Ball-Larus Path Profiling Across Multiple Loop Iterations

Daniele Cono D'Elia
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome
delia@dis.uniroma1.it

Camil Demetrescu
Dept. of Computer, Control and Management Engineering
Sapienza University of Rome
demetres@dis.uniroma1.it

Abstract

Identifying the hottest paths in the control flow graph of a routine can direct optimizations to portions of the code where most resources are consumed. This powerful methodology, called *path profiling*, was introduced by Ball and Larus in the mid 90's [4] and has received considerable attention in the last 15 years for its practical relevance. A shortcoming of the Ball-Larus technique was the inability to profile cyclic paths, making it difficult to mine execution patterns that span multiple loop iterations. Previous results, based on rather complex algorithms, have attempted to circumvent this limitation at the price of significant performance losses even for a small number of iterations. In this paper, we present a new approach to multi-iteration path profiling, based on data structures built on top of the original Ball-Larus numbering technique. Our approach allows the profiling of *all executed paths* obtained as a concatenation of up to k Ball-Larus acyclic paths, where k is a user-defined parameter. We provide examples showing that this method can reveal optimization opportunities that acyclic-path profiling would miss. An extensive experimental investigation on a large variety of Java benchmarks on the Jikes RVM shows that our approach can be even faster than Ball-Larus due to fewer operations on smaller hash tables, producing compact representations of cyclic paths even for large values of k.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques; D.2.2 [*Software Engineering*]: Tools and Techniques—programmer workbench; D.2.5 [*Software Engineering*]: Testing and Debugging—diagnostics, tracing

General Terms Algorithms, Measurement, Performance.

Keywords Profiling, dynamic program analysis, instrumentation.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509521

1. Introduction

Path profiling is a powerful methodology for identifying performance bottlenecks in a program. The approach consists of associating performance metrics, usually frequency counters, to paths in the control flow graph. Identifying hot paths can direct optimizations to portions of the code that could yield significant speedups. For instance, trace scheduling can improve performance by increasing instruction-level parallelism along frequently executed paths [13, 22]. The seminal paper by Ball and Larus [4] introduced a simple and elegant path profiling technique. The main idea was to implicitly number all possible acyclic paths in the control flow graph so that each path is associated with a unique compact path identifier (ID). The authors showed that path IDs can be efficiently generated at runtime and can be used to update a table of frequency counters. Although in general the number of acyclic paths may grow exponentially with the graph size, in typical control flow graphs this number is usually small enough to fit in current machine word-sizes, making this approach very effective in practice.

While the original Ball-Larus approach was restricted to acyclic paths obtained by cutting paths at loop back edges, profiling paths that span consecutive loop iterations is a desirable, yet difficult, task that can yield better optimization opportunities. Consider, for instance, the problem of eliminating redundant executions of instructions, such as loads and stores [7], conditional jumps [6], expressions [9], and array bounds checks [8]. A typical situation is that the same instruction is redundantly executed at each loop iteration, which is particularly common for arithmetic expressions and load operations [7, 9]. To identify such redundancies, paths that extend across loop back edges need to be profiled. Another application is trace scheduling [22]: if a frequently executed cyclic path is found, compilers may unroll the loop and perform trace scheduling on the unrolled portion of code. Tallam *et al.* [20] provide a comprehensive discussion of the

benefits of multi-iteration path profiling. We provide further examples in Section 3.

Different authors have proposed techniques to profile cyclic paths by modifying the original Ball-Larus path numbering scheme in order to identify paths that extend across multiple loop iterations [17, 19, 20]. Unfortunately, all known solutions require rather complex algorithms that incur severe performance overheads even for short cyclic paths, leaving the interesting open question of finding simpler and more efficient alternative methods.

Contributions. In this paper, we present a novel approach to multi-iteration path profiling, which provides substantially better performance than previous techniques even for long paths. Our method stems from the observation that any cyclic execution path in the control flow graph of a routine can be described as a concatenation of Ball-Larus acyclic paths (BL paths). In particular, we show how to accurately profile *all executed paths* obtained as a concatenation of up to k BL paths, where k is a user-defined parameter. The main results of this paper can be summarized as follows:

1. we reduce multi-iteration path profiling to the problem of counting n-grams, i.e., contiguous sequences of n items from a given sequence. To compactly represent collected profiles, we organize them in a forest of prefix trees (or tries) [14] of depth up to k, where each node is labeled with a BL path, and paths in a tree represent concatenations of BL paths that were actually executed by the program, along with their frequencies. We also present an efficient construction algorithm based on a variant of the k-SF data structure presented in [3];
2. we discuss examples of profile-driven optimizations that can be achieved using our approach. In particular, we show how the profiles collected with our method can help to restructure the code of an image processing application, achieving speedups up to about 30% on an image filter based on convolution kernels;
3. to evaluate the effectiveness of our ideas, we developed a Java performance profiler in the Jikes Research Virtual Machine [1]. To make fair performance comparisons with state-of-the-art previous profilers, we built our code on top of the BLPP profiler developed by Bond [10, 15], which provides an efficient implementation of the Ball-Larus acyclic-path profiling technique. Our Java code was endorsed by the OOPSLA 2013 Artifact Evaluation Committee and is available on the Jikes RVM Research Archive. We also provide a C implementation of our profiler for manual source code instrumentation;
4. we performed a broad experimental study on a large suite of prominent Java benchmarks on the Jikes Research Virtual Machine, showing that our profiler can collect profiles that would have been too costly to gather using previous multi-iteration techniques.

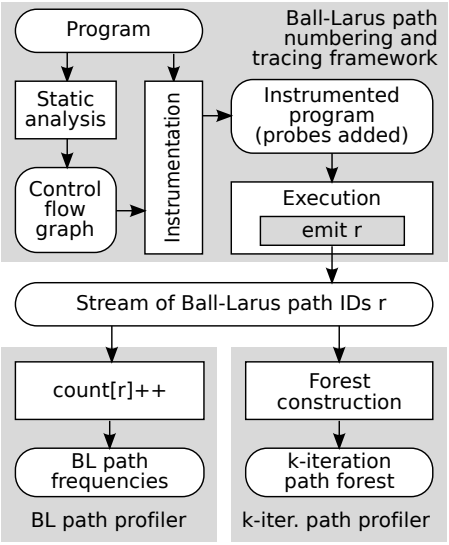


Figure 1: Overview of our approach: classical Ball-Larus profiling versus k-iteration path profiling, cast in a common framework.

Techniques. Differently from previous approaches [17, 19, 20], which rely on modifying the Ball-Larus path numbering to cope with cycles, our method does not require any modification of the original numbering technique described in [4]. The main idea behind our approach is to fully decouple the task of tracing Ball-Larus acyclic paths at run time from the task of concatenating and storing them in a data structure to keep track of multiple iterations. The decoupling is performed by letting the Ball-Larus profiling algorithm issue a stream of BL path IDs (see Figure 1), where each ID is generated when a back edge in the control flow graph is traversed or the current procedure is abandoned. As a consequence of this modular approach, our method can be implemented on top of existing Ball-Larus path profilers, making it simpler to code and maintain.

Our profiler introduces a technical shift based on a smooth blend of the path numbering methods used in intraprocedural path profiling with data structure-based techniques typically adopted in interprocedural profiling, such as calling-context profiling. The key to the efficiency of our approach is to replace costly hash table accesses, which are required by the Ball-Larus algorithm to maintain path counters for larger programs, with substantially faster operations on trees. With this idea, we can profile paths that extend across many loop iterations in comparable time, if not faster, than profiling acyclic paths on a large variety of industry-strength benchmarks.

Organization of the paper. In Section 2 we describe our approach and in Section 3 we discuss examples of possible performance optimizations based on the proposed technique. Section 4 describes the algorithmic ideas behind our tool and Section 5 provides implementation details. The results of

our experimental investigation are detailed in Section 6 and related work is surveyed in Section 7. Concluding remarks are given in Section 8.

2. Approach

In this section we provide an overview of our approach to multi-iteration path profiling. From a high-level point of view, illustrated in Figure 1, the entire process is divided into two main phases:

1. instrumentation and execution of the program to be profiled (top of Figure 1);
2. profiling of paths (bottom of Figure 1).

The first phase is almost identical to the original approach described in [4]. The target program is statically analyzed and a control flow graph (CFG) is constructed for each routine of interest. The CFG is used to instrument the original program by inserting probes, which allow paths to be traced at run time. When the program is executed, taken acyclic paths are identified using the inserted probes. The main difference with the Ball-Larus approach is that, instead of directly updating a frequency counters table here, we emit a stream of path IDs, which is passed along to the next stage of the process. This allows us to decouple the task of tracing taken paths from the task of profiling them.

The profiling phase can be either the original hash table-based method of [4] used to maintain BL path frequencies (bottom-left of Figure 1), or other approaches such as the one we propose, i.e., profiling concatenations of BL paths in a forest-based data structure (bottom-right of Figure 1). Different profiling methods can be therefore cast into a common framework, increasing flexibility and helping us make more accurate comparisons.

We start with a brief overview of the Ball-Larus path tracing technique, which we use as the first stage of our profiler.

2.1 Ball-Larus Path Tracing Algorithm

The Ball-Larus path profiling (BLPP) technique [4] identifies each acyclic path that is executed in a routine. Paths start on the method entry and terminate on the method exit. Since loops make the CFG cyclic, loop back edges are substituted by a pair of dummy edges: the first one from the method entry to the target of the loop back edge, and the second one from the source of the loop back edge to the method exit. After this (reversible) transformation, the CFG of a method becomes a DAG (directed acyclic graph) and acyclic paths can be enumerated.

The Ball-Larus path numbering algorithm, shown in Figure 2, assigns a value $val(e)$ to each edge e of the CFG such that, given N acyclic paths, the sum of the edge values along any entry-to-exit path is a unique numeric ID in [0, N-1]. A CFG example and the corresponding path IDs are shown in Figure 3: notice that there are eight distinct acyclic paths,

```
procedure bl_path_numbering():
 1: for each basic block v in reverse topological order do
 2:   if v is the exit block then
 3:     numPaths(v) ← 1
 4:   else
 5:     numPaths(v) ← 0
 6:     for each outgoing edge e = (v, w) do
 7:       val(e) = numPaths(v)
 8:       numPaths(v) += numPaths(w)
 9:     end for
10:   end if
11: end for
```

Figure 2: The Ball-Larus path numbering algorithm.

numbered from 0 to 7, starting either at the method's entry A, or at loop header B (target of back edge (E, B)).

