional information. As shown in Figure 3, around 25% (SPEC95 INT) and 50% (SPEC95 FP) of dynamic path counter increments are in code regions where all path frequencies could be obtained directly from the edge profile. By removing the instrumentation in these regions, we expect overhead to drop roughly proportionately to these frequencies.

3. TPP in a Staged Dynamic Optimizer

While dynamic optimization is an attractive technique because it can exploit run-time information, actually performing the optimization requires execution resources. As a result, dynamic optimizers have been designed to balance this resource utilization with the expected benefit of optimization. To this end many dynamic optimizers are organized as a series of stages, each requiring a larger investment of resources for optimization and resulting in a higher degree of optimization. Typically, an optimization stage is invoked when the system has seen a fragment of code execute a sufficient number of times that it can expect a sufficient return to justify a further investment in optimization.

To achieve greater levels of optimization, later stages make greater investments in analysis to enable additional optimizations; for example, Jalapeño [4] does mostly local optimizations in its first stage, global optimizations in the second stage, and builds static single assignment (SSA) form in its third stage. Likewise, systems can use more extensive profile information in each stage, as more aggressive optimizations can better exploit this information. For example, an early stage may try to minimize taken branches

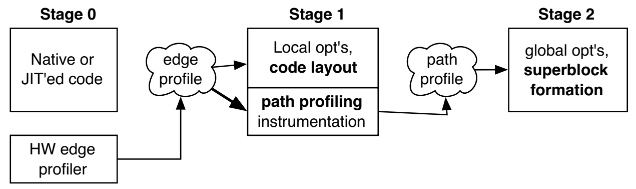


Figure 4. Staged optimization enables profile-directed profiling. A staged optimizer that first collects edge profiles can use that information to optimize the collection of a path profile.

through code layout, for which an edge profile is sufficient, while a later stage may perform superblock formation, where a path profile is desirable. Such a staging is shown in Figure 4.

Staging the profiling (as well as the optimization) seems natural because there will be a cost to collect (and to store) the profile, so that cost should only be born for code fragments that will reach the higher levels of optimization. For instrumentation-based path profiling, instrumentation can be added in an early stage (*e.g.*, stage 1 in Figure 4) for use in a later stage (*e.g.*, stage 2).

If the instrumentation is added after the dynamic optimizer has determined that a code region is hot, a rudimentary edge profile may be available for the code region. This paper shows how this edge profile can be used to reduce the overhead from executing instrumented code. While this reduction of overhead potentially comes at a cost of additional analysis and/or instrumentation time, it is important to note that those tasks can be executed *in parallel* with the application code, so they may minimally contribute to execution time on a system with under-utilized thread execution resources. We expect current trends toward multi-threading and chip multiprocessing to provide thread-parallel resources beyond what most workloads can exploit.

4. Techniques

In this paper, we present two orthogonal, but synergistic, techniques for reducing the overhead of Ball-Larus-style path profiling instrumentation. These techniques attack the two sources of overhead discussed in Section 2.2. The first technique (discussed in Section 4.1) reduces the number of possible paths enumerated by not counting suspected cold paths; in some routines, the number of possible paths is sufficiently low that hash table-based counters can be avoided. The second technique (discussed in Section 4.2) avoids counting paths whose frequencies can be derived directly from an edge profile.

4.1. Cold Path Elimination

As previously noted in Section 2.2, PP resorts to hash table-based counters to conserve memory for routines with many potential paths. A large number of potential paths typically results from code with stacked conditionals, like those shown in Figure 5a. In this structure, the number of paths grows exponentially with the height of the stack; in some routines, the number of acyclic paths exceeds 2^{64}.

In practice, though, only a very small fraction (*i.e.*, much less than one percent) of the potential paths are ever executed. For optimization, we are only concerned with measuring the frequency of frequently executed (or *hot*) paths. Unexecuted paths need not be allocated a counter, and we can do without tracking cold paths as well. Although we cannot, in general, know *a priori* which will be the hot paths, an edge profile can accurately identify cold paths in a program. Specifically, if a path P consists of edges $e_1, e_2, \cdots, e_n$, then the number of times that path P is executed ($f(P)$) is:

$$f(P) \leq \min_{1 \leq i \leq n} f(e_i)$$

Quantitatively, the execution count of a path cannot exceed the execution count of any edge on that path. Qualitatively, if any edge on the path is "cold," the entire path is cold. Thus, we identify cold paths by first identifying cold edges and then identifying a path as cold if any of the edges on it is cold. We discuss our criterion for cold edges below, in Section 4.1.1.

By logically removing cold edges from the CFG, we eliminate any (cold) paths that include them. After cold edges have been removed, the Ball-Larus algorithm is used to enumerate paths in the resulting graph. For the example in Figure 5, if two of the branches are highly biased, there are only four potential "hot" paths for the routine; in general, each cold branch edge removed can eliminate as many as half of the potential paths. We present a few subtleties in the algorithm related to loop back edges in Section 4.1.2.

Since we have not considered the cold edges in the above analysis, execution paths through cold edges do not compute a meaningful value in the path register. The computed value may alias to another path's value or be outside the range of the path counter array. To preserve the purity of our counters (as well as the rest of the address space), we mark the path register's value as *poisoned* on any transition into a cold region of the CFG. The poisoning is implemented by setting the upper two bits of the path register—doable with a single SPARC instruction (see Appendix A.2)—effectively making the path value a large negative number. Since no valid path will set these bits—PP truncates paths when the number of potential paths exceeds $10,000,000$—poisoned paths will contain values sufficiently far from valid paths that we can trivially distinguish between the two.

