Journal of Instruction-Level Parallelism 6 (2004) 1-28 Submitted 10/03; published 4/04

Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers

Toshiaki Yasue YASUE@JP.IBM.COM
Toshio Suganuma SUGANUMA@JP.IBM.COM
Hideaki Komatsu KOMATSU@JP.IBM.COM
Toshio Nakatani NAKATANI@JP.IBM.COM
IBM Tokyo Research Laboratory,
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan

Abstract

Collecting hot paths is important for restructuring and optimizing the target program effectively. It is, however, challenging for Just-In-Time (JIT) compilers, which must collect path profiles on the fly at runtime. In this paper, we propose an efficient online path profiling technique, called *structural path profiling* (SPP), suitable for JIT compilers. The key idea is to partition the target method into a hierarchy of the nested graphs based on the loop structure, and then to profile each graph independently. With SPP, we can collect accurate path profiles efficiently with low overhead. The experimental results show that our technique can collect path profiles with an accuracy of around 90% compared to the offline complete path profiles, while it incurs only 2-3% overhead on average in the active profiling phase.

1. Introduction

Just-In-Time (JIT) compilers [1, 2, 3] have the privilege of exploiting the online profile information from the currently executing program to achieve higher performance. Indeed, many of today's JIT compilers already use some profiling techniques to find hot edges by counting the number of traversals of the edges, called *edge profiles*, of the control flow graphs of the hot methods, and apply advanced optimizations such as node splitting [1], partial redundancy elimination [4], and code positioning [1]. Although JIT compilers have to pay the penalty of the runtime overhead for compilation and profiling during the execution of the program, they can often generate highly optimized code.

It is more challenging to find hot paths accurately by counting the number of traversals of the selected paths, called *path profiles*. In general, path profiles can provide more details on the behavior of the target method than edge profiles. Edge profiles may be sufficient to predict hot paths in a given method, but they cannot ensure identifying the individual hot paths accurately enough to apply some advanced optimizations such as tail-duplication [5, 6], cost and benefit based transformation [7, 8, 9], and loop peeling and loop unrolling for superblock formations [10]. Indeed, these optimizations are important for extracting parallelism to fully exploit the potential of the wide instruction word machines such as IA-64.

Two offline path profiling techniques [6, 11] have been proposed for static compilers, but neither of them can be applied by itself to JIT compilers without introducing an innovative framework to reduce the high overhead of the path profiling.

The goal of online profiling is to reduce the overhead of profiling while collecting sufficiently accurate profiles to optimize the target methods. In this paper, we define the *accuracy* of online profiles as the degree of agreement with respect to the offline complete profiles, as originally suggested by Feller [12].

While there are some simple and straightforward implementations, those approaches cannot satisfy our requirements. For example, even when we apply a path profiling technique only to the small number of hot methods, the overhead of profiling may remain unacceptable if the target method has a hot loop that cycles for a long time. In order to reduce the overhead, we could turn off the path profiling after a certain threshold is reached, but then we cannot collect the path profiles for the remaining part of the method following the hot loop. Therefore, it is hard to control the overhead using a single threshold count to cover the whole method. The *instrumentation sampling* framework of Arnold and Ryder [13] has a similar problem to be described later.

To solve these problems, we propose a new online path profiling framework, called *structural path profiling* (SPP), suitable for JIT compilers. The key idea is to partition a method into a hierarchy of nested graphs, called *structure graphs*, each of which represents a slice of the loop at the given loop nest level, and then to profile each graph independently. For example, a doubly nested loop has two structure graphs: an outer structure graph and an inner structure graph. The outer structure graph includes regular nodes, each of which represents a basic block at the outer loop level, and a loop node, which represents the inner loop. The inner structure graph includes only regular nodes, each of which represents a basic block of the inner loop. In the given method, the topmost structure graph, called the *outline structure graph*, is composed of regular nodes, each of which represents a basic block at the highest level of the method, and loop nodes, each of which represents a nested loop.

We use these structure graphs to collect path profiles, starting from the topmost graph and going deeper into the innermost graphs one by one. First we apply instrumentation for path profiling to the outermost level, that is, to the outline structure graph for the given method. Once the local path profiles are collected at the outermost level, we go deeper into the next level of the structure graphs. We repeat this process until we reach the bottom level of the structure graphs. Finally we construct a set of global path profiles for the target method by adjusting the local path profiles of all of the structure graphs. Since we do path profiling incrementally from the top level of the structure graphs to the lower levels, we can collect path profiles for the whole region of the method with low overhead. Furthermore, this mechanism allows us to control the instrumentation for each loop separately and to balance between the profiling overhead and the accuracy of the resulting profiles.

We implemented SPP in the IBM Java JIT compiler [3] and did experiments with the path profiling technique proposed by Ball and Larus [11] to find the right threshold counts in comparison to the offline complete version. We evaluated two aspects: accuracy and overhead. For accuracy, we used the *overlap percentage metric* [12], by comparing the collected path profiles with the offline complete path profiles. Our new framework performs with an accuracy percentage of around 90% on average using 1,000-profile count, which is one-tenth of the profile count required to achieve the same level of accuracy without using our framework. The profiling overhead is around 2-3% on average (in the active profiling phase) for total profile counts ranging from 100 to 10,000.

The contributions of our paper are:

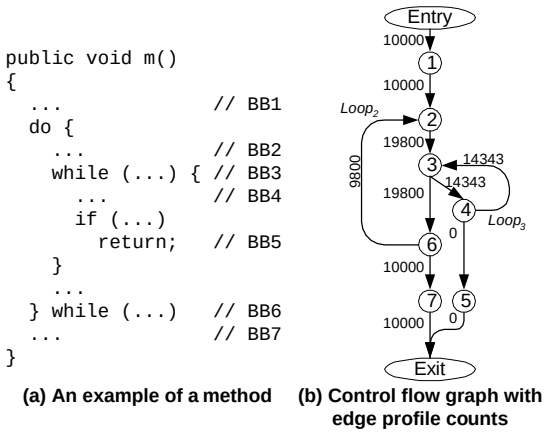


Figure 1: An example of a method with a nested loop and the corresponding control flow graph annotated with edge profile counts.

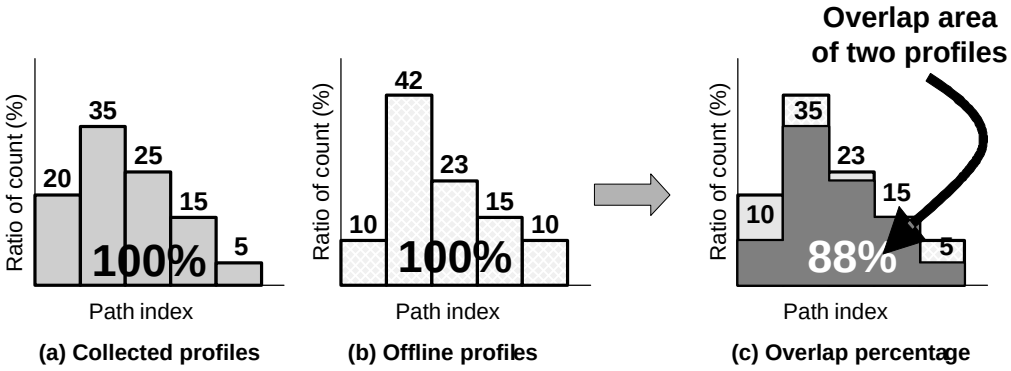


Figure 2: Overlap percentage.