BLPP places instrumentation on edges to compute a unique path number for each possible path. In particular, it uses a variable `r`, called *probe* or *path register*, to compute the path number. Variable `r` is first initialized to zero upon method entry and then is updated as edges are traversed. When an edge that reaches the method exit is executed, or a back edge is traversed, variable `r` represents the unique ID of the taken path. As observed, instead of using the path ID `r` to increase the path frequency counter (`count[r]++`), we defer the profiling stage by emitting the path ID to an output stream (`emit r`). To support profiling over multiple invocations of the same routine, we annotate the stream with the special marker $*$ to denote a routine entry event. Instrumentation code for our CFG example is shown on the left of Figure 3.

2.2 k-Iteration Path Profiling

The second stage of our profiler takes as input the stream of BL path IDs generated by the first stage and uses it to build a data structure that keeps track of the frequencies of *each and every distinct taken path* consisting of the concatenation of up to k BL paths, where k is a user-defined parameter. This problem is equivalent to counting all n-grams, i.e., contiguous sequences of n items from a given sequence of items, for each $n \leq k$. Our solution is based on the notion of *prefix forest*, which compactly encodes a list of sequences by representing repetitions and common prefixes only once. A prefix forest can be defined as follows:

Definition 1 (Prefix forest)**.** *Let $L = \langle x_1, x_2, \ldots, x_q \rangle$ be any list of finite-length sequences over an alphabet H. The* prefix forest *$\mathcal{F}(L)$ of L is the smallest labeled forest such that, $\forall$ sequence $x = \langle a_1, a_2, \ldots, a_n \rangle$ in L there is a path $\pi = \langle \nu_1, \nu_2, \ldots, \nu_n \rangle$ in $\mathcal{F}(L)$ where ν_1 is a root and $\forall j \in [1, n]$:*

1. *ν_j is labeled with a_j, i.e., $\ell(\nu_j) = a_j \in H$;*
2. *ν_j has an associated counter $c(\nu_j)$ that counts the number of times sequence $\langle a_1, a_2, \ldots, a_j \rangle$ occurs in L.*

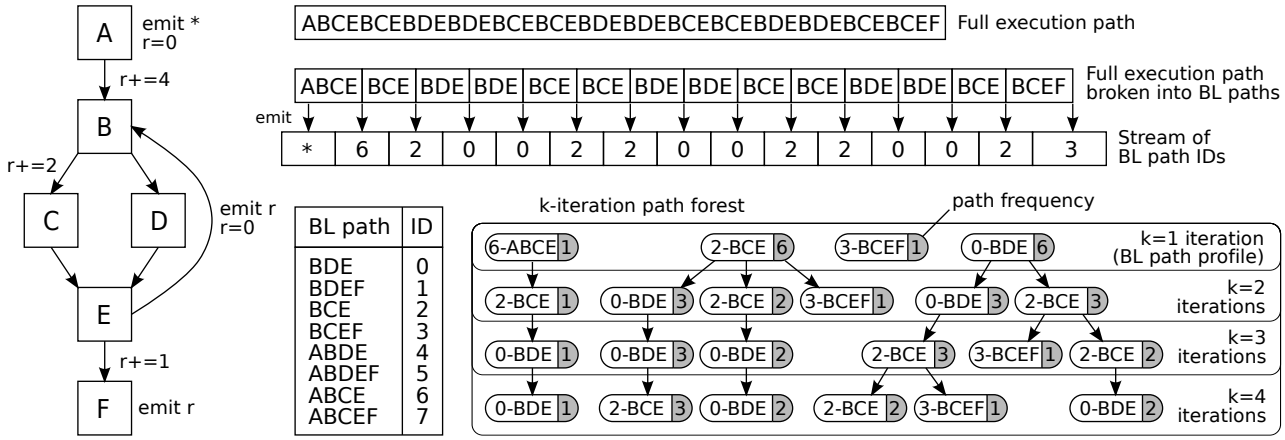


Figure 3: Control flow graph with Ball-Larus instrumentation modified to emit acyclic path IDs to an output stream and running example of our approach that shows a 4-iteration path forest (4-IPF) for a possible small execution trace.

Notice that L is a list and not a set to account for multiple occurrences of the same n-grams. By Definition 1, each sequence in L is represented as a path in the forest, and node labels in the path are exactly the symbols of the sequence, in the same order. The notion of minimality implies that, by removing even one single node, there would be at least one sequence of L not counted in the forest. Observe that there is a distinct root in the forest for each distinct symbol that occurs as first symbol of a sequence.

k-iteration path forest. The output of our profiler is a prefix forest, which we call *k-Iteration Path Forest* (k-IPF), that compactly represents all observed contiguous sequences of up to k BL path IDs:

Definition 2 (k-Iteration Path Forest)**.** *Given an input stream Σ representing a sequence of BL path IDs and $*$ markers, the* k-Iteration Path Forest *(k-IPF) of Σ is defined as k-IPF $= \mathcal{F}(L)$, where $L = \{$ list of all n-grams of Σ that do not contain $*$, with $n \leq k \}$.*

By Definition 2, the k-IPF is the prefix forest of all consecutive subsequences of up to k BL path IDs in Σ.

Example 1. *Figure 3 provides an example showing the 4-IPF constructed for a small sample execution trace consisting of a sequence of 44 basic blocks encountered during one invocation of the routine described by the control flow graph on the left. Notice that the full (cyclic) execution path starts at the entry basic block A and terminates on the exit basic block F. The first stage of our profiler issues a stream Σ of BL path IDs that are obtained by emitting the value of the probe register r each time a back edge is traversed, or the exit basic block is executed. Observe that the sequence of emitted path IDs induces a partition of the execution path into Ball-Larus acyclic paths. Hence, the sequence of executed basic blocks can be fully reconstructed from the sequence Σ of path IDs.*

The 4-IPF built in the second stage contains exactly one tree for each of the 4 distinct BL path IDs (0, 2, 3, 6) that occur in the stream. We observe that path frequencies in the first level of the 4-IPF are exactly those that traditional Ball-Larus profiling would collect. The second level contains the frequencies of taken paths obtained by concatenating 2 BL paths, etc. Notice that the path labeled with $\langle 2, 0, 0, 2 \rangle$ in the 4-IPF, which corresponds to the path $\langle B, C, E, B, D, E, B, D, E, B, C, E \rangle$ in the control flow graph, is a 4-gram that occurs 3 times in Σ and is one of the most frequent paths among those that span from 2 up to 4 loop iterations.

Properties. A k-IPF has some relevant properties:

1. $\forall$ nodes $\alpha \in k$-IPF, $k > 0$:

$$c(\alpha) \geq \sum_{\beta_i \,:\, edge\,(\alpha,\beta_i) \in\, k\text{-IPF}} c(\beta_i);$$

2. $\forall k > 0$, k-IPF $\subseteq (k+1)$-IPF.

By Property 1, since path counters are non-negative, they are monotonically non-increasing as we walk down the tree. The inequality $\geq$ in Property 1 may be strict ($>$) if the execution trace of a routine invocation does not end at the exit basic block; this may be the case when a subroutine call is performed at an internal node of the CFG.

Property 2 implies that, for each tree T_1 in the k-IPF there is a tree T_2 in the $(k+1)$-IPF such that T_2 is equal to T_1 after removing leaves at level $k+1$. Notice that a 1-IPF includes only acyclic paths and yields exactly the same counters as a Ball-Larus profiler [4].

3. Application Examples

In this section we consider examples of applications where k-iteration path profiling can reveal optimization opportunities or help developers comprehend relevant properties of

```
#define NEIGHBOR(m,i,dy,dx,w)                    \
    (*((m)+(i)+(dy)*(w)+(dx)))

#define CONVOLUTION(i) do {                      \
    val  = NEIGHBOR(img_in, (i),                 \
                    -2, -2, cols)*filter[0];     \
    val += NEIGHBOR(img_in, (i),                 \
                    -2, -1, cols)*filter[1];     \
    ...
    val += NEIGHBOR(img_in, (i),
                    +2, +2, cols)*filter[24];    \
    val = val*factor+bias;                       \
    img_out[i] = (unsigned char)                 \
       (val < 0 ? 0 : val > 255 ? 255 : val);    \
} while(0)

void filter_conv(unsigned char* img_in,
                 unsigned char* img_out,
                 unsigned char* mask,
                 char filter[25],
                 double factor, double bias,
                 int rows, int cols) {
    int val;
    long n = rows*cols, i;

    for (i = 0; i < n; i++)
        if (mask[i]) img_out[i] = img_in[i];
        else CONVOLUTION(i);
}
```

Figure 4: Masked image filtering code based on a convolution matrix.

a piece of software by identifying structured execution patterns that would be missed by an acyclic-path profiler. Our discussion is based on idealized examples found in real programs of the kind of behavior that can be exploited using multi-iteration path profiles. Since our methodology can be applied to different languages, we addressed both Java and C applications[1].

3.1 Masked Convolution Filters in Image Processing

As a first example, we consider a classic application of convolution filters to image processing, addressing the problem of masked filtering that arises when the user applies a transformation to a collection of arbitrary-shaped subregions of the input image. A common scenario is face anonymization, illustrated in the example of Figure 5. The case study discussed in this section shows that k-iteration path profiling with large values of k can identify regular patterns spanning multiple loop iterations that can be effectively exploited to speed up the code.

Figure 4 shows a C implementation of a masked image filtering algorithm based on a 5×5 convolution matrix[2]. The function takes as input a grayscale input image (8-bit depth) and a black & white mask image that specifies the regions of the image to be filtered. Figure 5 shows a sample input image (left), a mask image (center), and the output image (right) generated by the code of Figure 4 by applying a blur filter to the regions of the input image specified by the mask. Notice that the filter code iterates over all pixels of the input image and for each pixel checks if the corresponding mask is black (zero) or white (non-zero, i.e., 255). If the mask is white, the original grayscale value is copied from the input image to the output image; otherwise, the grayscale value of the output pixel is computed by applying the convolution kernel to the neighborhood of the current pixel in the input image. To avoid expensive boundary checks in the convolution operation, the mask is preliminarily cropped so that all values near the border are white (this operation takes negligible time).

Figure 6 shows a portion of the 10-IPF forest containing the trees rooted at the BL path IDs that correspond to: the path entering the loop (ID=0), the copy branch taken in the loop body when the mask is non-zero (ID=1), and the convolution branch taken in the loop body when the mask is zero (ID=2). The 10-IPF was generated on the workload of Figure 5 and was pruned by removing all nodes whose counters are less than 0.01% of the counter of their parents or less than 0.01% of the counter of their roots. For each node v in the forest, if v has a counter that is $X\%$ of the counter of its parent and is $Y\%$ of the counter of the root, then the edge leading to v is labeled with "$X\%(Y\%)$". A visual analysis of the forest shows that:

- the copy branch (1) is more frequent than the convolution branch (2);
- 98.9% of the times a copy branch (1) is taken, it is repeated consecutively at least 10 times, and only 0.1% of the times is immediately followed by a convolution branch;
- 95% of the times a convolution branch (2) is taken, it is repeated consecutively at least 10 times, and only 0.6% of the times is immediately followed by a copy branch;

This entails that both the copy and the convolution operations are repeated along long consecutive runs. The above properties are typical of masks used in face anonymization and other common image manipulations based on user-defined selections of portions of the image. The collected profiles suggest that consecutive iterations of the same branches may be selectively unrolled as shown in Figure 7. Each iteration of the outer loop, designed for a 64-bit plat-

[1] We profiled Java programs using the k-BLPP tool described in Section 5 and C programs with manual source code instrumentation based on a C implementation of the algorithms and data structures of Section 4, available at `http://www.dis.uniroma1.it/~demetres/kstream/`.