Having distinguished the poisoned counter values, we must consider what to do when one is encountered. Two alternatives worth considering are:

1. Incrementing a *cold counter* to track the frequency of all cold paths in the code region.

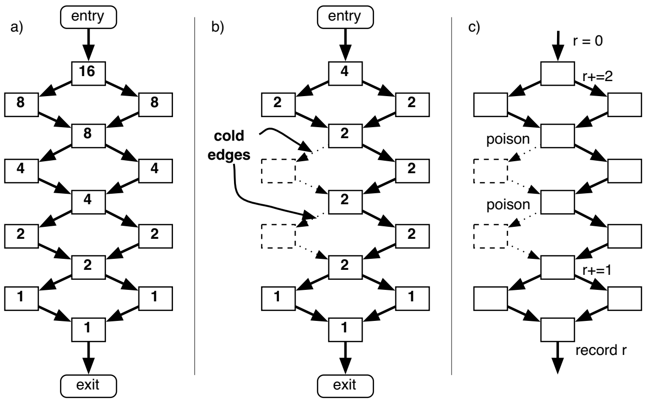


Figure 5. Avoiding a large number of potential paths, by not enumerating likely cold paths. a) Stacked conditionals yield a number of paths exponential with the height of the stack; each block is labeled with the number of paths reachable from that point on. b) In practice, many edges are cold; each cold edge ignored can cut the potential number of paths in half. c) TPP instrumented code; cold edges are instrumented to "poison" the path register, so that paths through cold edges are not counted incorrectly.

2. Ignoring poisoned path counters.

The first alternative is appealing, as it can provide feedback to the dynamic optimizer on the quality of the resulting path profile. For example, if a path that was cold before instrumentation later becomes hot, the cold counter will have a nontrivial number of counts. Based on the number of cold counts, the optimizer can consider re-instrumenting the code region to collect a better path profile. The second alternative would be worth considering if it significantly reduced overhead, but in our experiments execution time decreases by only 1% (*i.e.*, average overhead decreases from 14% to 13% for SPEC95).

In either case, in code where the path register can potentially be poisoned, the counter increment instrumentation must check for poisoned path registers. When a cold counter is maintained, it should be incremented on a poisoned path rather than the counter indicated by the path register. Thus, logically, the instrumentation at the end of a path looks like:

```
if r < 0  // if path register poisoned
   cold_counter++
else
   count[r]++
```

This potential branching at the end of each acyclic path can increase the runtime penalty of the instrumentation code. In Appendix A, we demonstrate how this logic can be implemented with a conditional move.

4.1.1. Cold Criterion Our algorithm is orthogonal to the criterion for deciding whether an edge is cold. In our results in Section 5, we consider a criterion based on branch biases. This *local* criterion removes edges where the ratio of the edge's execution frequency to that of its source block is below a supplied threshold.

Some edges in the CFG that are not cold according to the criterion may be cold transitively. Specifically, any block or edge is considered cold if there is no hot path from ENTRY to EXIT through it. All cold blocks and edges are logically removed from the CFG. Instrumentation for poisoning the path register is introduced whenever a hot block is the source of a cold edge.

4.1.2. Handling Back Edges Back edges need special consideration in the Ball-Larus algorithm because they are not part of any acyclic path. In the original algorithm, back edges are instrumented to increment the counter associated with the current path register value and reset the path register value to 0. This remains unchanged for hot back edges. In this section, we discuss how cold edges (i) increment path counters and (ii) reset the value of the path register.

When cold back edges are logically removed from the CFG, no entry and exit dummy edges are inserted. As a result, the path numbering algorithm will not assign a number to paths that terminate at the back edge. Thus, the instrumentation on cold back edges ignores the value of the path register and always records an execution of a cold path[2].

Similarly, if all back edges for a particular loop header have been removed, then there will be no entry dummy edge to the loop header. In this situation, all (cold) back edges for the loop header must initialize the path register to a poisoned value, as no paths starting at the loop header will have been enumerated (Figure 6a). If not all back edges are cold or one of the entry edges was truncated, then an entry dummy edge exists, and all back edges (both hot and cold) can reset the path register to zero (Figure 6b). Lastly, if the whole loop body is cold, then the path register need not be modified; we know that it will contain a poisoned value upon execution of the back edge, and that is what we want it to contain (Figure 6c).

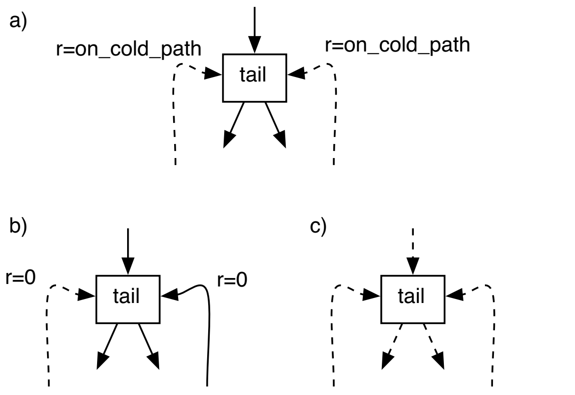


Figure 6. Instrumenting a cold back edge for reinitializing the path register. a) Cold back edge poisoning path register. b) Cold back edge resetting path register. c) Cold back edge not modifying the path register. (Cold edges are shown as dashed arcs.)