- *A new online path profiling framework, called structural path profiling, suitable for JIT compilers*, and

- *Experimental data to validate the effectiveness of our new framework.*

The rest of this paper is organized as follows. The next section describes the background of our work, showing the advantages of path profiles as compared to edge profiles and the issues in online path profiling. Then we describe SPP and its implementation in detail in Section 3. Section 4 presents the experimental results for evaluating the storage requirements, the accuracy, and the overhead when applying this technique. In Section 5, we discuss some improvements of the current implementation of SPP. Section 6 summarizes the related work, and finally Section 7 presents our conclusions.

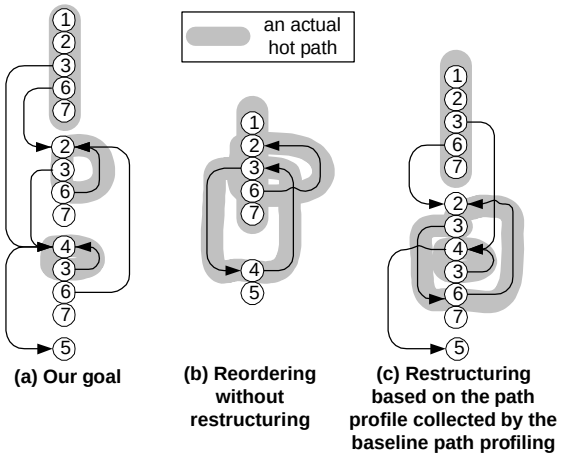


Figure 3: Restructuring by loop peeling

2. Background

In this section, we explain the background of using path profiling for JIT compilers. We first discuss the advantages of path profiles over edge profiles by showing how the accuracy of hot path information can affect a program restructuring optimization. We then show why the naive approach for online path profiling does not work well to balance the profiling overhead and the accuracy of profiles.

Throughout this paper, we use the example shown in Figure 1. In the example, (a) shows a method that contains a doubly nested loop, and (b) shows the corresponding control flow graph annotated with the edge profiles when the method is called 10,000 times. Here we call the outer loop $loop_2$ and the inner loop $loop_3$, respectively. We suppose there are three hot paths in the method: the straight line path *1-2-3-6-7*, the outer loop iteration path *2-3-6-2*, and the inner loop iteration path *3-4-3*. Here, we assume that neither loop is executed for most of the executions of the method, but that each loop is iterated heavily once it has been entered.

We define the accuracy of online profiles as the degree of agreement with respect to the offline complete profiles. To measure this degree, we use the metric of overlap percentage [12], which evaluates what percentage of a set of profiles overlaps with another set of profiles. Figure 2 shows an example. In this figure, a set of collected profiles and a set of offline complete profiles are shown in (a) and (b) respectively, each of which shows what percentage each path occupies in the total profiles. Then we calculate the overlap area of the two profiles as shown in (c), and get 88% of overlap as the accuracy of the collected profiles.

2.1 Edge profiles vs. path profiles

Edge profiles can predict hot paths successfully when most of the conditional branches are biased in one direction and thus predictable, as is typical of numerical programs [14, 15, 16, 17]. They cannot, however, accurately distinguish the paths that include unbiased conditional branches, as is typical of non-numerical programs. Although edge profiles may be sufficient for roughly detecting the hot paths even in the non-numerical programs [14], we

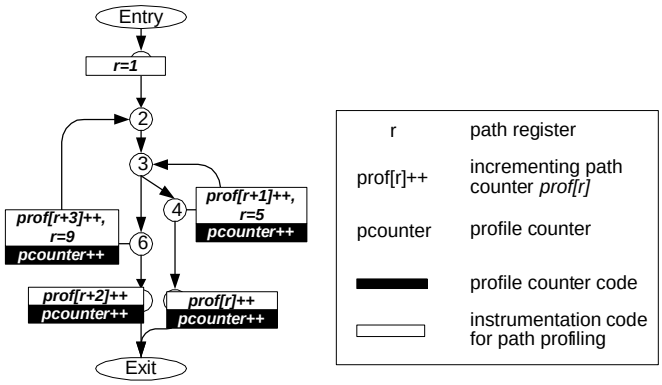


Figure 4: Instrumentation code and profile counter inserted by BPP.

need more accurate path profiles for JIT compilers to apply code restructuring and advanced optimizations (such as loop peeling, loop unrolling, tail duplication, and partial redundancy eliminations) to the detected hot paths for achieving the highest possible performance.

In Figure 1, for example, it is hard to predict the outcome of the branches at 3 and 6, both of which are almost even in their branch directions. In this example, both loops are rarely entered, but they are iterated for many cycles once they are entered. In fact, there are three correct hot paths, *1-2-3-6-7*, *2-3-6-2*, and *3-4-3*. Figure 3(a) shows our goal for restructuring when we apply loop peeling based on these three hot paths detected. Edge profiles fail to predict these hot paths, and would provide an estimated hot path *1-2-3-6-7*. This will lead to the transformation with code reordering but not with loop peeling, as shown in Figure 3(b). As we describe it in more details in the following section, a naive approach for online path profiling (we will call it as *BPP*) can predict only two paths, *1-2-3-6-7* and *3-4-3*, and this will lead to the transformation shown in Figure 3(c). Our goal is to detect those hot paths more accurately which should lead to the transformation shown in Figure 3(a).

Path profiles are a superset of edge profiles and we can easily generate edge profiles from path profiles. Namely, by collecting path profiles, we can apply both optimizations: one based on path profiles and the other based on edge profiles. We can deal with both profiles uniformly in our SPP framework, though we primarily show the case of path profiling for the rest of this paper.

2.2 Issues in online path profiling

The critical issue in online path profiling is how to balance between the profiling overhead and the accuracy of the resulting profiles. Unlike offline path profiling [6, 11], the time for collecting profiles is limited, even if we apply it only to a few hot methods.

In order to collect accurate path profiles with conventional path profiling techniques, we must incur profiling overhead in proportion to the number of executed paths for every invocation of the method. This means that the growth of the profiling overhead can be explosive if the method includes some hot loops. If we limit the total counts of the collected profiles with a threshold in order to avoid this explosion by using the *ephemeral instrumen-*

Table 1: Path profiles collected by BPP.

Path index	*Paths*	*Online profiles*	*Offline profiles*
1	Entry-1-2-3-4-5-Exit	0	0
2	Entry-1-2-3-4-(3)	5	345
3	Entry-1-2-3-6-7-Exit	124	9,462
4	Entry-1-2-3-6-(2)	0	193
5	(4)-3-4-5-Exit	0	0
6	(4)-3-4-(3)	167	13,660
7	(4)-3-6-7-Exit	4	345
8	(4)-3-6-(2)	0	338
9	(6)-2-3-4-5-Exit	0	0
10	(6)-2-3-4-(3)	0	338
11	(6)-2-3-6-7-Exit	0	193
12	(6)-2-3-6-(2)	0	9,269
Total		300	34,143

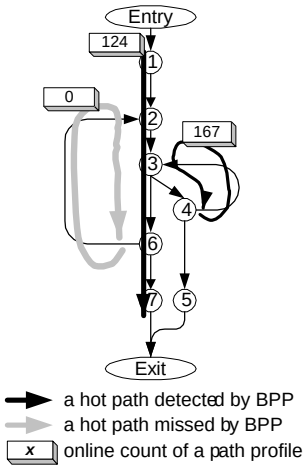


Figure 5: Hot paths detected by BPP.

tation technique [18], we cannot guarantee that the collected path profiles cover the whole region of the method. For example, if the target hot method has several disjoint loops, one of which iterates for a long time, then it is difficult to collect path profiles for every loop evenly. In some cases, path profiles for some loops cannot even be collected if profiling is terminated too early. That is, the counter reaches the threshold before collecting path profiles for other loops while the first hot loop is still iterating.