[2] The source code of our example is provided at `http://www.dis.uniroma1.it/~demetres/kstream/`.

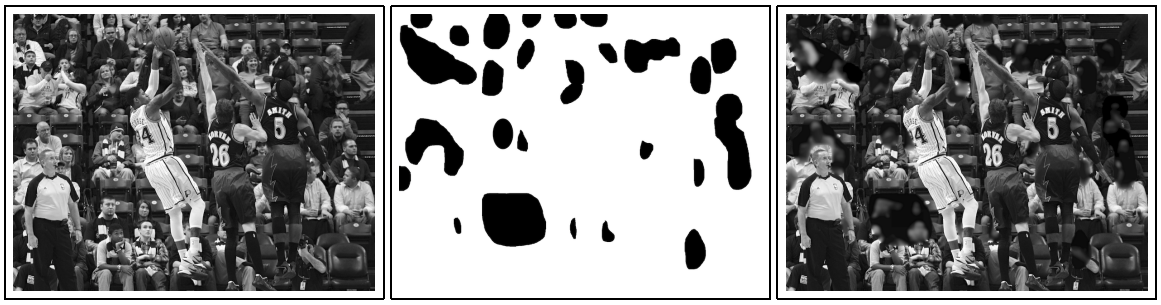


Figure 5: Masked blur filter example: original 3114×2376 image (left), filter mask (center), filtered image (right).

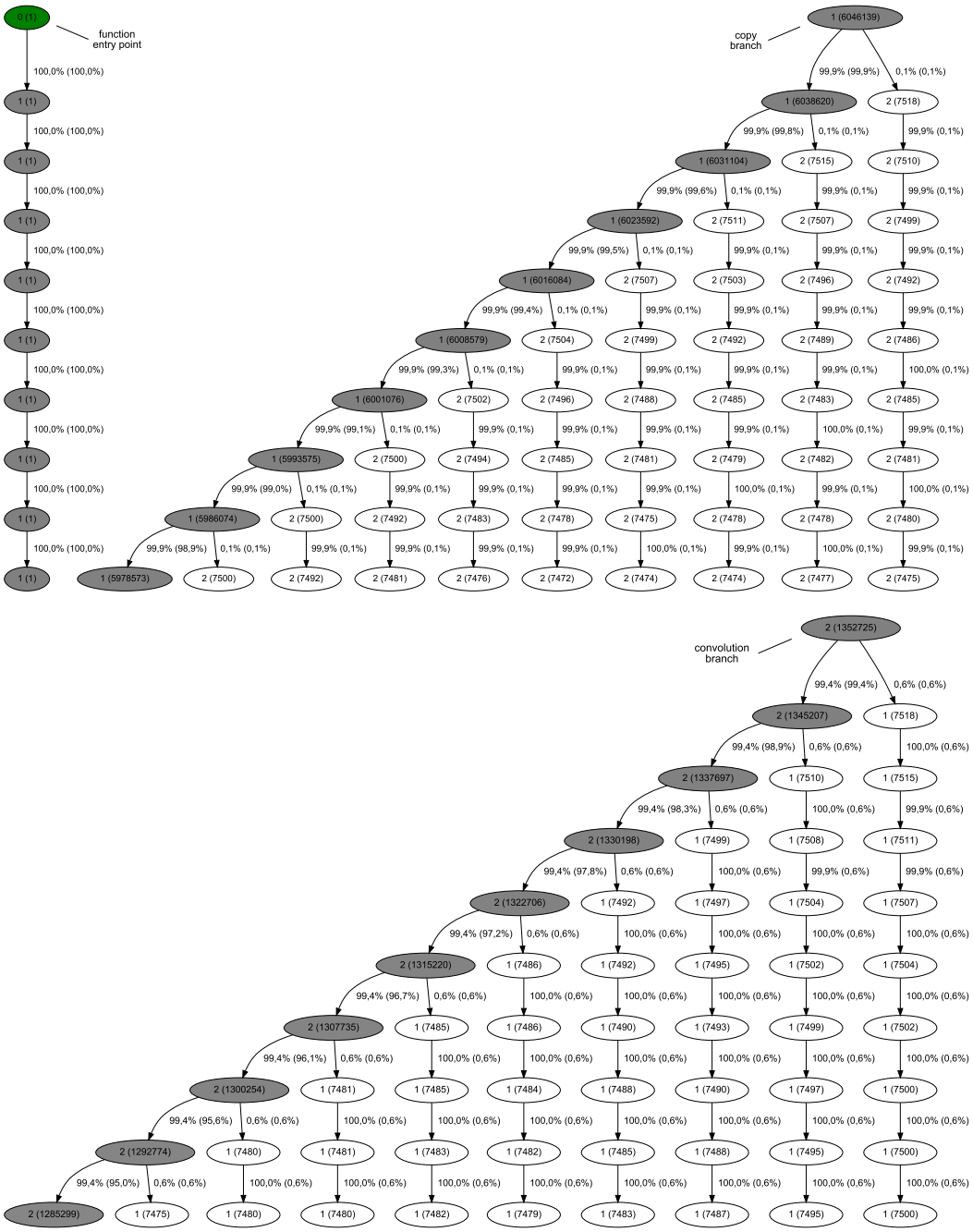


Figure 6: 10-IPF forest of the code of Figure 4 on the workload of Figure 5.

```
for (i = 0; i < n-7; i += 8) {

   if (*(long*)(mask+i) == 0xFFFFFFFFFFFFFFFF)
      *(long*)(img_out+i) = *(long*)(img_in+i);

   else if (*(long*)(mask+i) == 0) {
      CONVOLUTION(i);
      CONVOLUTION(i+1);
      CONVOLUTION(i+2);
      CONVOLUTION(i+3);
      CONVOLUTION(i+4);
      CONVOLUTION(i+5);
      CONVOLUTION(i+6);
      CONVOLUTION(i+7);
   }

   else for (j = i; j < i+8; j++)
      if (mask[j]) img_out[j] = img_in[j];
      else CONVOLUTION(j);
}
```

Figure 7: Optimized 64-bit version of the loop of Figure 4.

form, works on 8 (rather than 1) pixels at a time. Three cases are possible:

1. the next 8 mask entries are all 255 (white): the 8 corresponding input pixel values are copied to the output image at once with a single assignment instruction;
2. the next 8 mask entries are all 0 (black): the kernel is applied sequentially to each of the next 8 input pixels;
3. the next 8 mask entries are mixed: an inner loop performs either copy or convolution on the corresponding pixels.

Performance analysis. To assess the benefits of the optimization performed in Figure 7, we conducted several tests on recent commodity platforms (Intel Core 2 Duo, Intel Core i7, Linux and MacOS X, 32 and 64 bits, `gcc -O3`), considering a variety of sample images and masks with regions of different sizes and shapes. We obtained non-negligible speedups on all our tests, with a peak of about 21% on the workload of Figure 5 (3114×2376 pixels) and about 30% on a larger 9265×7549 image with a memory footprint of about 200 MB. In general, the higher the white entries in the mask, the faster the code, with larger speedups on more recent machines. As we expected, for entirely black masks the speedup was instead barely noticeable: this is due to the fact that the convolution operations are computationally demanding and tend to hide the benefits of loop unrolling.

Discussion. The example discussed in this section shows that both the ability to profile paths across multiple iterations, and the possibility to handle large values of k played a crucial role in optimizing the code. Indeed, acyclic-path profiling would count the number of times each branch is taken, but would not reveal that they appear in consecutive runs. Moreover, previous multi-iteration approaches that only handle very small values of k would not capture the long runs that make the proposed optimization effective.

For the example of Figure 5, an acyclic-path profile would indicate that the copy branch is taken 81.7% of the times, but not how branches are interleaved. From this information, we would be able to deduce that the average length of a sequence of consecutive white values in the mask is ≥ 4. Our profile shows that, 98.9% of the times the actual length is at least 10, fully justifying our optimization that copies 8 bytes at a time. The advantage of k-iteration path profiling increases for masks with a more balanced ratio between white and black pixels: for a 50-50 ratio, an acyclic-path profile would indicate that the average length of consecutive white/black runs is ≥ 1, yielding no useful information for loop unrolling purposes.

The execution pattern where the same branches are repeatedly taken over consecutive loop iterations is common to several other applications, which may benefit from optimizations that take advantage of long repeated runs. For instance, the `LBM_performStreamCollide` function of the `lbm` benchmark included in the SPEC CPU 2006 suite iterates over a 3D domain, simulating incompressible fluid dynamics based on the Lattice Boltzmann Method. An input geometry file specifies obstacles that determine a steady state solution. The loop contains branches that depend upon the currently scanned cell, which alternates between obstacles and void regions of the domain, producing a k-IPF similar to that of Figure 6 on typical workloads.

3.2 Instruction Scheduling

Young and Smith [22] have shown that path profiles spanning multiple loop iterations can be used to improve the construction of superblocks in trace schedulers.

Global instruction scheduling groups and orders the instructions of a program in order to match the hardware resource constraints when they are fetched. In particular, trace schedulers rely on the identification of traces (i.e., sequences of basic blocks) that are frequently executed. These traces are then extended by appending extra copies of likely successors blocks, in order to form a larger pool of instructions for reordering. A trace that is likely to complete is clearly preferable, since instructions moved before an early exit point are wasted work.