4.2. Obvious Path Elimination

As was shown in Figure 2, the frequency of some paths can be derived directly from an edge profile. We refer to such paths as *obvious paths*. Formally, an obvious path is a path that contains at least one edge that lies only on that path. This edge is called the *defining edge* for that obvious path, and the execution count of an obvious path is equal to the execution count of its defining edge.

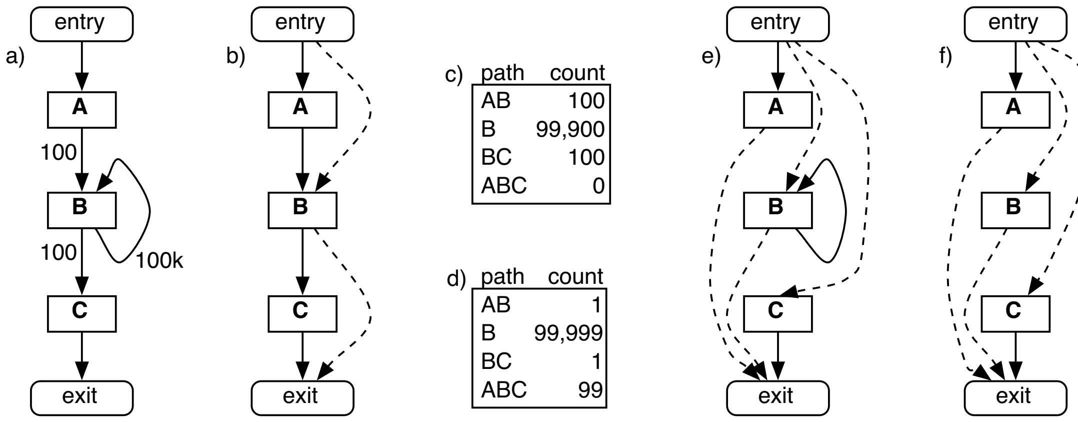
In regions where all paths are obvious (*e.g.*, the routine in Figure 2), no instrumentation is necessary. Eliminating this instrumentation reduces overhead from both path register updates and counter increments. While loops are responsible for a large fraction of the path increments, they inhibit elimination of obvious paths. In Section 4.2.1, we demonstrate this problem and how cutting loops with high iteration counts from the CFG enables them to be considered for obvious path removal. In Section 4.2.2, we explain how TPP detects obvious paths and show how it is possible to eliminate some of the instrumentation from a routine when only some of the paths are obvious; we also discuss how to recover the complete path profile from what is collected by TPP and an edge profiler.

4.2.1. Considering Loops for Obvious Path Elimination Because of their relative execution frequency, loops contribute the lion's share of the increments to path counters as well as the overhead. In many cases, the paths through the loop body are obvious. For example, Figure 7a shows a loop whose body consists of a single block; there is only one path through the loop body, and we know it will be executed 100,100 times. There is little point to instrumenting such a loop.

A difficulty arises because the Ball-Larus algorithm does not consider the loop in isolation. After the back edge is removed and the dummy edges are inserted (shown in Figure 7b), there are now four paths through the routine, and none of them are obvious. The only benefit of this path profiling instrumentation over an edge profile is that it indicates how many times the routine executed exactly one iteration of the loop. That is, the case where the loop always executes more than one iteration (Figure 7c) can be distinguished from the case where only one execution executes more than one iteration (Figure 7d). While this information *could* be used to drive optimization (*e.g.*, peeling the first iteration[3]), the expected benefit is marginal when the average

2 With indirect branches, a block can potentially be the source of both a cold back edge and a hot back edge of different loops. In such a situation, there is a dummy edge to exit from the block that was not eliminated, so there are valid paths that end at the block. Thus, all back edges from the block can include the standard path recording instrumentation. We have not observed this situation in practice.

3 The information necessary to decide whether peeling is worthwhile can be collected by peeling the first iteration and collecting an edge profile.



c)

path	count
AB	100
B	99,900
BC	100
ABC	0

d)

path	count
AB	1
B	99,999
BC	1
ABC	99

Figure 7. Cutting loops from the CFG to enable obvious path elimination. a) A single block loop with a high iteration count. b) This code will not be considered obvious because there are two paths into and two paths out of block B; there are many path profiles that match this edge profile. c) A matching path profile with many AB and BC paths. d) A matching path profile with many ABC paths. e) Dummy edges from ENTRY and to EXIT are substituted for edges into and out of loop bodies. f) The back edge is removed resulting in a graph with three obvious paths.

iteration count is high[4].

The loss of information can be quantified via the average length of the dynamic paths measured. When loops are separated from the rest of the CFG, some paths are truncated. These truncated paths reduce the average path length relative to a profile where the loops are left in place. In practice, we've found the loss to be negligible: average path length is reduced only 1.5% when loops with average iteration counts as little as 10 are detached. When the threshold is raised to 100, the effect is imperceptible.

If this loss of information is deemed acceptable, the CFG can be transformed to enable the detection of obvious paths. We truncate all the entry and exit edges of the loop, replacing them with dummy edges, as explained in Section 2.1. Figure 7e shows the example after this substitution for $A \rightarrow B$ and $B \rightarrow C$. After dummy edges are substituted for back edges (Figure 7f), there are three non-overlapping paths from ENTRY to EXIT, so the routine consists of only obvious paths. The algorithm tracks the relationship between the removed CFG edges and the dummy edges to allow the correct attribution of edge counts to paths.