Here is an example we derived by applying the Ball-Larus technique [11] to online use, though it was originally designed for an offline use. In order to control the total number of profiles, we use a profile counter to stop collecting profiles for each target method after a given constant number of paths are collected. We call this version of the path profiling technique *Baseline Path Profiling* (BPP).

Figure 4, Table 1, and Figure 5 show the result of BPP when we applied it to the same method as shown in Figure 1 while collecting 300 profiles. Figure 4 shows the instrumentation code generated by BPP including some code for a profile counter for controlling the total number of samples collected for each path. Table 1 shows both the online version and the offline version of the path profiles collected. The offline version detected three hot paths, $path_3$, $path_6$, and $path_{12}$, while the online version detected only two hot paths, $path_3$ and $path_6$. In other words, the online version missed the hot path, $path_{12}$, which represents the iteration of $loop_2$, because BPP stopped the path profiling before executing the backedge of $loop_2$. The thick arrows in Figure 5 show the actual hot paths labeled with their profile counts as collected by BPP.

Figure 3(c) shows the result of the loop peeling by using the above BPP profiles. Although this result is better than the one by using the edge profiles (Figure 3(b)), it still fails to optimize the hot path, *2-3-6-2*, because BPP cannot detect it.

A possible alternative approach for online path profiling is to employ the *instrumentation sampling* framework proposed by Arnold and Ryder [13] and apply BPP within that framework. The framework performs code duplication and uses a compiler-inserted, counter-based sampling technique to switch control between the instrumented code and the non-instrumented code. The sampling rate controls both the profiling overhead and the quality of the profiles. This may significantly reduce the overall overhead of the runtime profiling when compared with the offline profiling, but it will be difficult to determine an appropriate sampling rate for each target method. If we set a low sampling rate, it will take a long time to collect the number of profiles required and this can lead to delays of the recompilations for optimizations using the profile results. If we use a high sampling rate, it will collect the number of profiles required earlier, but it is possible to miss profiles for some hot loops.

3. Structural path profiling

The major problem in BPP as described in the previous section is that we cannot control the number of profiles independently for each of the loops contained in the target method, and thus have to execute the loops all the way to completion to get the profiles for the whole region of the method.

In order to address this problem, we propose a new online profiling framework, called structural path profiling (SPP), suitable for JIT compilers. The key idea of SPP is to

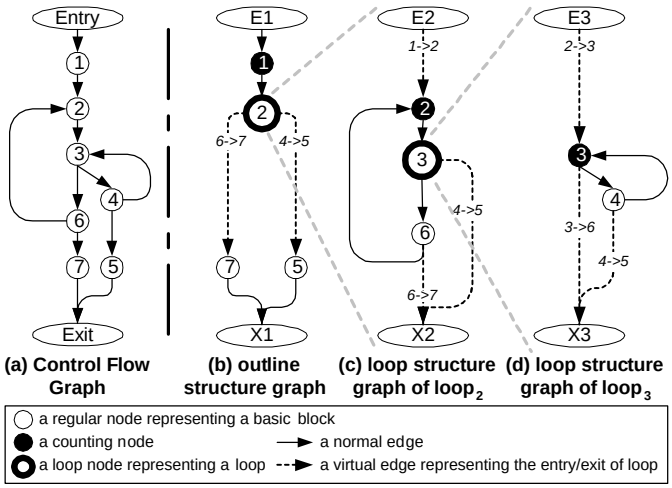


Figure 6: Structure graphs for the example.

partition the target method into a hierarchy of nested graphs, called structure graphs, based on the loop structure of the method, and then apply path profiling to each structure graph independently. In constructing structure graphs, each of the inner loops is reduced into a single node, so we can perform path profiling for each structure graph without worrying the overhead of profiling inner loops.

This approach can solve the problem of online path profiling for the following two reasons. First, because we activate path profiling only for one level of the hierarchy of the nested loops, we can limit the profiling overhead. Second, because we collect a number of samplings sufficiently large for each structure graph, we can recover highly accurate path profiles of the whole method from the local path profiles.

In this section, we first introduce some definitions of structure graphs and then describe each phase of SPP in detail: construction of structure graphs in Section 3.2, generation of the instrumentation code in Section 3.3, the profiling controller in Section 3.4, and generation of the global profiles in Section 3.5. Finally we describe an example of optimizations using the results in Section 3.6.

3.1 Structure graphs

A structure graph is defined as a graph that consists of nodes representing basic blocks or loops, and edges representing the control flow between the nodes. For example, Figure 6 shows three structure graphs constructed from the original control flow graph for the method in Figure 1. Here, a bold circle shows a *loop node*, and a dashed line shows a *virtual edge* that represents an edge entering into the loop or an edge exiting from the loop. The label of each virtual edge shows the corresponding edge in the original control flow graph.

There are two kinds of structure graphs: an outline structure graph and a loop structure graph. An *outline structure graph* is defined as a graph in which every strongly connected region is replaced with a loop node in the control flow graph. The *counting node* of the outline structure graph is defined as the node representing the method entry. If the entry

```
procedure makeStrGraph(Method m)
1. generate control flow graph og for m
2. // loop region detection by Havlak's algorithm
3. analyzeLoop(og, Entry)
4. make a hierarchy of structure graphs of m from loop nesting tree made by
     calling the procedure analyzeLoop
5. // structure graph construction
6. foreach sg ∈ structure graphs of m do
7.    make loop nodes for all inner loops in sg
8.    foreach ln ∈ loop nodes of sg do
9.       lg := corresponding loop structure graphs to ln
10.      change destination of all incoming edges to ln and make virtual entry
           edges of lg corresponding to them
11.      change source of all outgoing edges to ln and make virtual exit edges
           of lg corresponding to them
12.   enddo
13. enddo
```

Figure 7: Algorithm to construct structure graphs.

node is located in a loop, the loop node corresponding to the loop is used for the counting node. Figure 6(b) shows the outline structure graph for the method.

A *loop structure graph* is defined as a graph for each loop header node, detected as the target node of a backedge, and it forms a strongly connected region including the loop header node. If the strongly connected region contains some other loop header nodes that form inner strongly connected regions, they are replaced with loop nodes. The counting node of the loop structure graph is defined as the loop header node. The hierarchical relationship between each nested loop is determined based on the inclusion relationships between them. In Figure 6, (c) and (d) show the loop structure graphs corresponding to $loop_2$ and $loop_3$, respectively.

Because SPP slices the method into each level of loop nests, it cannot find any path crossing to a different loop nesting level. In practice, however, this is not a serious problem, since loop optimizations are usually applied for each loop level separately.