Superblocks are defined as sequences of basic blocks with a single entry point and multiple exit points; they are useful for maintaining the original program semantics during a global code motion. Superblock formation is usually driven by edge profiles: however, path profiles usually provide better information to determine which traces are worthwhile to enlarge (i.e., those for which execution reaches the ending block most of the times). Figure 8 shows how superblock construction may benefit from path profiling information for

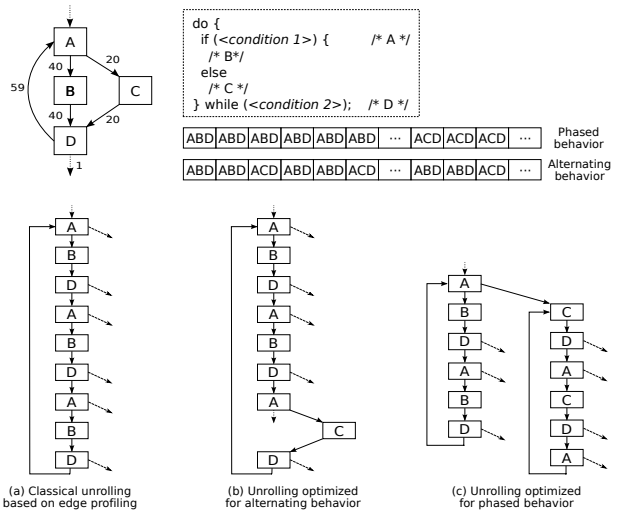


Figure 8: Superblock construction using cyclic-path profiles.

two different behaviors, characterized by the same edge profile, of a `do ... while` loop.

Path profiling techniques that do not span multiple loop iterations chop execution traces into pieces separated at back edges, hence the authors collect execution frequencies for *general paths* [23], which contain any contiguous sequences of CFG edges up to a limiting path length; they use a path length of 15 branches in the experiments.

Example. Phased and alternating behaviors as in Figure 8 are quite common among many applications, thus offering interesting optimization opportunities. For instance, the convolution filter discussed in the previous section is a clear example of phased behavior. An alternating behavior is shown by the `checkTaskTag` method of class `Scanner` in the `org.eclipse.jdt.internal.compiler.parser` package of the `eclipse` benchmark included in the `DaCapo` release 2006-10-MR2. In Figure 9 we show a subtree of the 11-IPF generated for this method; in the subtree, we pruned all nodes with counters less than 10% of the counter of the root. Notice that, after executing the BL path with ID 38, in 66% of the executions the program continues with 86, and in 28% of the executions with BL path 87. When 86 follows 38, in 100% of the executions the control flow takes the path $\langle 86, 86, 86, 755\rangle$, which spans four loop iterations and may be successfully unrolled to perform instruction scheduling. Interestingly, sequence $\langle 38, 86, 86, 86, 755, 38, 86, 86, 86, 755, 38\rangle$ of 11 BL path IDs, highlighted in Figure 9, accounts for more than 50% of all executions of the first BL path in the sequence, showing that sequence $\langle 38, 86, 86, 86, 755\rangle$ is likely to be repeated consecutively more than once.

Discussion. The work presented in [22] focused on assessing the benefits of using general paths for global instruction scheduling, rather than on how to profile them. As we will see in Section 7, compared to our approach the technique

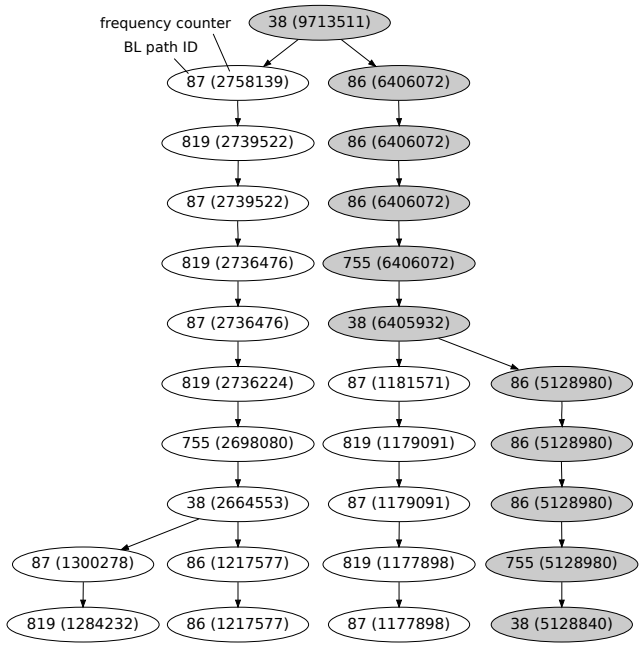


Figure 9: Subtree of the 11-IPF of method `org.eclipse.jdt.internal.compiler.parser.Scanner.checkTaskTag` taken from release 2006-10-MR2 of the `DaCapo` benchmark suite.

proposed by Young [23] for profiling general paths scales poorly for increasing path lengths both in terms of space usage and running time. We believe that our method, by substantially reducing the overhead of cyclic-path profiling, has the potential to provide a useful ingredient for making profile-guided global instruction scheduling more efficient in modern compilers.

4. Algorithms

In this section, we show how to efficiently construct a k-IPF profile starting from a stream of BL path IDs. We observe that building explicitly a k-IPF concurrently with program execution would require updating up to k nodes for each stream item: this may considerably slow down the program even for small values of k. To overcome this problem, we construct an intermediate data structure that can be updated quickly, and then convert it into a k-IPF more efficiently when the stream is over. As intermediate data structure, we use a variant of the *k-slab forest* (k-SF) introduced in [3].

Main idea. The variant of the k-SF we present in this paper is tailored to keep track of all n-grams of a sequence of symbols, for all $n \leq k$. The organization of our data structure stems from the following simple observation: if we partition a sequence into chunks of length $k-1$, then any subsequence of length up to k will be entirely contained within two consecutive chunks of the partition. The main idea is therefore to consider all subsequences that start at the beginning of a chunk and terminate somewhere in the next chunk, and join them in a prefix forest (the k-SF). Such a

forest contains a distinct tree for each distinct symbol that appears at the beginning of any chunk and, as we will see later on in this section, encodes information about all n-grams with $n \leq k$. Indeed, by Definition 1, node counters in the k-SF keep track of the number of occurrences of each distinct subsequence of length between k and $2k-2$ that starts at the beginning of a chunk. The number of occurrences of a given n-gram can be reconstructed off-line by finding all subpaths of length n of the k-SF labeled with the symbols of the n-gram and by summing up the counters of the end-nodes of the subpaths. The partition of the sequence into chunks induces a division of the forest into upper and lower regions (*slabs*) of height up to $k-1$. This organization implies that the k-SF can be constructed on-line as stream items are revealed to the profiler by adding or updating up to two nodes of the forest at a time (one in the upper region and one in the lower region), instead of k nodes as we would do if we incremented explicitly the frequencies of n-grams as soon as they are encountered in the stream.

The following example applies the concepts described above to a stream of BL path IDs.

Example 2. *Let us consider again the example given in Figure 3. For $k = 4$, we can partition the stream into maximal chunks of up to $k-1 = 3$ consecutive BL path IDs as follows:*

$$\Sigma = \langle *, \underbrace{\boxed{6,\ 2,\ 0}}_{c_1}, \underbrace{\boxed{0,\ 2,\ 2}}_{c_2}, \underbrace{\boxed{0,\ 0,\ 2}}_{c_3}, \underbrace{\boxed{2,\ 0,\ 0}}_{c_4}, \underbrace{\boxed{2,\ 3}}_{c_5} \rangle.$$

The 4-SF of Σ, defined in terms of chunks $c_1, \ldots, c_5$, is shown in Figure 10. Notice for instance that 2-gram $\langle 0, 0 \rangle$ occurs three times in Σ and five times in the 4-SF. However, only three of them end in the bottom slab and hence are counted in the frequency counters.

To obtain a k-IPF starting from the k-SF, for each BL path ID that appears in the stream we will eventually construct the set of nodes in the k-SF associated with it and join the subsequences of length up to k starting from those nodes into a prefix forest.

Formal k-SF definition. The above description of the k-SF can be more formally and precisely summarized as follows:

Definition 3 (k-slab forest)**.** *Let $k \geq 2$ and let $c_1, c_2, c_3, \ldots, c_m$ be the chunks of Σ obtained by: (1) splitting Σ at $*$ markers, (2) removing the markers, and (3) cutting the remaining subsequences every $k-1$ consecutive items. The k-slab forest (k-SF) of Σ is defined as k-SF $= \mathcal{F}(L)$, where $L = \{$list of all prefixes of $c_1 \cdot c_2$ and all prefixes of length $\geq k$ of $c_i \cdot c_{i+1}$, $\forall i \in [2, m-1]\}$ and $c_i \cdot c_{i+1}$ denotes the concatenation of c_i and c_{i+1}.*

By Definition 3, since each chunk c_i has length up to $k-1$, then a k-SF has at most $2k-2$ levels and depth $2k-3$. As observed above, the correctness of the k-SF representation

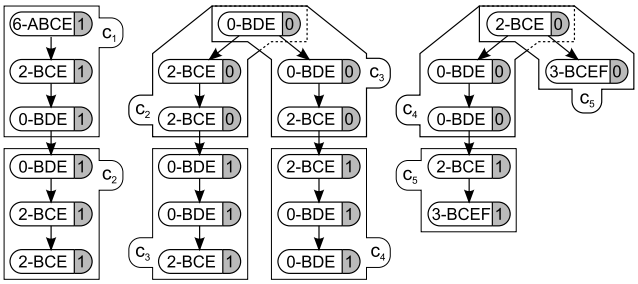


Figure 10: 4-SF resulting from the execution trace of Figure 3.

stems from the fact that, since each occurrence of an n-gram with $n \leq k$ appears in $c_i \cdot c_{i+1}$ for some i, then there is a tree in the k-SF representing it.

Example 3. *In accordance with Definition 3, the forest of Figure 10 for the stream of Example 2 is $\mathcal{F}(L)$, where $L = \langle \langle 6 \rangle, \langle 6, 2 \rangle, \langle 6, 2, 0 \rangle, \langle 6, 2, 0, 0 \rangle, \langle 6, 2, 0, 0, 2 \rangle, \langle 6, 2, 0, 0, 2, 2 \rangle, \langle 0, 2, 2, 0 \rangle, \langle 0, 2, 2, 0, 0 \rangle, \langle 0, 2, 2, 0, 0, 2 \rangle, \langle 0, 0, 2, 2 \rangle, \langle 0, 0, 2, 2, 0 \rangle, \langle 0, 0, 2, 2, 0, 0 \rangle, \langle 2, 0, 0, 2 \rangle, \langle 2, 0, 0, 2, 3 \rangle \rangle$.*

k-SF construction algorithm. Given a stream Σ formed by $*$ markers and BL path IDs, the k-SF of Σ can be constructed by calling the procedure `process_bl_path_id`(r) shown in Figure 11 on each item r of Σ. The streaming algorithm, which is a variant of the k-SF construction algoritm given in [3] for the different setting of bounded-length calling contexts, keeps the following information:

- a hash table R, initially empty, containing pointers to the roots of trees in the k-SF, hashed by node labels; since no two roots have the same label, the lookup operation `find`(R, r) returns the pointer to the root containing label r, or `null` if no such root exists;
- a variable n that counts the number of BL path IDs processed since the last $*$ marker;
- a variable τ (top) that points either to `null` or to the current k-SF node in the upper part of the forest (levels 0 through $k-2$);
- a variable β (bottom) that points either to `null` or to the current k-SF node in the lower part of the forest (levels $k-1$ through $2k-3$).