In general, the code before, in, and after the loop (represented by blocks A, B, and C) could be any (acyclic) code region. Since these regions are now independent, if one of them is not entirely obvious, only it needs to be instrumented.

4.2.2. Obvious Path Detection and Elimination Algorithm To find obvious paths in a DAG, we first find the set of all *defining edges* (edges that lie on only one path). By definition, there is a single path from ENTRY to the defining edge and a single path to EXIT from it. Thus, to find all defining edges, we find the set of blocks with a single path from ENTRY (ENTRY_S) and those with a single path to EXIT (EXIT_S); finding each set requires a single traversal of the DAG. Edges with a source in ENTRY_S and a target in EXIT_S are defining edges.

Upon finding a defining edge and its obvious path, we cannot immediately eliminate the instrumentation on its edges, because it may be needed to record other (non-obvious) paths. In fact, we only eliminate obvious paths when all the paths reaching a recording point are obvious, because that allows the whole region to be left uninstrumented, yielding a significant overhead reduction. For this reason, eliminating instrumentation is a two-step process: (i) elimination of unnecessary recording points and (ii) elimination of unnecessary edge instrumentation.

A recording point is unnecessary if it is only reachable by obvious paths. Thus we traverse the DAG from ENTRY to EXIT only using non-defining edges. Any recording point **not** reached during this traversal is an *obvious recording point* and can be eliminated using the algorithm shown in Figure 8.

If there is no path from an edge to a non-obvious recording point, then that edge does not need to be instrumented; all paths through such an edge are obvious paths. The algorithm for finding these edges is a reachability analysis similar to the previous one for finding obvious recording points, except that we do a backward traversal starting at all the non-obvious recording points. After the analysis completes, we remove all the unreached edges from the DAG, eliminating any instrumentation on them. In case all the recording points in the DAG are found to be obvious, all paths in the routine are obvious, and the routine need not be instrumented at all. In general, if only a part of the routine

4 In the more general case, where there is control flow both above and below the loop, the Ball-Larus path profile will expose path correlations between the code before and after the loop, when the loop is exited after a single iteration. Again, when the average loop iteration count is high, the expected benefit of this information is small.

```
EdgeSet FindObviousRecordingPoints(CFG $g$):
    EdgeSet obvRecordPoints = pred(exit($g$))
    EdgeSet WS = non-defining successors of entry($g$)
    while WS not empty
        edge $e$ = WS.remove()
        // recording point reached by a non-obvious path
        if $e \in$ obvRecordPoints
            remove $e$ from obvRecordPoints
        else
            for each edge $f$ = target($e$) $\rightarrow v$
                if not IsDefining($f$)
                    add $f$ to WS
    return obvRecordPoints
```

Figure 8. An $O(e)$ algorithm for finding obvious recording points.

is obvious, the above algorithm can eliminate instrumentation from just that part.

To reconstruct the complete path profile, the above algorithm can be used again during profile analysis to find the set of defining edges actually eliminated during instrumentation. For each such defining edge, the (single) path containing that edge is traced by tracing unique paths from ENTRY to its source and from its target to EXIT. The execution count of this path is set equal to the execution count of its defining edge. Alternatively, the relationships between defining edges and paths can be retained from the original analysis.

4.3. Interaction between Obvious Path Elimination and Cold Path Elimination

Obvious path elimination can be done either before or after cold path elimination. If obvious path elimination is done after cold path elimination then additional paths will be classified as obvious. While the defining edges of these additional obvious paths are on a single hot path, they are on a number of cold paths as well. As a result, the actual execution count of the defining edge would be *more* than the execution count of the obvious path. However, when a representative edge profile is used, the misattribution can be controlled by selecting the threshold for cold path elimination. Misattribution of counts is accounted for in our results in Section 5.

5. Experimental Methodology and Results

In this section, we evaluate the overhead-accuracy trade-off of our proposed techniques. We have developed a tool called *Targeted Path Profiler* (TPP) that instruments executables for targeted path profiling. TPP is based on the PP implementation of the Ball-Larus efficient path profiling algorithm [8]. PP is part of Executable Editing Library (EEL), a binary analysis and editing tool for SPARC that hides many of the platform-specific details of modifying executables [17]. We use Quick Profiler and Tracing Tool (QPT2), the edge profiler from [8], to collect edge profiles. TPP eliminates cold paths from a routine only when doing so permits the substitution of a counter array for a hash table. Obvious paths, however, are eliminated from all routines. In general, TPP reuses the analysis and instrumentation from PP where possible but adds functionality to analyze and instrument routines with cold paths and detect and eliminate instrumentation for obvious paths. More details about the implementation can be found in Appendix A.

Results were collected for the SPEC95 and SPEC2000 benchmark suites. We present results of a subset; PP and QPT2 failed to correctly instrument the omitted benchmarks benchmarks in our environment. In a few cases, we identified the problematic routines in the benchmarks and excluded these routines from instrumentation (to enable relative comparisons, the routines were excluded for runs of both PP and TPP). The benchmarks were compiled using the Sun C compiler (version 5.2) and run with the reference input set. As we expect the edge profiles collected in an early stage of the optimizer to be representative of the execution during the path profile collection, our experiments used edge profiles for the `ref` input set. Runtime results were collected on a quiet Sun-Blade-1000 running SPARC Solaris 5.8. In order to allow comparisons to previous work [8, 9], we present mostly results from SPEC95; note that TPP works even better with SPEC2000.