3.2 Construction of structure graphs

Figure 7 shows the algorithm for constructing structure graphs for the given method. After generating the control flow graph, we detect loop regions by calling the function analyzeLoop according to Havlak's algorithm [19], which is an extension of Tarjan's loop detection algorithm to handle irreducible loops in almost linear time. After detecting loop regions, we construct structure graphs by reducing each inner loop into a loop node. When we encounter an infinite loop with no exit edge, we add a dummy edge from the loop node to the exit node of the structure graph. Each virtual edge maintains the original edge information to generate instrumentation code in the next phase.

```
procedure genInstCode(Method m)
1. // generate instrumentation code for each structure graph
2. // and counters used for adjusting profiles
3. foreach sg ∈ structure graphs of m do
4.    // generating instrumentation code of path profiling
5.    // by applying Ball-Larus path profiling to sg
6.    applyBallLarusPathProfiling(sg)
7.    generate a profile counter to the counting node of sg
8.    foreach e ∈ edges of sg do
9.       if (instrumentation code is assigned to e) then
10.         find insertion point p for the code of e
11.         register instrumentation code of e to p
12.      endif
13.   enddo
14. enddo
15. // insert instrumentation code into m
16. foreach n ∈ all regular nodes of m do
17.   if (hasInstInfo(n)) then
18.      insert instrumentation code to n
19.   endif
20. enddo
21. foreach e ∈ all regular edges of m do
22.   if (hasInstInfo(e)) then
23.      insert a basic block b into e
24.      insert instrumentation code into b
25.   endif
26. enddo
```

Figure 8: Algorithm to generate instrumentation code.

When two different loop nestings share the same header node, we construct only one loop structure graph for them. Although SPP has no problem in dealing with these kinds of loops, it is more desirable to separate such loop nestings into different loop structure graphs by creating a separate header node for each loop.

3.3 Generation of instrumentation code

Figure 8 shows the algorithm to generate instrumentation code. First, we generate instrumentation code for path profiling on each structure graph by using a conventional path profiling technique. While we currently use the Ball-Larus technique because of its low overhead, we could use the Young-Smith technique [6] instead.

We then generate the code for a profile counter at the counting node of each structure graph to control the total counts of path profiles and to monitor the completion of profiling at runtime. If the structure graph includes an irreducible loop, we generate the same code at every entry edge to the graph except for the edges linked to the counting node of the graph.

Finally each piece of instrumentation code for each structure graph is inserted at the corresponding point in the method, and the structure graph information is stored. The native code for each block of instrumentation code is generated by using a dynamic instrumentation technique [18, 20] to make it possible to turn each instance of instrumentation code on and off at runtime.

A path profile is collected with a sequence of instrumentation code by initializing and updating the path register r and incrementing its corresponding path counter $prof[r]$ as shown in Figure 11. We note here that there is a chance for profiling to start from the middle of a sequence without initializing the path register. We currently ignore such cases except that we insert the code for the range check for every update of the path counter.

3.4 Profiling Controller

The *profiling controller* manages the progress of profiling for a set of structure graphs. As shown in Figure 9, it begins with path profiling of the given hot method by invoking the function startProfiling for the outline structure graph. During the profiling, when the profile counter of any structure graph reaches the specified threshold value, the function profileController is called.

The function profileController checks whether the path profiling for the structure graph has been finished. If not, it disables profiling and then starts profiling for each of the inner loop structure graphs. If the current structure graph is a leaf graph, the function setCompletion is called to check if profiling this method has been completed.

The controller also periodically monitors the progress of profiling for each structure graph using the function checkCompletion. If the execution of path profiling of a structure graph takes too long, it concludes that the region for the structure graph is rarely executed, and stops profiling to proceed to the next level of the structure graph. The whole profiling process terminates when the controller sets the isCompleted flag for the outline structure graphs.

Figure 10 illustrates the behavior of the profiling controller for the method of Figure 1. The labeled numbers indicate the order of the events. As shown in this figure, the profiling successively proceeds from the topmost structure graph to the inner structure graphs one by one. Figure 11 shows the examples of the instrumentation code when enabled for each level of the structure graphs. Each part represents the code corresponding to the status of (2), (6), and (10) in Figure 10, respectively.

3.5 Generation of global profiles

After performing path profiling, we need to adjust the local profile counts for each structure graph to obtain the global path profiles of the target method. This adjustment corrects the counts of local path profiles for each loop structure graph by multiplying by a coefficient. This coefficient A_X can be calculated with the following formula, where X is the structure graph, Y is the outer structure graph of X, N_X is the loop node representing X in Y, $ENTRY_X$ is the entry node of X, C_P is the profile count of path P, and $P_X(N)$ is a set of paths including the node N in X.

```
// All the elements of isStarted[*], isFinished[*], and isCompleted[*] are initialized
// with FALSE.

procedure STARTPROFILING(Graph sg)
1. isStarted[sg] := TRUE
2. enable all instrumentation code blocks of sg

procedure PROFILECONTROLLER(Graph sg)
3. if (not isFinished[sg]) then
4.    disable all instrumentation code blocks of sg
5.    isFinished[sg] := TRUE
6.    if (sg has inner loop structure graphs) then
7.       foreach lg ∈ inner loop structure graphs of sg do
8.          if (not isStarted(lg)) STARTPROFILING(lg)
9.       enddo
10.   else
11.      SETCOMPLETION(sg)
12.   endif
13. endif

procedure SETCOMPLETION(Graph sg)
14. isCompleted[sg] := TRUE
15. if (sg is a loop structure graph) then
16.   og := outer structure graph of sg
17.   foreach lg ∈ inner loop structure graphs of sg do
18.      if (not isCompleted[lg]) return
19.   enddo
20.   SETCOMPLETION(og)
21. endif

procedure CHECKCOMPLETION(Graph sg)
22. if (isCompleted[sg]) return
23.   if (not isFinished[sg] and ELAPSEDTIME(sg) > threshold) then
24.      PROFILECONTROLLER(sg) // time out for sg
25.   else
26.      foreach lg ∈ inner loop structure graphs of sg do
27.         if (not isCompleted[lg]) then
28.            CHECKCOMPLETION(lg)
29.            if (not isCompleted[lg]) return
30.         endif
31.      enddo
32.      isCompleted[sg] := TRUE
33. endif
```

Figure 9: Algorithm for profile controller.

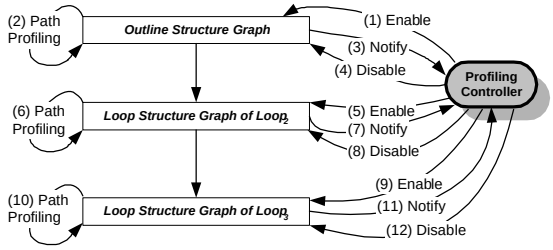


Figure 10: Operation of profiling controller in hierarchical order.