The main idea of the algorithm is to progressively add new paths to an initially empty k-SF. The path formed by the first $k-1$ items since the last $*$ marker is added to one tree of the upper part of the forest. Each later item r is added at up to two different locations of the k-SF: one in the upper part of the forest (lines 13–17) as a child of node τ (if no child of τ labeled with r already exists), and the other one in the lower part of the forest (lines 21–25) as a child of node β (if no child of β labeled with r already exists). Counters of processed nodes already containing r are incremented by one (either line 27 or line 29).

```
procedure process_bl_path_id(r):
 1: if r = * then
 2:     n ← 0
 3:     τ ← null
 4:     return
 5: end if
 6: if n mod (k − 1) = 0 then
 7:     β ← τ
 8:     τ ← find(R, r)
 9:     if τ = null then
10:         add root τ with ℓ(τ) = r and c(τ) = 0 to k-SF and R
11:     end if
12: else
13:     find child ω of node τ with label ℓ(ω) = r
14:     if ω = null then
15:         add node ω with ℓ(ω) = r and c(ω) = 0 to k-SF
16:         add arc (τ, ω) to k-SF
17:     end if
18:     τ ← ω
19: end if
20: if β ≠ null then
21:     find child υ of node β with label ℓ(υ) = r
22:     if υ = null then
23:         add node υ with ℓ(υ) = r and c(υ) = 0 to k-SF
24:         add arc (β, υ) to k-SF
25:     end if
26:     β ← υ
27:     c(β) ← c(β) + 1
28: else
29:     c(τ) ← c(τ) + 1
30: end if
31: n ← n + 1
```

Figure 11: Streaming algorithm for k-SF construction.

```
procedure make_k_ipf():
 1: I ← ∅
 2: for each node ρ ∈ k-SF do
 3:     if ℓ(ρ) ∉ I then
 4:         add ℓ(ρ) to I and let s(ℓ(ρ)) ← ∅
 5:     end if
 6:     add ρ to s(ℓ(ρ))
 7: end for
 8: let the k-IPF be formed by a dummy root φ
 9: for each r ∈ I do
10:     for each ρ ∈ s(r) do
11:         join_subtree(ρ, φ, k)
12:     end for
13: end for
14: remove dummy root φ from the k-IPF

procedure join_subtree(ρ, γ, d):
 1: δ ← child of γ in the k-IPF s.t. ℓ(δ) = ℓ(ρ)
 2: if δ = null then
 3:     add new node δ as a child of γ in the k-IPF
 4:     ℓ(δ) ← ℓ(ρ) and c(δ) ← c(ρ)
 5: else
 6:     c(δ) ← c(δ) + c(ρ)
 7: end if
 8: if d > 1 then
 9:     for each child σ of ρ in the k-SF do
10:         join_subtree(σ, δ, d − 1)
11:     end for
12: end if
```

Figure 12: Algorithm for converting a k-SF into a k-IPF.

Both τ and β are updated to point to the child labeled with r (lines 18 and 26, respectively). The running time of the algorithm is dominated by lines 8 and 10 (hash table accesses), and by lines 13 and 21 (node children scan). Assuming that operations on R require constant time, the per-item processing time is $O(\delta)$, where δ is the maximum degree of a node in the k-SF. Our experiments revealed that δ is on average a typically small constant value.

As an informal proof that each subsequence of length up to k is counted exactly once in the k-SF, we first observe that, if the subsequence extends across two consecutive chunks, then it appears exactly once in the forest (connecting a node in the upper slab to a node in the lower slab). In contrast, if the subsequence is entirely contained in a chunk, then it appears twice: once in the upper slab of the tree rooted at the beginning of the chunk, and once in the lower slab rooted in at the beginning of the preceding chunk. However, only the counter in the lower part of the forest is updated (line 27): for this reason, the sum of all counters in the k-SF is equal to the length of the stream.

k-SF to k-IPF conversion. Once the stream Σ is over, i.e., the profiled thread has terminated, we convert the k-SF into a k-IPF using the procedure `make_k_ipf` shown in Figure 12. The key intuition behind the correctness of the conversion algorithm is that for each sequence in the stream of length up to k, there is a tree in the k-SF containing it.

The algorithm creates a set I of all distinct path IDs that occur in the k-SF and for each r in I builds a set $s(r)$ containing all nodes ρ of the k-SF labeled with r (lines 2–7). To build the k-IPF, the algorithm lists each distinct path ID r and joins to the k-IPF the top $k-1$ levels of the subtrees of the k-SF rooted at the nodes in $s(r)$. Subtrees are added as children of a dummy root, which is inserted for the sake of convenience and then removed. The join operation is specified by procedure `join_subtree`, which performs a traversal of a subtree of the k-SF of depth less than k and adds nodes to k-IPF so that all labeled paths in the subtree appear in the k-IPF as well, but only once. Path counters in the k-SF are accumulated in the corresponding nodes of the k-IPF to keep track of the number of times each distinct path consisting of the concatenation of up to k BL paths was taken by the profiled program.

5. Implementation

In this section we describe the implementation of our profiler, which we call k-BLPP, in the Jikes Research Virtual Machine [1].

5.1 Adaptive Compilation

The Jikes RVM is a high-performance metacircular virtual machine: unlike most other JVMs, it is written in Java. Jikes RVM does not include an interpreter: all bytecode must be first translated into native machine code. The unit of compilation is the method, and methods are compiled lazily by a fast non-optimizing compiler – the so-called *baseline* compiler – when they are first invoked by the program. As execution continues, the Adaptive Optimization System monitors program execution to detect program hot spots and selectively recompiles them with three increasing levels of optimization. Note that all modern production JVMs rely on some variant of selective optimizing compilation to target the subset of the hottest program methods where they are expected to yield the most benefits.

Recompilation is performed by the *optimizing* compiler, that generates higher-quality code but at a significantly larger cost than the baseline compiler. Since Jikes RVM quickly recompiles frequently executed methods, we implemented k-BLPP in the optimizing compiler only.

5.2 Inserting Instrumentation on Edges

k-BLPP adds instrumentation to hot methods in three passes:

1. building the DAG representation;
2. assigning values to edges;
3. adding instrumentation to edges.

k-BLPP adopts the *smart path numbering* algorithm proposed by Bond and McKinley [11] to improve performance by placing instrumentation on cold edges. In particular, line 6 of the canonical Ball-Larus path numbering algorithm shown in Figure 2 is modified such that outgoing edges are picked in decreasing order of execution frequency. For each basic block edges are sorted using existing edge profiling information collected by the baseline compiler, thus allowing us to assign zero to the hottest edge so that k-BLPP does not place any instrumentation on it.

During compilation, the Jikes RVM generates *yield points*, which are program points where the running thread determines if it should yield to another thread. Since JVMs need to gain control of threads quickly, compilers insert yield points in method prologues, loop headers, and method epilogues. We modified the optimizing compiler to also store the path profiling probe on loop headers and method epilogues. Ending paths at loop headers rather than back edges causes a path that traverse a header to be split into two paths: this difference from canonical Ball-Larus path profiling is minor because it only affects the first path through a loop [10].

Note that optimizing compilers do not insert yield points in a method when either it does not contain branches (hence its profile is trivial) or it is marked as uninterruptible. The second case occurs in internal Jikes RVM methods only; the compiler occasionally inlines such a method into an application method, and this might result in a loss of information only when the execution reaches a loop header contained in the inlined method. However, this loss of information appears to be negligible [10].

5.3 Path Profiling

The k-SF construction algorithm described in Section 2.2 is implemented using a standard first-child, next-sibling representation for nodes. This representation is very space-efficient, as it requires only two pointers per node: one to its leftmost child and the other to its right nearest sibling.

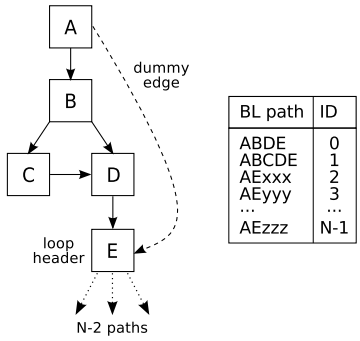


Figure 13: Routine with an initial branch before the first cycle.

Tree roots are stored and accessed through an efficient implementation[3] of a hash map, using the pair represented by the Ball-Larus path ID and the unique identifier associated to the current routine (i.e., the compiled method ID) as key. Note that this map is typically smaller than a map required by a traditional BLPP profiler, since tree roots represent only a fraction of the distinct path IDs encountered during the execution. Consider, for instance, the example shown in Figure 13: this control flow graph has N acylic paths after backedges have been removed. Since cyclic paths are truncated on loop headers, only path IDs 0 and 1 can appear after the special marker $*$ in the stream, thus leading to the creation of an entry in the hash map. Additional entries might be created when a new tree is added to the k-SF (line 10 of the streaming algorithm shown in Figure 11); however, experimental results show that the number of tree roots is usually small, while N increases with the complexity (i.e., number of branches and loops) of the routine.

6. Experimental Evaluation

In this section we report the result of an extensive experimental evaluation of our approach. The goal is to assess

[3] `HashMapRVM` is the stripped-down implementation of the `HashMap` data structure used by core parts of the Jikes RVM runtime and by Bond's BLPP path profiler.