TPP can be configured with two parameters: (i) the threshold for the cold edge criterion and (ii) the threshold for disconnecting loops from the CFG. A TPP configuration is specified as TPP(i, j), where i is the cold edge threshold (edges responsible for less than $i\%$ of flow from their source are removed), and j is the loop disconnection threshold (loops with entry edges executed less than $j\%$ of the loop header execution frequency are removed).

Our accuracy metric, attribution of definite flow (AoDF), is $(100\% - \%_{overcount} - \%_{undercount})$. Predominantly, reductions in AoDF are due to "cold counts" that cannot be attributed to any particular path[5]. As previously noted in section 4.3, TPP may also overcount obvious paths due to the elimination of cold paths. We compute the percentage of paths undercounted and overcounted (weighted by path length in blocks), relative to PP's results.

5.1. Results

Our experiments demonstrate that targeted path profiling (TPP) reduces the overhead of the original Ball-Larus algorithm by about half (SPEC95, Table 2) to almost two-thirds

5 Comparison of profiles collected by PP and TPP is complicated by the fact that the two tools may truncate paths along different edges. To enable a quantitative comparison of the two profiles, we split all recorded paths at the union of both truncated edge sets. When PP truncates at an edge that TPP designates as cold, PP and TPP will have different number of counts for the path, as TPP will only count the executions that do not pass through any cold edges.

	Overhead (Attribution of Definite Flow)									
Benchmark	PP		TPP (1,1)		TPP (5,5)		TPP (10,10)		TPP (5,15)	
compress	23.7%	(100%)	15.9%	(100%)	16.2%	(100%)	12.0%	(99.7%)	12.2%	(99.7%)
gcc	95.6%	(100%)	68.6%	(99.2%)	55.7%	(96.0%)	52.8%	(91.9%)	52.6%	(95.9%)
go	67.6%	(100%)	48.4%	(99.9%)	50.5%	(99.6%)	47.9%	(98.5%)	49.2%	(99.5%)
ijpeg	12.2%	(100%)	9.0%	(99.9%)	7.6%	(99.9%)	8.4%	(99.9%)	8.0%	(99.6%)
li	18.3%	(100%)	15.6%	(100%)	14.2%	(100%)	12.6%	(98.8%)	12.4%	(98.8%)
m88ksim	44.0%	(100%)	35.3%	(99.7%)	38.0%	(97.9%)	40.6%	(97.0%)	36.5%	(98.0%)
perl	34.2%	(100%)	7.4%	(99.4%)	8.1%	(96.8%)	6.5%	(93.0%)	8.0%	(96.4%)
vortex	34.6%	(100%)	9.2%	(99.9%)	8.4%	(99.5%)	7.0%	(99.2%)	6.2%	(99.6%)
INT Avg:	41.3%	(100%)	26.2%	(99.8%)	24.8%	(98.7%)	23.5%	(97.3%)	23.1%	(98.4%)
applu	4.8%	(100%)	4.4%	(100%)	4.6%	(100%)	4.5%	(100%)	4.6%	(100%)
apsi	8.6%	(100%)	5.2%	(100%)	3.4%	(99.7%)	2.6%	(99.5%)	2.4%	(99.5%)
hydro2d	29.7%	(100%)	3.6%	(98.6%)	3.6%	(98.0%)	3.2%	(93.9%)	3.7%	(98.0%)
mgrid	0.8%	(100%)	0.8%	(100%)	0.7%	(99.1%)	0.5%	(99.1%)	0.5%	(98.4%)
su2cor	7.3%	(100%)	2.7%	(100%)	1.7%	(100%)	1.4%	(100%)	1.4%	(100%)
tomcatv	12.2%	(100%)	0.8%	(99.9%)	0.9%	(99.9%)	0.6%	(99.9%)	0.6%	(99.9%)
turb3d	20.2%	(100%)	11.4%	(99.9%)	8.6%	(99.8%)	7.0%	(99.2%)	6.4%	(99.1%)
FP Avg:	11.9%	(100%)	4.1%	(99.8%)	3.4%	(99.5%)	2.8%	(98.8%)	2.8%	(99.3%)
Average:	27.6%	(100%)	15.9%	(99.8%)	14.8%	(99.1%)	13.8%	(98.0%)	13.6%	(98.8%)

Table 2. Comparison of targeted path profiling (TPP) with Ball-Larus path profiling (PP) for a subset of SPEC95 benchmarks. *Overhead* is the increase in execution time due to profiling. *Attribution of Definite Flow* is the fraction of dynamic paths (weighted by path length in basic blocks) that can be derived from the instrumentation.