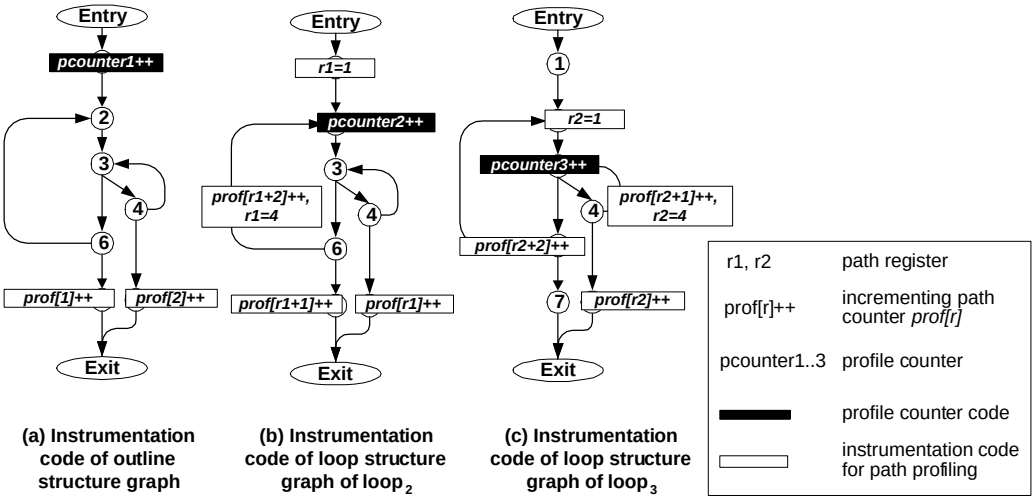


Figure 11: Instrumentation code and profile counter inserted by SPP.

```
procedure ADJUSTSPP(Method m)
 1. if (loop structure graphs exist in m) then
 2.    og := get outline structure graph of m
 3.    A[og] := 1
 4.    ADJUSTPROFILES(og)
 5. endif

 procedure ADJUSTPROFILES(Graph sg)
 6. foreach ln ∈ loop nodes in sg do
 7.    // calculate the frequency of ln
 8.    loopCount := 0
 9.    foreach p ∈ paths passing though ln do
10.       loopCount += profile count of p
11.    enddo
12.    lg := loop structure graph of ln
13.    entryCount := 0
14.    foreach p ∈ paths starting from the entry node of lg do
15.       entryCount += profile count of p
16.    enddo
17.    A[lg] := A[sg] * (loopCount / entryCount)
18.    if (lg has inner loop structure graphs) then
19.       ADJUSTPROFILES(lg)
20.    endif
21. enddo
```

Figure 12: Algorithm to generate global profiles.

Table 2: Path profiles collected by SPP.

Graph	*index*	*Paths*	*Online profiles*		*Offline profiles*	*Path index of BPP*
			Local profiles	*Global profiles*		
outline structure	O-1	E1-1-[2]-7-X1	100	100.0	10,000	3,(2,4,7,11)
graph	O-2	E1-1-[2]-5-X1	0	0.0	0	1,(2,4,5,9)
loop structure	L2-1	E2-2-[3]-X2	0	0.0	0	1,(5)
graph of $Loop_2$	L2-2	E2-2-[3]-6-X2	50	98.0	9,800	3,(2,7)
	L2-3	E2-2-[3]-6-(2)	1	2.0	200	2,(8)
	L2-4	(6)-2-[3]-X2	0	0.0	0	9,(5)
coefficient	L2-5	(6)-2-[3]-6-X2	1	2.0	200	11,(7)
1.96	L2-6	(6)-2-[3]-6-(2)	48	94.1	9,600	12,(8)
loop structure	L3-1	E3-3-4-X3	0	0.0	0	1,9
graph of $Loop_3$	L3-2	E3-3-4-(3)	2	6.8	683	2,10
	L3-3	E3-3-X3	56	189.3	19,117	3,4,11,12
	L3-4	(4)-3-4-X3	0	0.0	0	5
coefficient	L3-5	(4)-3-4-(3)	40	135.2	13,660	6
3.38	L3-6	(4)-3-X3	2	6.8	683	7,8
total			300	634.1	63,943	

$$A_X = \begin{cases} 1 & \text{if } X \text{ is an outline structure graph} \\ \frac{A_Y * \sum_{q \in P_Y(N_X)} C_q}{\sum_{p \in P_X(ENTRY_X)} C_p} & \text{otherwise} \end{cases}$$

This formula computes the coefficient of each graph as the ratio of the execution count for the loop node in the outer graph over the total of the profile count for all of the paths representing the entries to the loop. The profile adjustment algorithm is shown in Figure 12. We perform the adjustment by calling adjustSPP if the target method has one or more loops, starting from the outline structure graph and proceeding to the lower levels in order.

We use the example in Table 2 to show how we can adjust the local profiles to generate the global profiles. Here, we assume that we collected 100 profiles for each structure graph, or a total of 300 profiles for the whole method. Table 2 shows the list of paths, local profiles that indicate the actual counts of the profiles, the global profiles after the adjustments, and the offline complete profiles. The table also indicates the corresponding path indices defined by BPP as listed in Table 1 in the rightmost column.

The local count of each path in the loop structure graph of $loop_2$ is multiplied by (1*100)/51 (=1.96) to generate the global profile. In the dividend, the value 1 is the coefficient of the outline structure graph and the value 100 is the sum of the local profile counts of $path_{O-1}$ and $path_{O-2}$, each of which contains the loop node of $loop_{O-2}$. The divisor 51 is the sum of the local profile counts of $path_{L2-1}$, $path_{L2-2}$, and $path_{L2-3}$, each of which represents the loop entry paths of the loop structure graph of $loop_2$. Similarly, the local count for $loop_3$ is multiplied by (1.96*100)/58 (=3.38) to get the global profile, where

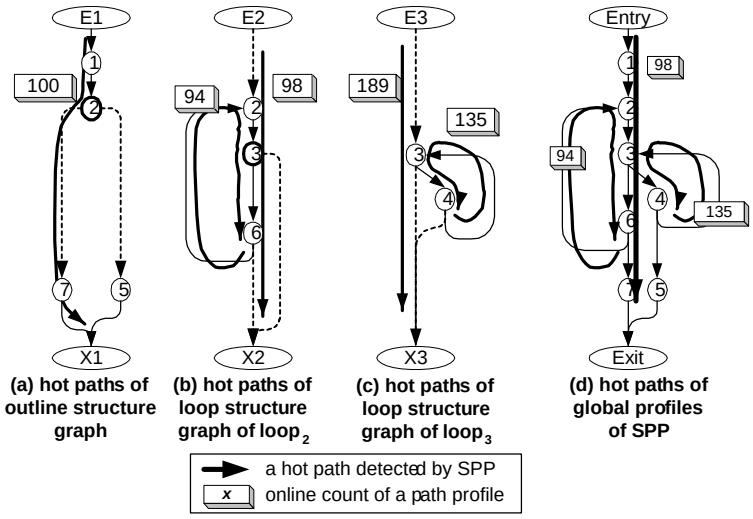


Figure 13: Hot paths detected by SPP

1.96 is the coefficient of the loop structure graph for $loop_2$, 100 is the sum of the local profile counts of the paths from $path_{L2-1}$ to $path_{L2-6}$, and 58 is the sum of local profile counts of $path_{L3-1}$, $path_{L3-2}$, and $path_{L3-3}$.

As we can see from the Table 2, the global profiles after the adjustment closely parallel the offline profiles.