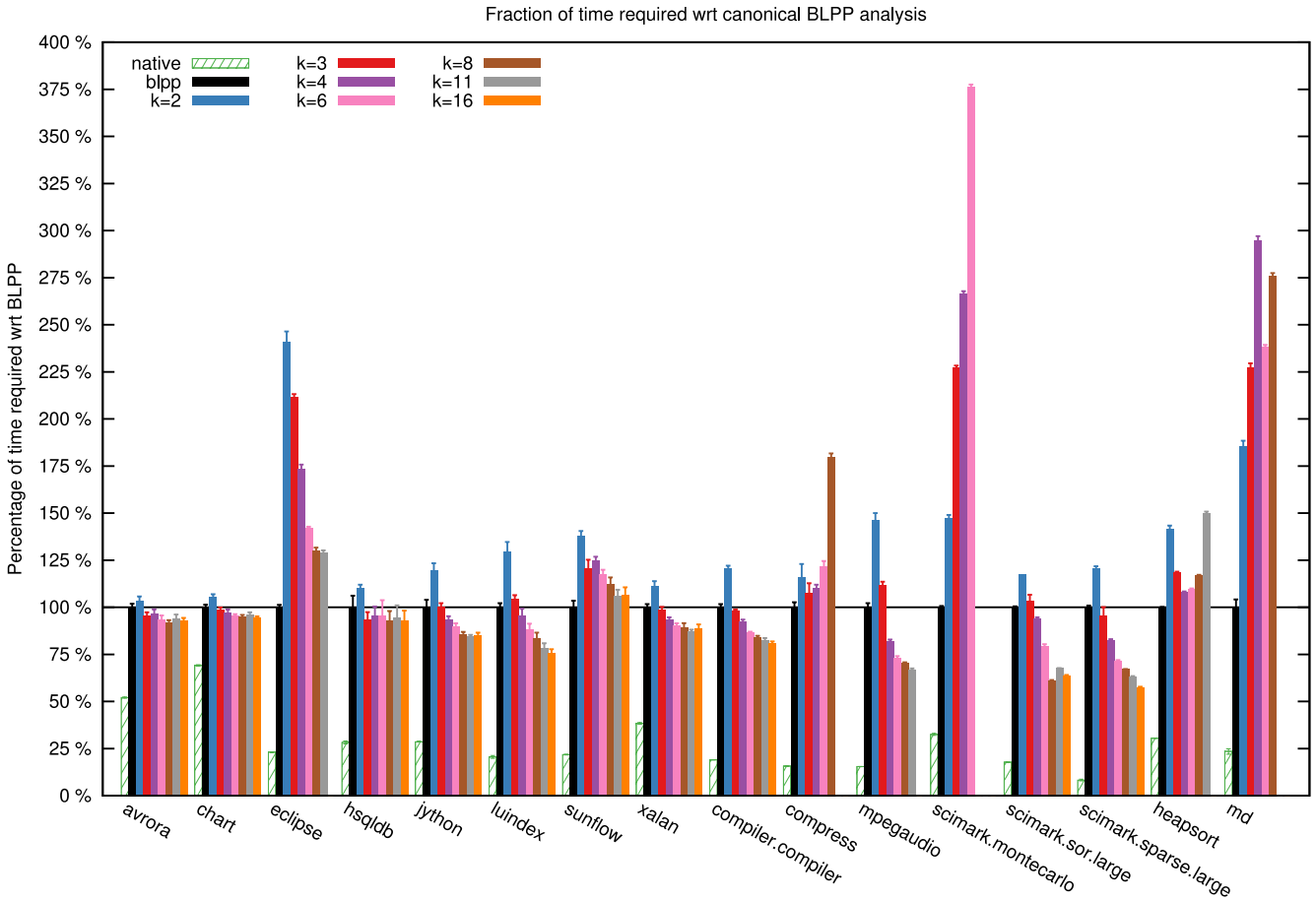


Figure 14: Performance of k-BLPP relative to BLPP.

the performance of our profiler compared to previous approaches and to study properties of path profiles that span multiple iterations for several representative benchmarks.

6.1 Experimental Setup

Bechmarks. We evaluated k-BLPP against a variety of prominent benchmarks drawn from three suites. The `DaCapo` suite [5] consists of a set of open source, real-world applications with non-trivial memory loads. We use the superset of all benchmarks from `DaCapo` releases 2006-10-MR2 and 9.12 that can run successfully with Jikes RVM, using the largest available workload for each benchmark. The `SPEC` suite focuses on the performance of the hardware processor and memory subsystem when executing common general purpose application computations[4]. Finally, we chose two memory-intensive benchmarks from the `Java Grande` 2.0 suite [12] to further evaluate the performance of k-BLPP.

Compared codes. In our experiments, we analyzed the native (uninstrumented) version of each benchmark and its instrumented counterparts, comparing k-BLPP for different values of k (2, 3, 4, 6, 8, 11, 16) with the BLPP profiler developed by Bond [10, 15], which implements the canonical Ball-Larus acyclic-path profiling technique. We upgraded the original tool by Bond to take advantage of native threading support introduced in newer Jikes RVM releases; the code is structured as in Figure 1, except that it does not produce any intermediate stream, but it directly performs `count[r]++`. The software evaluated in this section has been analyzed and endorsed by the OOPSLA 2013 Artifact Evaluation Committee, and the source code is available in the Jikes RVM Research Archive.

Platform. Our experiments were performed on a 2.53GHz Intel Core2 Duo T9400 with 128KB of L1 data cache, 6MB of L2 cache, and 4 GB of main memory DDR3 1066, running Ubuntu 12.10, Linux Kernel 3.5.0, 32 bit. We ran all of the benchmarks on Jikes RVM 3.1.3 (default `production` build) using a single core and a maximum heap size equal to half of the amount of physical memory.

Metrics. We considered a variety of metrics, including wall-clock time, number of operations per second performed by the profiled program, number of hash table operations, data structure size (e.g., number of hash table items for

[4] Unfortunately, only a few benchmarks from SPEC JVM2008 can run successfully with Jikes RVM due to limitations of the GNU classpath.

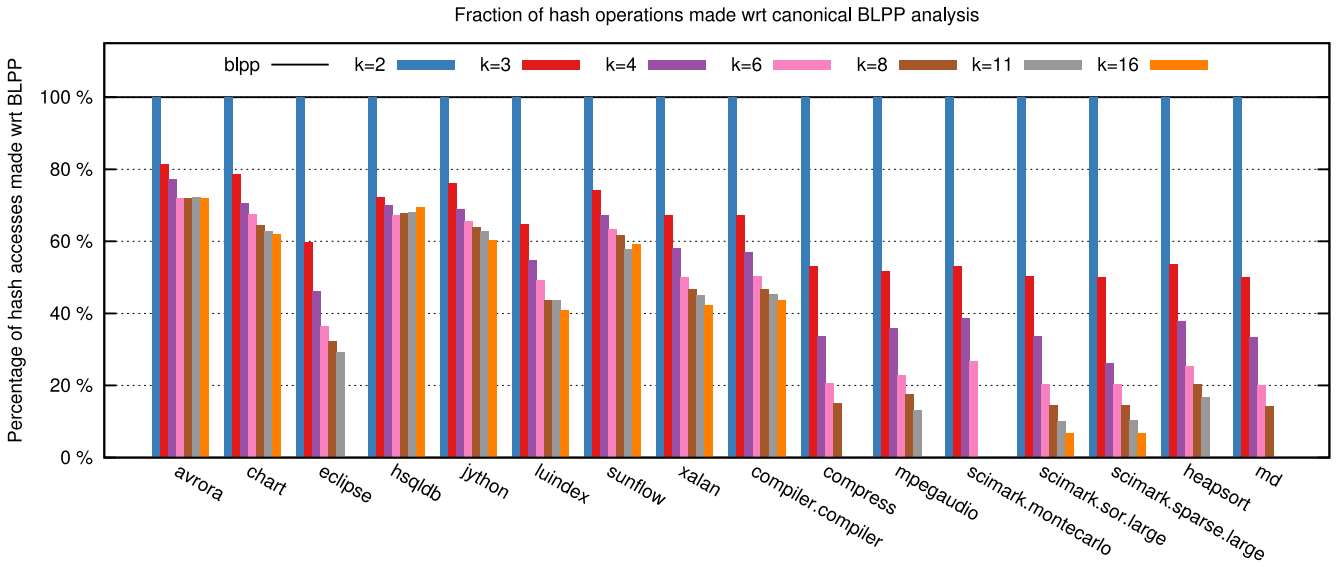


Figure 15: Number of hash table operations performed by k-BLPP relative to BLPP.

BLPP and number of k-SF nodes for k-BLPP), and statistics such as average node degree of the k-SF and the k-IPF and average depth of k-IPF leaves. To interpret our results, we also "profiled our profiler" by collecting hardware performance counters with `perf` [18], including L1 and L2 cache miss rate, branch mispredictions, and cycles per instruction (CPI).

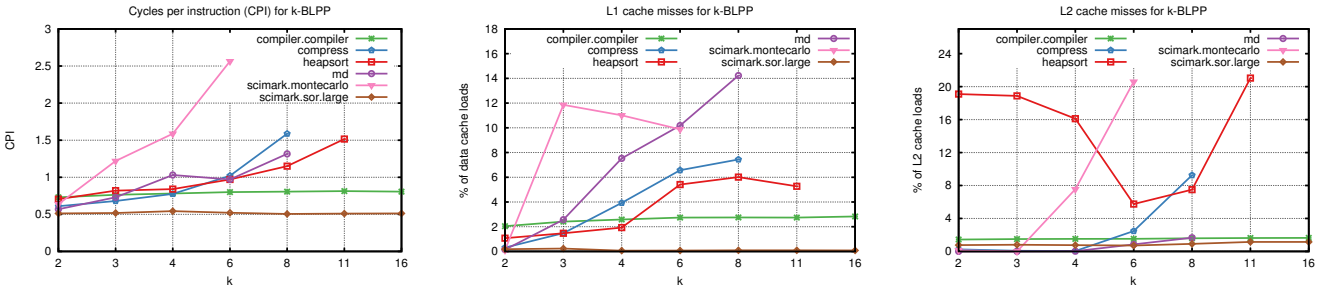
Methodology. For each benchmark/profiler combination, we performed 10 trials, each preceded by a warmup execution, and computed the arithmetic mean. We monitored variance, reporting confidence intervals for performance metrics stated at a 95% confidence level. Performance measurements were collected on a machine with negligible background activity.

6.2 Experimental Results

Performance overhead. In Figure 14 we report for each benchmark the profiling overhead of k-BLPP relative to BLPP. The chart shows that for 12 out of 16 benchmarks the overhead decreases for increasing values of k, providing up to almost 45% improvements over BLPP. This is explained by the fact that hash table accesses are performed by `process_bl_path_id` every $k-1$ items read from the input stream between two consecutive routine entry events (lines 8 and 10 in Figure 11). As a consequence, the number of hash table operations for each routine call is $O(1+N/(k-1))$, where N is the total length of the path taken during the invocation. In Figure 15 we report the measured number of hash table accesses for our experiments, which decreases as predicted on all benchmarks with intense loop iteration activity. Notice that, not only does k-BLPP perform fewer hash table operations, but since only a subset of BL path IDs are inserted, the table is also smaller yielding further performance improvements. For codes such as `avrora` and `hsqldb`, which perform on average a small number of iterations, increasing k beyond this number does not yield any benefit.

On `eclipse`, k-BLPP gets faster as k increases, but differently from all other benchmarks in this class, it remains slower than BLPP by at least 25%. The reason is that, due to structural properties of the benchmark, the average number of node scans at lines 13 and 21 of `process_bl_path_id` is rather high (58.8 for $k=2$ down to 10.3 for $k=16$). In contrast, the average degree of internal nodes of the k-SF is small (2.6 for $k=2$ decreasing to 1.3 for $k=16$), hence there is intense activity on nodes with a high number of siblings. No other benchmark exhibited this extreme behavior. We expect that a more efficient implementation of `process_bl_path_id`, e.g., by adaptively moving hot children to the front of the list, could reduce the scanning overhead for this kind of worst-case benchmarks as well.