	Overhead (Attribution of Definite Flow)									
Benchmark	PP		TPP (1,1)		TPP (5,5)		TPP (10,10)		TPP (5,15)	
gzip	35.9%	(100%)	27.7%	(99.7%)	25.0%	(99.5%)	26.0%	(97.7%)	19.6%	(99.5%)
gcc	87.2%	(100%)	48.4%	(99.4%)	36.7%	(97.5%)	33.7%	(96.0%)	33.3%	(97.5%)
mcf	12.7%	(100%)	4.1%	(100%)	3.9%	(100%)	0.5%	(100%)	0.8%	(100%)
crafty	38.3%	(100%)	29.5%	(100%)	36.9%	(95.9%)	24.0%	(94.7%)	20.2%	(95.8%)
parser	33.0%	(100%)	18.6%	(100%)	18.9%	(100%)	16.1%	(97.3%)	18.2%	(99.5%)
perlbmk	29.6%	(100%)	11.9%	(99.7%)	9.6%	(97.4%)	5.9%	(95.8%)	5.5%	(99.5%)
gap	32.1%	(100%)	16.0%	(99.8%)	14.7%	(98.5%)	13.5%	(96.5%)	15.2%	(98.5%)
vortex	51.6%	(100%)	13.5%	(99.9%)	18.8%	(99.6%)	11.0%	(99.4%)	8.4%	(99.7%)
bzip2	62.6%	(100%)	32.2%	(99.8%)	26.8%	(98.6%)	27.3%	(98.5%)	27.2%	(98.4%)
twolf	50.7%	(100%)	21.7%	(100%)	22.6%	(99.5%)	19.8%	(97.6%)	20.7%	(99.5%)
INT Avg:	43.4%	(100%)	22.4%	(99.8%)	21.4%	(98.6%)	17.8%	(97.3%)	16.9%	(98.6%)
mgrid	1.1%	(100%)	1.5%	(100%)	0.7%	(99.3%)	0.7%	(99.3%)	1.2%	(99.3%)
art	67.2%	(100%)	8.0%	(100%)	8.0%	(100%)	8.5%	(100%)	9.0%	(100%)
lucas	3.5%	(100%)	0.4%	(100%)	0.9%	(95.8%)	1.5%	(95.8%)	1.8%	(100%)
fma3d	17.0%	(100%)	5.1%	(100%)	6.3%	(99.8%)	4.8%	(97.9%)	5.9%	(98.9%)
FP Avg:	22.2%	(100%)	3.7%	(100%)	4.0%	(98.7%)	3.9%	(98.2%)	4.5%	(99.6%)
Average:	37.3%	(100%)	17.0%	(99.9%)	16.4%	(98.7%)	13.8%	(97.6%)	13.3%	(98.9%)

Table 3. Comparison of targeted path profiling (TPP) with Ball-Larus path profiling (PP) for a subset of SPEC2000 benchmarks.

(SPEC2000, Table 3) with a minimal loss of information. Most of the overhead reduction is achieved by eliminating the very infrequently executed paths, as shown by the results for TPP(1,1), which has the most conservative thresholds. In both SPEC95 and SPEC2000, the floating point benchmarks benefit more than integer benchmarks, which correlates to the higher fraction of obvious paths shown in Figure 3.

Tables 2 and 3 show the overhead and accuracy of TPP under a variety of parameters. While the results are relatively insensitive to the selection of parameters in this range, it should be noted that overhead does not decrease monotonically with reduction of precision; we attribute this variation to second order effects due to data layout. TPP(5,15), which uses a 5% threshold for the cold edge criterion and a 15% threshold for loop exits (*i.e.*, average iteration count > 6), seems to be a good trade-off between accuracy and overhead. For SPEC95 (SPEC2000), average overhead is reduced from 28% (37%) to 14% (13%) while retaining an attribution of definite flow of 98.8% (98.9%).

Benchmark	Enumerated Paths (in thousands) PP	TPP(1,1)		TPP(5,5)		TPP(10,10)		TPP(5,15)	
compress	2.7	0.5	(18%)	0.5	(18%)	0.4	(15%)	0.4	(15%)
gcc	14,970,000	9,036,000	(60%)	6,554,000	(44%)	5,244,000	(35%)	6,370,000	(43%)
go	688,000	649,000	(94%)	629,000	(91%)	606,000	(88%)	616,000	(90%)
ijpeg	5,490	25.3	(< 1%)	18.5	(< 1%)	15.9	(< 1%)	13.8	(< 1%)
li	53,000	9.2	(< 1%)	9.1	(< 1%)	9.0	(< 1%)	9.0	(< 1%)
m88ksim	73,000	30.0	(< 1%)	25.8	(< 1%)	24.3	(< 1%)	24.8	(< 1%)
perl	1,237,000	27.9	(< 1%)	21.7	(< 1%)	20.1	(< 1%)	19.0	(< 1%)
vortex	2,055,000	98,200	(5%)	151	(< 1%)	150	(< 1%)	150	(< 1%)
applu	133,000	4.0	(< 1%)	6.4	(< 1%)	6.4	(< 1%)	4.6	(< 1%)
apsi	131,000	6.7	(< 1%)	7.2	(< 1%)	6.9	(< 1%)	6.9	(< 1%)
hydro2d	130,000	2.4	(< 1%)	1.8	(< 1%)	1.8	(< 1%)	1.8	(< 1%)
mgrid	5,900	2.6	(< 1%)	0.5	(< 1%)	0.4	(< 1%)	0.3	(< 1%)
su2cor	149,000	10.1	(< 1%)	9.6	(< 1%)	10.6	(< 1%)	10.2	(< 1%)
tomcatv	8,700	0.3	(< 1%)	0.3	(< 1%)	0.3	(< 1%)	0.3	(< 1%)
turb3d	75,000	6.7	(< 1%)	1.7	(< 1%)	1.6	(< 1%)	1.6	(< 1%)

Table 4. Comparison of the number of static paths between PP and TPP with various cold edge and loop disconnection criteria for a subset of SPEC95 benchmarks.