3.6 An example of optimizations

From the global profiles in Table 2, we can extract three hot paths for use in optimizations, as follows. First, we can easily find five hot paths, $path_{O-1}$, $path_{L2-2}$, $path_{L2-6}$, $path_{L3-3}$, and $path_{L3-5}$. Among these paths, $path_{O-1}$, $path_{L2-2}$, and $path_{L2-6}$ contain a loop node. Since $path_{L3-5}$ does not include a loop node and has no relationship with the outer loop structure graph, the path *3-4-3*, $path_{L3-5}$ itself, can be extracted as a hot path. We also see that the two loops are not iterated in most cases, from the two hot paths, $path_{L2-2}$ and $path_{L3-3}$. Therefore, we can conclude that the path *1-2-3-6-7* is a hot path as the combination of $path_{O-1}$, $path_{L2-2}$, and $path_{L3-3}$, and similarly, the path *2-3-6-2* is a hot path as the combination of $path_{L2-6}$ and $path_{L3-3}$. The thick arrows in Figure 13(a)-(c) show the hot paths detected from the global profile in Table 2 and the combined result is illustrated in (d).

Using the extracted hot paths, we can apply the loop peeling optimization to separate those paths, and the basic block layout optimization using the code positioning algorithm by Pettis and Hansen [21]. Figure 3(a) shows the result of this transformation. There are three benefits in this transformation. First, it improves the hardware branch predictions for $node_3$ and $node_6$, which are now branching in one particular direction after the transformation, and this will improve the hit ratio of hardware branch prediction. Second, it improves the I-cache locality by placing the hot connected basic blocks in a straight line. Third, it reduces the number of forward or unconditional jumps.

4. Experimental results

This section presents some experimental results showing the effectiveness of SPP in comparison to BPP. We implemented SPP in the IBM Java JIT compiler using the Ball-Larus technique as the base path profiling instrumentation technique for each structure graph. Our system is a multilevel compilation system with a mixed mode interpreter [3]. When the system detects a hot method and promotes it to the next optimization level, we performed path profiling to collect profiles on its dynamic behavior for use in the higher optimization levels.

4.1 Benchmarking methodology

All the results presented in this section were obtained on a Pentium 4 2.0 GHz uniprocessor machine with 512 MB of memory, running Windows 2000 SP2, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, version 1.3.1 prototype build. For evaluating our technique, we used SPECjvm98, SPECjbb2000 [22], and jBYTEmark. SPECjvm98 was run in the interactive mode with the default large input size, and with the initial and maximum heap sizes of 256 MB. We used the first run for each test as the active profiling period. For SPECjbb2000, we chose the configuration of one warehouse with no ramp-up time, different from the standard SPEC rule, in order to capture as much profiling as possible. For jBYTEmark, we ran each benchmark with a separate JVM.

We used the same threshold of the total profile count for each method when comparing the two results by SPP and BPP. For SPP, we divided the given total profile count into the set of structure graphs evenly. For example, if we have four structure graphs for a given method, including the outline structure graph, with the total profile count of 1,000, we collect the local profile until it reaches 250 for each graph. We applied SPP and BPP to the same hot methods for a fair comparison.

To avoid the buffer overflow problem, we used both SPP and BPP only for those methods that had fewer than 1,000 paths. When the number of paths exceeded 1,000, we used edge profiling instead of path profiling. For our benchmarks, only a few methods fell into this category.

4.2 Storage requirements

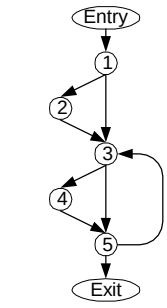
We first show the storage requirements of SPP and BPP. For Table 3, the first two columns show the total number of different paths defined by each technique to indicate the size of the buffer. The last two columns show the total number of instrumentation code blocks inserted by each technique to indicate the size of the compiled code. Instrumentation code blocks usually needs 2 to 19 instructions.

The table indicates that SPP can reduce the profile buffer size by about 21% on average compared to BPP. This is because it partitions loop regions into independent graphs and eliminates any paths across loop boundaries. For example, for the method in Figure 14, SPP creates ten paths while BPP creates twelve paths.

The total number of instrumentation code blocks required for SPP is about 26% larger on average than that for BPP. The instrumentation code block is inserted for initializing or updating the path register, or incrementing the path counter. SPP needs more instru-

Table 3: Storage requirements.

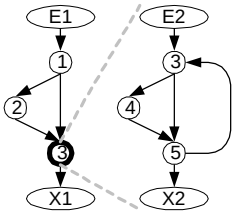
		Total number of paths		*Total number of instrumentation code blocks*	
		SPP	*BPP*	*SPP*	*BPP*
SPECjvm98	mtrt	2,617	3,370	909	807
	jess	4,332	5,534	1,642	1,420
	compress	2,142	2,829	627	560
	db	2,324	2,943	755	671
	mpegaudio	2,673	3,255	1,021	879
	jack	4,379	5,197	1,365	1,224
	javac	12,234	13,563	4,376	3,780
SPECjbb2000		7,912	9,321	2,744	2,224
jBYTEmark	Assignment	1,037	1,781	500	361
	Bit field Ops	851	1,334	366	275
	FFT	816	1,297	355	263
	FP Emulation	1,235	1,774	520	424
	Huffmann	933	1,600	419	305
	IDEA	851	1,333	377	277
	LU	956	1,553	433	314
	NeuralNet	936	1,413	434	313
	Num Sort	864	1,325	1,325	270
	String Sort	892	1,393	397	290
Total		47,966	60,815	18,565	14,657



(a) control flow graph

```
 1: Entry-1-2-3-4-5-Exit
 2: Entry-1-2-3-4-5-(3)
 3: Entry-1-2-3-5-Exit
 4: Entry-1-2-3-5-(3)
 5: Entry-1-3-4-5-Exit
 6: Entry-1-3-4-5-(3)
 7: Entry-1-3-5-Exit
 8: Entry-1-3-5-(3)
 9: (5)-3-4-5-Exit
10: (5)-3-4-5-(3)
11: (5)-3-5-Exit
12: (5)-3-5-(3)
```

(b) path list of BPP



(c) structure graphs for (a)

```
 1: E1-1-2-(3)-X1
 2: E1-1-(3)-X1
 3: E2-3-4-5-X2
 4: E2-3-4-5-(3)
 5: E2-3-5-X2
 6: E2-3-5-(3)
 7: (5)-3-4-5-X2
 8: (5)-3-4-5-(3)
 9: (5)-3-5-X2
10: (5)-3-5-(3)
```

(d) path list of SPP

Figure 14: Differences in the numbers of paths between SPP and BPP.

mentation code blocks than BPP. This is because SPP must initialize and update the path register for each structure graph separately, while BPP needs it only once for the whole method. For example, for the method in Figure 1, SPP needs ten instrumentation code blocks as shown in Figure 11, while BPP needs only five instrumentation code blocks as shown in Figure 4.

4.3 Accuracy

Our goal for path profiling is to obtain reliable information for both hot paths and rare paths, which can be used to estimate the cost and benefit of optimizations. Therefore we need to evaluate the accuracy of the overall path profiles themselves rather than just the coverage of hot paths [23].

To evaluate the accuracy of the online profiles compared to the offline version, we used the *overlap percentage* [12], the same metric as used by Arnold and Ryder [13]. That is, we evaluate what percentage of the profiles collected by SPP or BPP overlaps with its respective offline complete profiles. For example, the overlap percentage of SPP in Table 2 is 99.7%, and the overlap percentage of BPP in Table 1 is 69.7%.