Benchmarks `compress`, `scimark.monte_carlo`, `heapsort`, and `md` made an exception to the general trend we observed, with performance overhead increasing, rather than decreasing, with k. To justify this behavior, we collected and analyzed several hardware performance counters and noticed that on these benchmarks our k-BLPP implementation suffers from increased CPI for higher values of k. Figure 16 (a) shows this phenomenon, comparing the four outliers with other benchmarks in our suite. By analyzing L1 and L2 cache miss rates, reported in Figure 16 (b) and Figure 16 (c), we noticed that performance degrades due to poor memory access locality. We believe this to be an issue of our current implementation of k-BLPP, in which we did not make any effort aimed at improving cache efficiency in accessing the k-SF, rather than a limitation of the general approach we propose. Indeed, as nodes may be unpredictably scattered in memory due to the linked structure of the for-



(a) (b) (c)

Figure 16: Hardware performance counters for k-BLPP: (a) cycles per instruction, (b) L1 cache miss rate, (c) L2 cache miss rate.

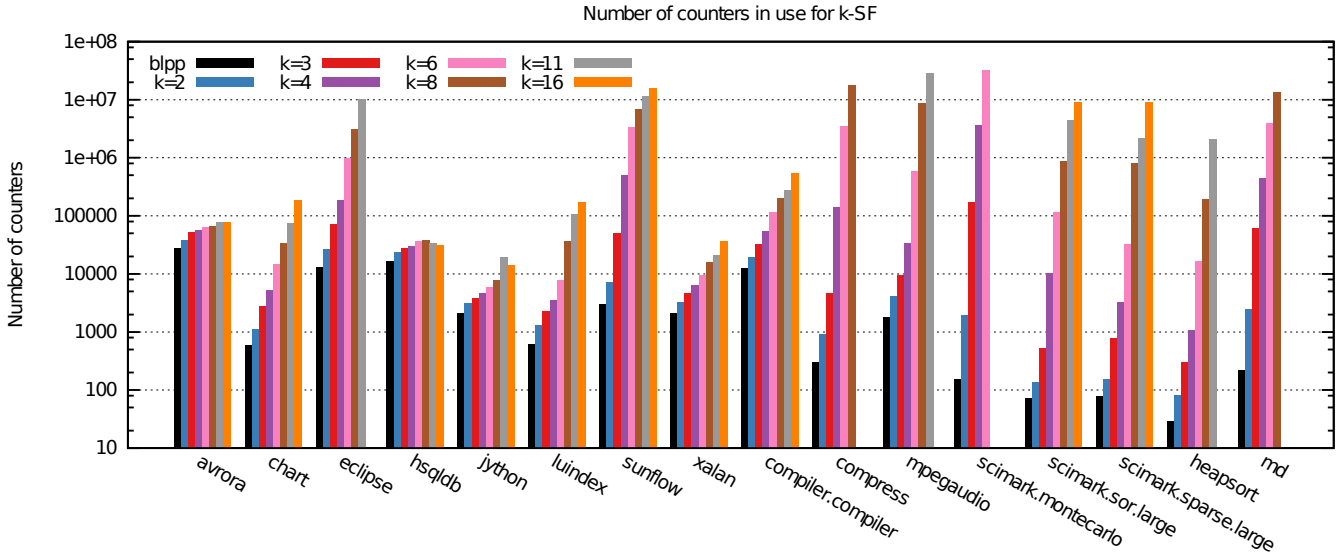


Figure 17: Space requirements: number of hash table entries in BLPP and number of nodes in the k-SF.

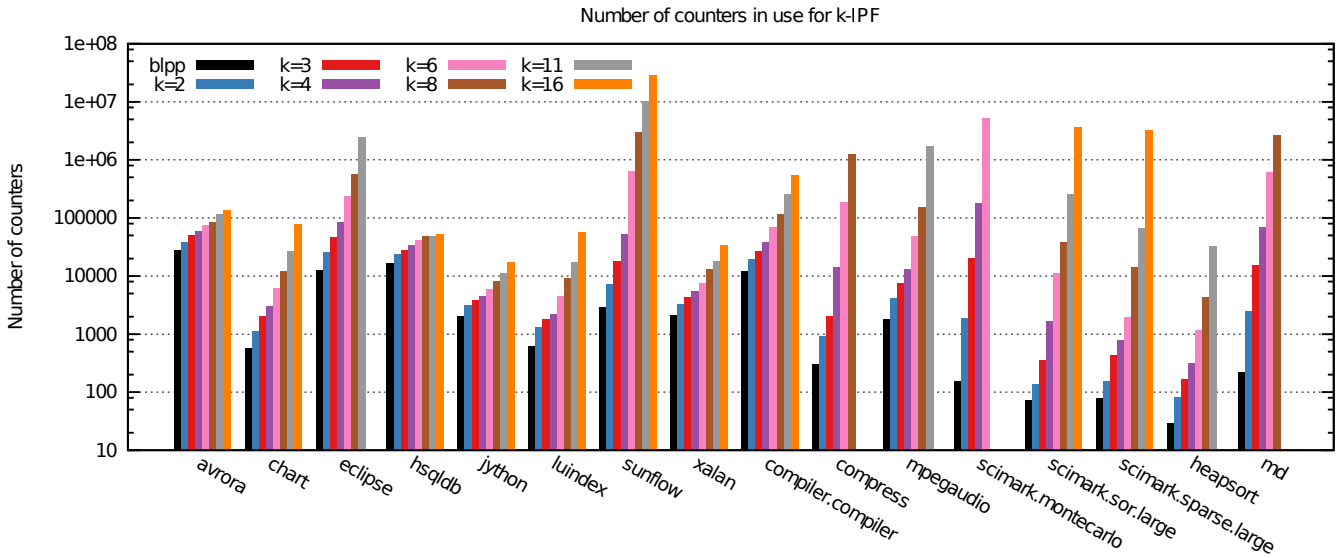


Figure 18: Number of paths profiled by BLPP and k-BLPP.

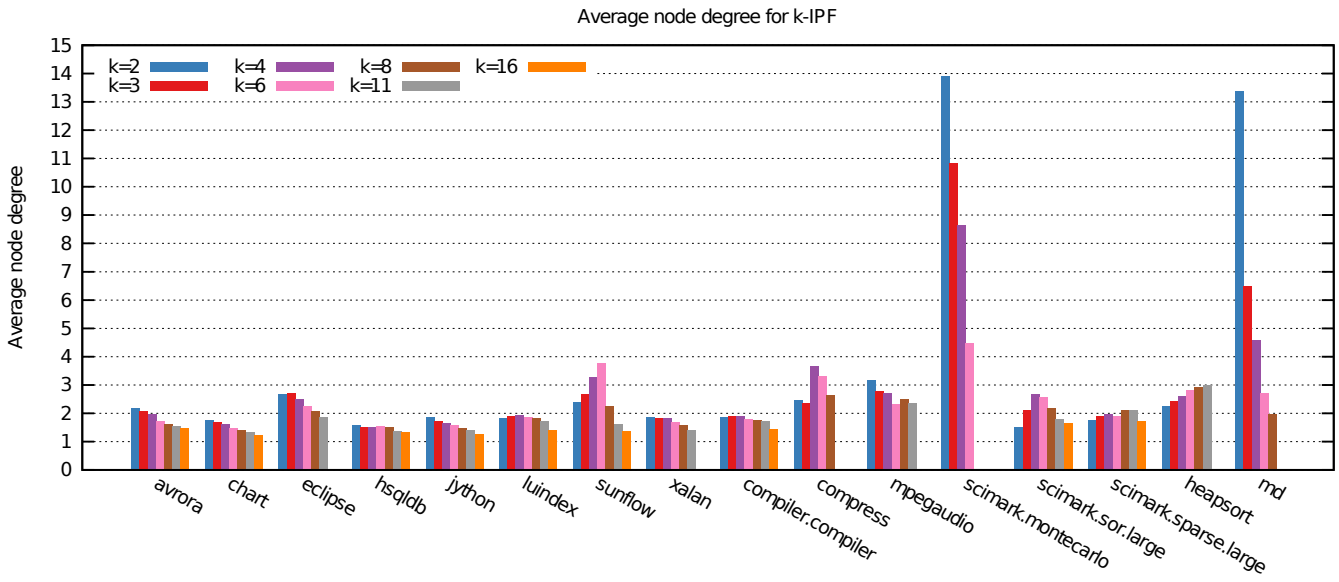


Figure 19: Average degree of k-IPF internal nodes.

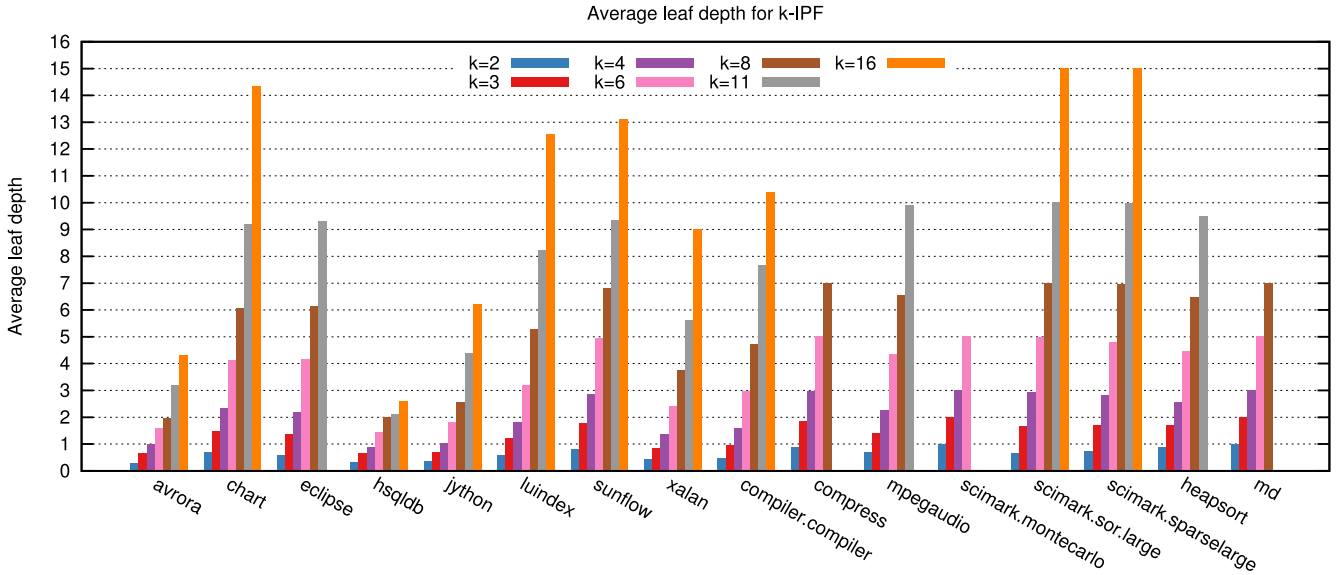


Figure 20: Average depth of k-IPF leaves.

est, pathological situations may arise where node scanning incurs several cache misses.