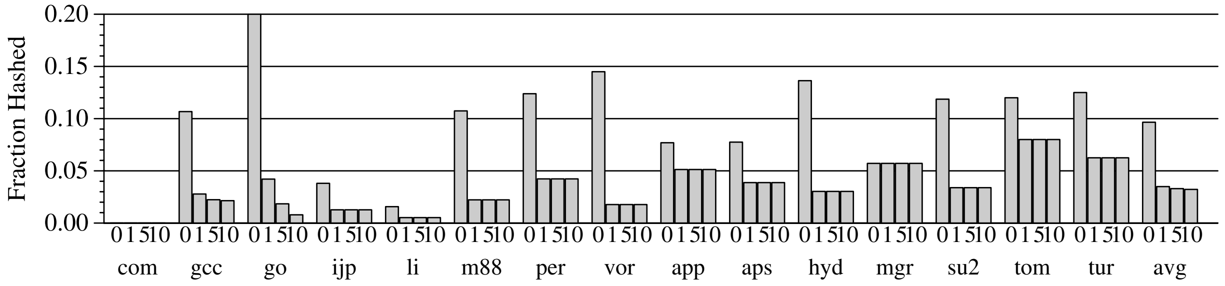


Figure 9. Reduction in fraction of routines that require hash table-based counters. Data shown for *O*: original PP, and TPP with *1*, *5*, and *10*% cold threshold. Again, it can be seen that most of the benefit is achievable with the lowest threshold.

The reduction of overhead comes primarily from cold path removal and the synergy between cold path removal and obvious path removal. Roughly 60% of the overhead reduction comes from cold path removal alone (data not shown). Cold path removal is effective at removing static paths from consideration. Table 4 shows that more than 99% of static paths are removed in all but four SPEC95 benchmarks. In eight cases, more than 99.99% of the static paths are removed. This reduction in the number of static paths enumerated does a good job of reducing the number of routines that require hash tables. As shown in Figure 9, on average, the number of hashed routines is reduced by two-thirds. For integer codes, which have many more hashed routines, the average reduction exceeds 80%. Without cold path removal, the benefit from obvious path removal is negligible (data not shown).

While TPP's optimizations reduce its run-time overhead, the additional analysis required slows instrumentation. We estimate that TPP takes on average 74% longer (96% for INT and 19% for FP) to analyze and instrument a routine than PP does. We have spent no effort in optimizing TPP's instrumentation performance and expect that this could be significantly improved because TPP does relatively little work beyond PP. In addition, it is desirable to eliminates cold paths only when it enables conversion of a hash-table to an array or a region to be considered obvious, which requires estimating of the benefit of our optimizations. Currenly, for simplicity, our implementation processes some routines as many as three times to decide whether to deploy the TPP optimized code. We believe that this inefficiency can be avoided by restructuring the code. Furthermore, the run-time cost of instrumentation could be reduced by doing some of the analysis (*e.g.*, building the CFG) off-line.

6. Related Work

Complimentary research has explored how to improve the quality of information from path profiles by extending beyond the intraprocedural, acyclic paths measurable by the Ball-Larus algorithm. Interprocedural Path Profiling [18] extends the Ball-Larus technique to interprocedural paths. More recently, a technique to profile overlapping path fragments (from which longer interprocedural and cyclic paths can be estimated) has been proposed [21]. The increased overhead of these techniques (*e.g.*, the second algorithm's overhead is a factor of four larger than the Ball-Larus algorithm [21]) can potentially be reduced by targeted path profiling.

Another technique for reducing the overhead of Ball-Larus-style path profiling, Selective Path Profiling (SPP) [3], tries to eliminate "probes" (updates of the path register) not necessary for distinguishing a specified subset of acyclic paths. Their results show a significant reduction in static probes for "tree-like" methods when only a few (1-5) paths are of interest; no overhead results are provided. In contrast, TPP tries to reduce overhead by eliminating path counter increments (when path counts are "obvious") and converting regions from hash table-based to array-based counters.

7. Conclusion

This paper presents targeted path profiling, a low overhead path profiling technique for dynamic optimization systems. Targeted path profiling, a modification of the Ball-Larus efficient path profiling algorithm, reduces overhead by converting routines from hash table-based counters to array-based counters and by eliminating instrumentation when all paths in a region can be derived from an edge profile. This profile-guided profiling technique requires an edge profile to be available at instrumentation time, but we expect this to be the case in a staged dynamic optimizer.

Our results demonstrate that targeted path profiling is a promising technique for reducing overhead; our tool TPP has one-half (SPEC95) or almost one-third (SPEC2000) of the overhead of the Ball-Larus tool on which it is based, while collecting only slightly less information. As this reduction in profiling overhead likely comes at a cost of additional analysis and instrumentation effort, TPP can be viewed as a technique to push work from application threads into threads of the dynamic optimization system. Given trends toward higher degrees of thread-level parallelism in hardware, we believe this is a desirable trade-off.

8. Acknowledgments

This research was supported in part by NSF CCR-0311340 and EIA-0224453 and equipment donations from Intel and AMD. We thank Glenn Ammons for his assistance with PP and Vikram Adve and the anonymous reviewers for suggestions to improve the paper.

References

[1] G. Ammons. Personal Communication, 2003.

[2] J. M. Anderson et al. Continuous profiling: where have all the cycles gone? In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 1–14. ACM Press, 1997.

[3] T. Apiwattanapong and M. J. Harrold. Selective Path Profiling. In *Proc. of Program Analysis for Software Tools and Engineering (PASTE)*, Nov. 2002.