Figure 15 shows the overlap percentages for the SPECjvm98 benchmark when varying the threshold value of the total profile count for each hot method. For compress and db, the overlap percentage of SPP is significantly higher than that of BPP regardless of the given threshold value of the total profile count. These two benchmarks are loop intensive, and BPP fails to profile some hot loops in several hot methods as described in Section 2.2. For mpegaudio, SPP achieves a higher overlap percentage with a smaller threshold count.

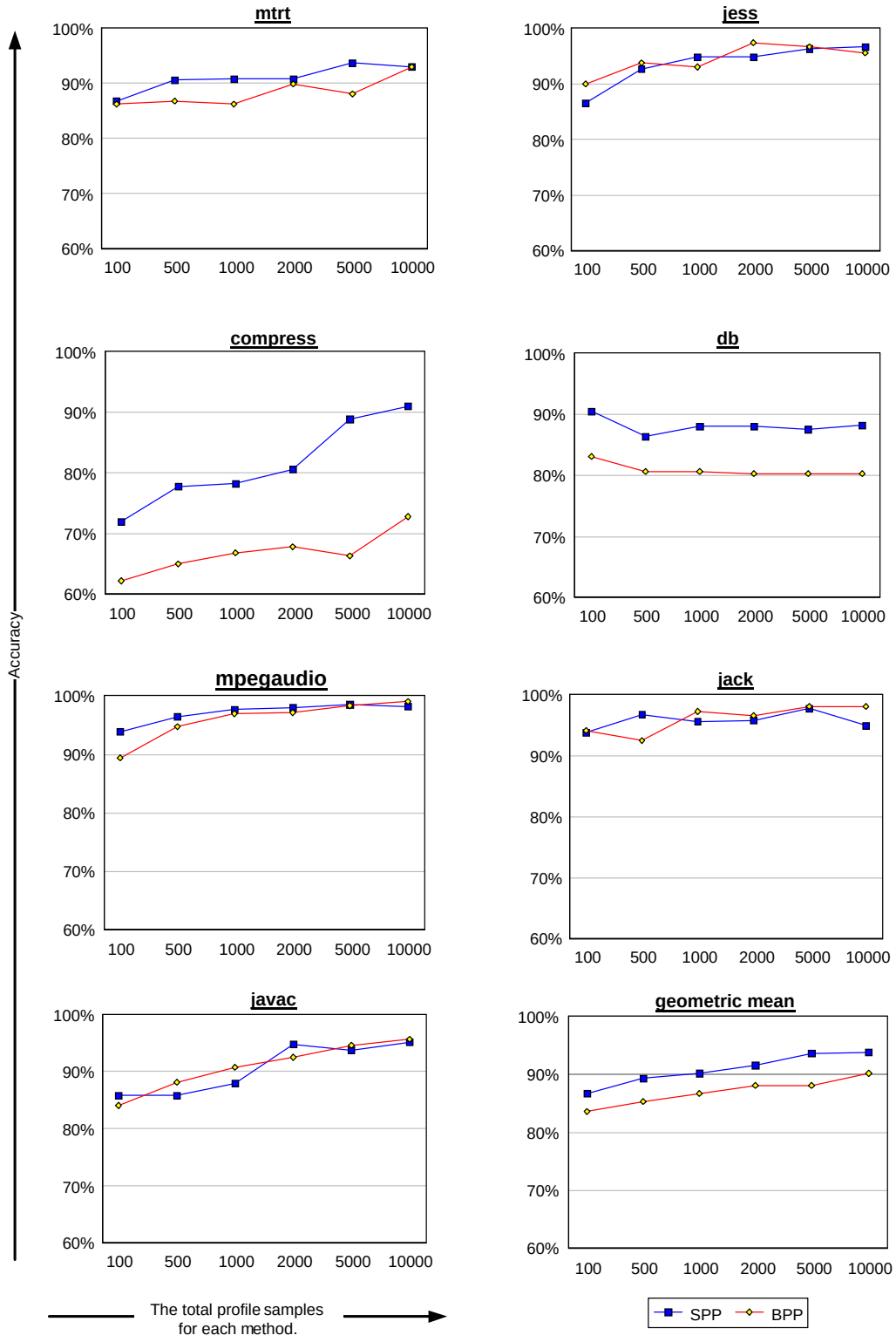


Figure 15: Comparison of accuracy (overlap percentages) of SPECjvm98 benchmarks between SPP and BPP for each threshold.