Notice that, since we never delete either entries from the hash table or nodes from the k-SF, our implementation does not place any additional burden on the garbage collector. The profiler causes memory release operations only when a thread terminates, dumping all of its data structures at once.

Space usage. Figure 17 compares the space requirements of BLPP and k-BLPP for different values of k. The chart reports the total number of items stored in the hash table by BLPP and the number of nodes in the k-SF. Since both BLPP and k-BLPP exhaustively encode exact counters for all distinct taken paths of bounded length, space depends on intrinsic structural properties of the benchmark. Programs with intense loop iteration activity are characterized by substantially higher space requirements by k-BLPP, which collects profiles containing up to several millions of paths. Notice that on some benchmarks we ran out of memory for large values of k, hence some bars in the charts we report in this section are missing. In Figure 18 we report the number of nodes in the k-IPF, which corresponds to the number of paths profiled by k-BLPP. Notice that, since a path may be represented more than once in the k-SF, the k-IPF represents a more compact version of the k-SF.

Techniques	Profiled paths	Average cost w.r.t. BLPP	Variables	Full accuracy	Inter-procedural
BLPP [4]	BL paths	-	1	$\surd$	
SPP [2], TPP [16], PPP [11]	subset of BL paths	smaller	1 or more	$\surd$	
Tallam *et al.* [20]	overlapping paths	larger	many		$\surd$
Roy *et al.* [19]	k-iteration paths	larger	many	$\surd$	
Li *et al.* [17]	finite-length paths	larger	1 or more	$\surd$	
Young [23]	general paths	larger	-	$\surd$	
This paper	k-iteration paths	smaller	1	$\surd$	

Table 1: Comparison of different path profiling techniques.

Structural properties of collected profiles. As a final experiment, we measured structural properties of the k-IPF such as average degree of internal nodes (Figure 19) and the average leaf depth (Figure 20). Our tests reveal that the average node degree generally decreases with k, showing that similar patterns tend to appear frequently across different iterations. Some benchmarks, however, such as `sunflow` and `heapsort` exhibit a larger variety of path ramifications, witnessed by increasing node degrees at deeper levels of the k-IPF. The average leaf depth allows us to characterize the loop iteration activity of different benchmarks. Notice that for some benchmarks, such as `avrora` and `hsqldb`, most cycles consist of a small number of iterations: hence, by increasing k beyond this number, k-BLPP does not collect any additional useful information.

Discussion. From our experiments, we could draw two main conclusions:

1. Using tree-based data structures to represent intraprocedural control flow makes it possible to substantially reduce the performance overhead of path profiling by decreasing the number of hash operations - and also by operating on smaller tables. This approach yields the first profiler that can handle loops that extend across multiple loop iterations faster than the general Ball-Larus technique based on hash tables for maintaining path frequency counters, while collecting at the same time significantly more informative profiles. We observed that, due to limitations of our current implementation of k-BLPP such as lack of cache friendliness for some worst-case scenarios, on a few outliers our profiler was slower than Ball-Larus, with a peak of 3.76x slowdown on one benchmark.
2. Since the number of profiled paths in the control flow graph typically grows exponentially for increasing values of k, space usage can become prohibitive if paths spanning many loop iterations have to be exhaustively profiled. We noticed, however, that most long paths have small frequency counters, and are therefore uninteresting for identifying optimization opportunities. Hence, a useful addition to our method, which we do not address in this work, would be to prune cold nodes on-the-fly from the k-SF, keeping information for hot paths only.

7. Related Work

The seminal work of Ball and Larus [4] has spawned much research interest in the last 15 years, in particular on profiling acyclic paths with a lower overhead by using sampling techniques [10, 11] or choosing a subset of *interesting* paths [2, 16, 21]. On the other hand, only a few works have dealt with cyclic-path profiling.

Tallam *et al.* [20] extend the Ball-Larus path numbering algorithm to record slightly longer paths across loop back edges and procedure boundaries. The extended Ball-Larus paths overlap and, in particular, are shorter than two iterations for paths that cross loop boundaries. These overlapping paths enable very precise estimation of frequencies of potentially much longer paths, with an average imprecision in estimated total flow of those paths ranging from -4% to $+8\%$. However, the average cost of collecting frequencies of overlapping paths is 4.2 times that of canonical BLPP on average.

Roy and Srikant [19] generalize the Ball-Larus algorithm for profiling k-iteration paths, showing that it is possible to number these paths efficiently using an inference phase to record executed backedges in order to differentiate cyclic paths. One problem with this approach is that, since the number of possible k-iteration paths grows exponentially with k, path IDs may overflow in practice even for small values of k. Furthermore, very large hash tables may be required. In particular, their profiling procedure aborts if the number of static paths exceeds $60,000$, while this threshold is reached on several small benchmarks already for $k = 3$ [17]. This technique incurs a larger overhead than BLPP: in particular, the slowdown may grow to several times the BLPP-associated overhead as k increases.

Li *et al.* [17] propose a new path encoding that does not rely on an inference phase to explicitly assign identifiers to all possible paths before the execution, yet ensuring that any finite-length acyclic or cyclic path has a unique ID. Their path numbering algorithm needs multiple variables to record probe values, which are computed by using addition and multiplication operations. Overflowing is handled by using *breakpoints* to store probe values: as a consequence, instead of a unique ID for each path, a unique series of breakpoints is assiged to each path. At the end of program's execution, the *backwalk* algorithm reconstructs the executed paths starting from breakpoints. This technique has been integrated with

BLPP to reduce the execution overhead, resulting in a slowdown of about 2 times on average with respect to BLPP, but also showing significant performance loss (up to a 5.6 times growth) on tight loops (i.e., loops that contain a small number of instructions and iterate many times). However, the experiments reported in [17] were performed on single methods of small Java programs, leaving further experiments on larger industry-strength benchmarks to future work.

Of a different flavor is the technique introduced by Young [23] for profiling *general paths*, i.e., fixed-length sequences of taken branches that might span multiple loop iterations (see Section 3.2). Unfortunately, this technique scales poorly for increasing path lengths l both in terms of space usage and running time. In particular, the running time is proportional not only to the length of the stream of taken branches, but also to the number of possible sequences of length l, that is likely to be exponential in l. In order to reduce the per-taken-branch update time, the algorithm uses also additional space with respect to that required for storing the path counters and identifiers; such space is proportional to the number of possible sequences of length l as well.

A comparison of different path profiling techniques known in the literature with our approach is summarized in Table 1. The "variables" column reports the number of probes required to construct path IDs and does not apply to the work by Young [23], which uses a different approach.

8. Conclusions

In this paper we have presented a novel approach to cyclic-path profiling, which combines the original Ball-Larus path numbering technique with a prefix tree data structure to keep track of concatenations of acyclic paths across multiple loop iterations. A large suite of experiments on a variety of prominent benchmarks shows that, not only does our approach collect significantly more detailed profiles, but it can also be faster than the original Ball-Larus technique by reducing the number of hash table operations.

An interesting open question is how to use sampling-based approaches such as the one proposed by Bond and McKinley [10] to further reduce the path profiling overhead. We believe that the bursting technique, introduced by Zhuang *et al.* [24] in the different scenario of calling-context profiling could be successfully combined with our approach, allowing an overhead reduction while maintaining reasonable accuracy in mining hot paths. In particular, a bursting profiler allows the analyzed application to run unhindered between two sampling points, then it collects a burst of profiling events for an interval defined as *burst length*. Overhead is further reduced through adaptive mechanisms that inhibit redundant profiling due to repetitive execution sequences.

Another way to reduce the profiling overhead may be to exploit parallelism. We note that our approach, which decouples path tracing from profiling using an intermediate data stream, is amenable to multi-core implementations by letting the profiled code and the analysis algorithm run on separate cores using shared buffers. A promising line of research is to explore how to partition the data structures so that portions of the stream buffer can be processed in parallel.

Finally, we observe that, since our approach is exhaustive and traces taken paths regardless of their hotness, it would be interesting to explore techniques for reducing space usage, by pruning cold branches of the k-SF on the fly to keep the memory footprint smaller, thus allowing us to deal with even longer paths.

Acknowledgments. A warm acknowledgment goes to Irene Finocchi for her contributions to the design of the approach presented in this paper and for her help in setting up our experimental package for the OOPSLA 2013 artifact evaluation process. We are indebted to Michael Bond for several interesting discussions, for his invaluable support with his BLPP profiler, and for shedding light on some tricky aspects related to the Jikes RVM internals. We would also like to thank Erik Brangs for his help with the Jikes RVM and José Simão for his hints in selecting an adequate set of benchmarks for the Jikes RVM. Finally, we are grateful to the anonymous OOPSLA 2013 referees for their extremely thorough reviews and for their many useful comments.

References

[1] B. Alpern, C. R. Attanasio, J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. E. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. ISSN 0018-8670.

[2] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proc. ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 35–42. ACM, 2002.

[3] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. k-calling context profiling. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2012, pages 867–878, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

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

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[6] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 146–158, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6.

[7] R. Bodík, R. Gupta, and M. L. Soffa. Load-reuse analysis: design and evaluation. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 64–76, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5.

[8] R. Bodík, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

[9] R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. *SIGPLAN Not.*, 39(4):596–611, Apr. 2004. ISSN 0362-1340.

[10] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *Proc. 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140. IEEE Computer Society, 2005.

[11] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *CGO*, pages 205–216. IEEE Computer Society, 2005.

[12] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 81–88. ACM, 1999.

[13] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, July 1981. ISSN 0018-9340.

[14] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9): 490–499, 1960.

[15] Jikes RVM Research Archive. PEP: continuous path and edge profiling. `http://jikesrvm.org/Research+Archive`.

[16] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *CGO*, pages 239–250, 2004.

[17] B. Li, L. Wang, H. Leung, and F. Liu. Profiling all paths: A new profiling technique for both cyclic and acyclic paths. *Journal of Systems and Software*, 85(7):1558 – 1576, 2012. ISSN 0164-1212.

[18] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`.

[19] S. Roy and Y. N. Srikant. Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. In *CGO*, pages 70–80, 2009.

[20] S. Tallam, X. Zhang, and R. Gupta. Extending path profiling across loop backedges and procedure boundaries. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 251–264, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[21] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, pages 351–362. ACM, 2007.

[22] C. Young and M. D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 115–123, 1998.

[23] R. C. Young. *Path-based compilation*. PhD thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, 1998.

[24] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, pages 263–271, 2006.