[4] M. Arnold et al. Adaptive Optimization in the Jalapeño JVM. In *Conference on Object-Oriented*, pages 47–65, 2000.

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

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

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

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

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

[10] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, 2–4 December 1996. ACM Press.

[11] T. M. Conte, B. A. Patel, and J. S. Cox. Using Branch Handling Hardware to Support Profile-Driven Optimization. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 12–21, San Jose, CA, USA, 30 November–2 December 1994. ACM Press.

[12] J. Dean et al. ProfileMe : Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

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

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

[15] W.-M. W. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, pages 229 – 248, 1993.

[16] A. Klaiber. The Technology Behind Crusoe Processors. Technical report, Transmeta Technical Brief, 2000.

[17] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[18] D. Melski and T. Reps. Interprocedural Path Profiling. In *Proceedings of the 8th International Conference on Compiler Construction*, pages 47–62, Amsterdam, The Netherlands, 22–26 March 1999. Springer-Verlag.

[19] B. P. Miller et al. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, 1995.

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

[21] S. Tallam, X. Zhang, and R. Gupta. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In *IEEE-ACM International Symposium on Code Generation and Optimization (CGO)*, March 2004.

[22] O. Traub, S. Schechter, and M. Smith. Ephemeral Instrumentation for Lightweight Program Profiling, 2000.

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

[24] C. X. Zhang et al. System Support for Automated Profiling and Optimization. In *Proc. 16th Symposium on Operating System Principles*, pages 15–26, Oct. 1997.

A. Instrumentation Code

TPP, like the PP tool on which it is based, instruments executables by inserting snippets, or several consecutive SPARC instructions, into program binaries along control flow graph edges. TPP uses two modes for array-based path counting: addressing (used when the maximum edge increment fits in 11 bits) and indexing (used when the maximum increment requires up to 13 bits). Due to the similarity of the code snippets, we show only code for addressing mode (Appendix A.1); indexing mode is slightly more complex because the path register must be scaled and added to the array base address. Appendix A.2 shows the code placed on presumed cold edges to poison the path register. Appendix A.3 discusses the need to instrument dummy edges.

A.1. Path Recording Instrumentation

For the array-based recording code, TPP allocates an array of 4-byte counters in the global data segment—one for each path enumerated. In addition, TPP allocates a "cold counter" at the start of each routine's array of counters. Thus, the address of a routine's cold counter (`cc_addr`) is simply `array_base - 4` (counter size is four bytes), where `array_base` is the address of the first *hot* counter in the counter array.

Paths end at loop back edges and EXIT. TPP instruments both of these cases with snippets that increment a path counter. In routines with some cold paths, TPP inserts snippets that increment a hot path or cold path counter, depending on whether the path register has been poisoned. The following code snippet accomplishes this in addressing mode:

```
sethi  %hi(cc_addr - inc), %tmp1
or     %tmp1, %low(cc_addr - inc), %tmp1
sra    %pr, 31, %tmp2
movrnz %tmp2, %tmp1, %pr
ld     [%pr + inc], %tmp1
add    %tmp1, 1, %tmp1
st     %tmp1, [%pr + inc]
```

The first two instructions set register `tmp1` to `cc_addr - inc` rather than simply `cc_addr` because the load and store instructions implicitly add `inc` to the counter address. If the path register is negative, the `sra` and `movrnz` instructions set the path register to the address of the cold counter; otherwise, if the current path is hot, the path register remains unchanged and will hold the address of the path's counter. The last three instructions increment the counter whose address is in the path register. This snippet uses two temporary registers besides the path register.

A.2. Cold Edge Instrumentation

Because path numbers are relatively small ($< 10,000,000$), TPP uses the highest (31st) bit of the path register to signify that the path register has been poisoned: A negative path register indicates a cold path, and nonnegative path register indicates a hot path. When indexing mode is used, the execution of some hot paths will cause the path register to become negative temporarily, due to a negative increment. However, the path register will be nonnegative at the end of the path because path numbers are nonnegative.

As presented in Section 4, our algorithm instruments every cold edge that has a hot source. TPP inserts the following snippet along these edges to mark the current path as cold:

```
sethi   %hi(0xC0000000), %pr
```

The snippet sets the **two** highest bits and clears the other 30 bits of the path register because non-cold edges encountered later on the same path may increment or decrement the path register by small amounts. Setting the path register to a value that is not close to either the smallest or largest negative value insulates the path register's negativity from these increments and decrements.

A.3. Instrumenting Dummy Edges

Because TPP introduces additional dummy edges when loops are disconnected, it was necessary for us to add instrumentation to the dummy edges themselves. In contrast, when PP inserts dummy edges (for back edges or path truncation) it marks them as "uneditable." Their intuition was that, as dummy edges are not part of the original CFG, they cannot be instrumented [1]. PP avoids instrumenting dummy edges by placing a subset of them (exit dummy edges from truncations and all entry dummy edges) in the spanning tree used in selecting the instrumented edges [7]. This approach cannot be used in TPP without adding cycles to the spanning tree. To support instrumentation of dummy edges, TPP tracks the relationships between dummy edges and the removed edges so that instrumentation can be transferred. If an edge $a \rightarrow b$ was removed, then TPP places the instrumentation for the exit dummy edge ($a \rightarrow EXIT$) on the edge $a \rightarrow b$, followed by the instrumentation for the entry dummy edge ($ENTRY \rightarrow b$).