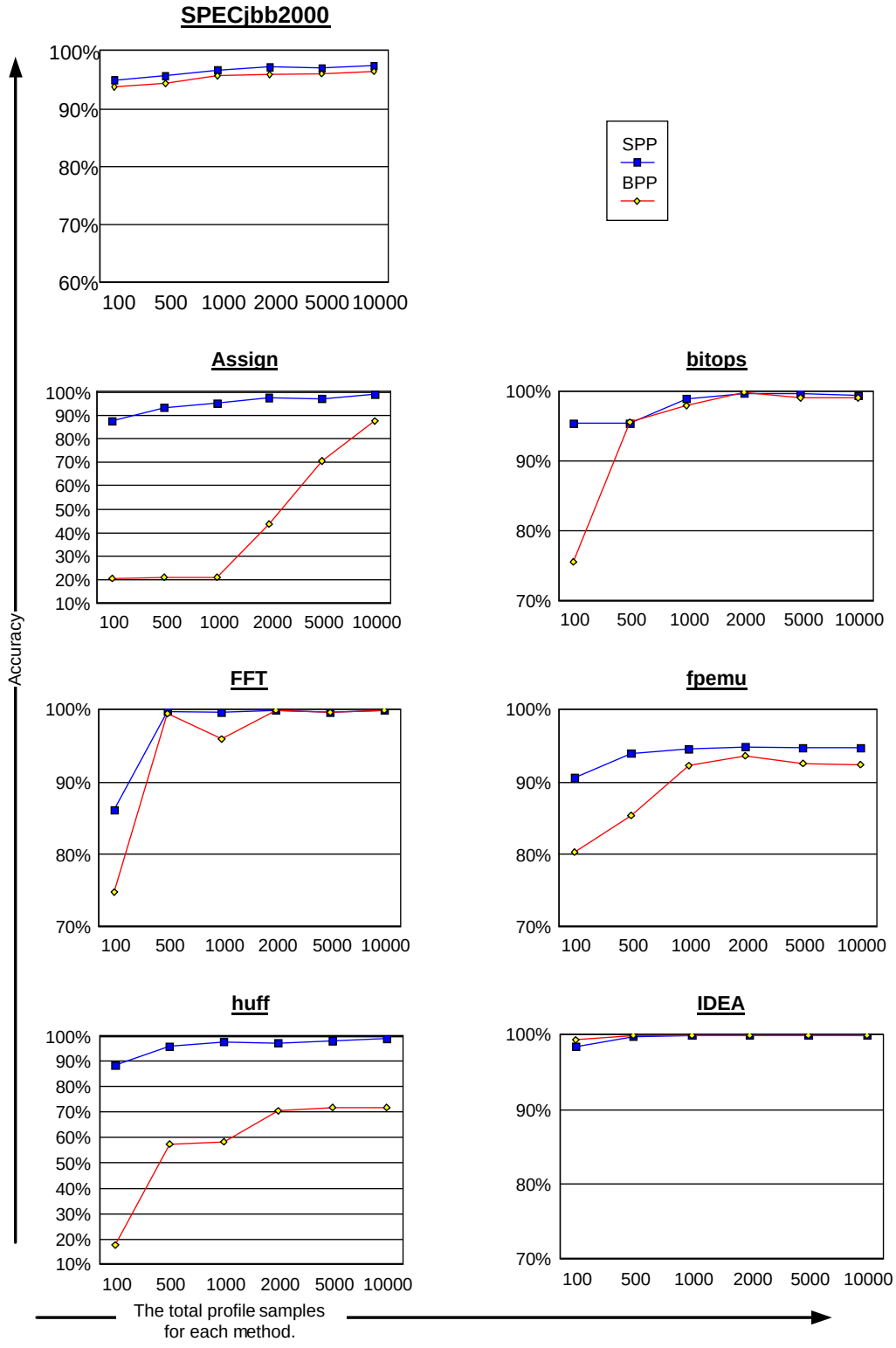


Figure 16: Comparison of accuracy (overlap percentages) of SPECjjbb2000 and jBYTEmark benchmarks between SPP and BPP for each threshold.

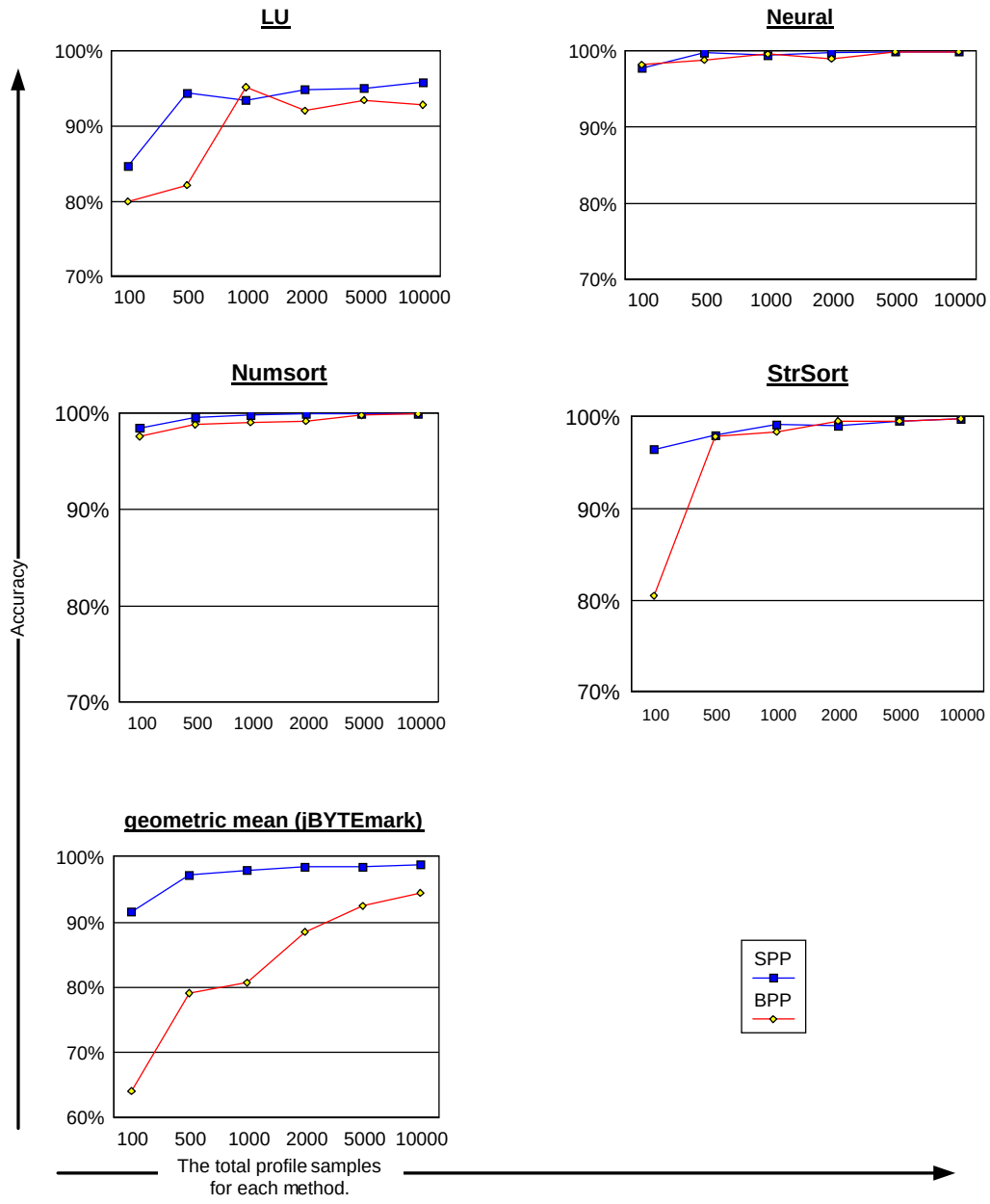


Figure 16: Comparison of accuracy (overlap percentages) of SPECjjbb2000 and jBYTEmark benchmarks between SPP and BPP for each threshold. (continued)

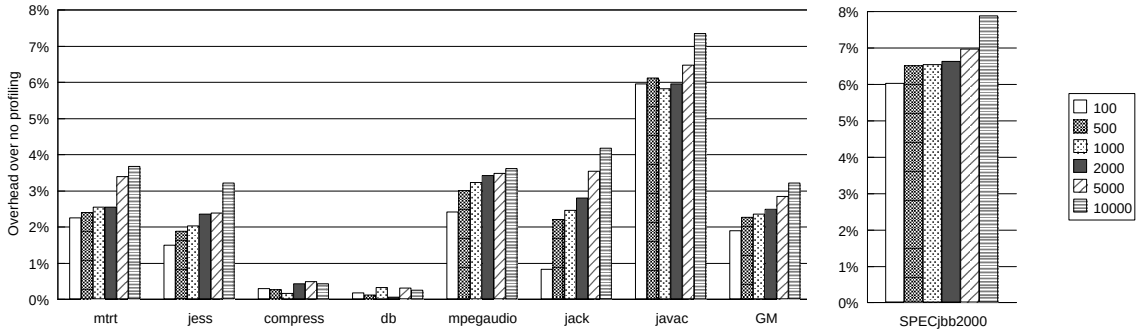


Figure 17: Overhead of SPP in the first run of SPECjvm98 benchmark and SPECjbb2000 benchmark over the same run without profiling. (smaller is better)

On the other hand, for jess, jack, and javac, SPP performs slightly worse than BPP for some thresholds. Because the total profile count is divided and distributed evenly among the structure graphs in a given method, SPP may miss some hot paths when the method contains many loops and the profile count assigned to each structure graph becomes small. This can be improved by employing a sampling technique or by adopting a variable profile count for each local profile as described in the following section.

Figure 16 shows the same comparisons for each of the SPECjbb2000 and jBYTEmark benchmark. Because most programs in jBYTEmark are numerical applications, most conditional branches were predictable and thus the accuracy was very high. The accuracy of BPP was lower than SPP especially for lower thresholds. This is due to the same reason as described in Section 2.2.

In summary, as we can see from the geometric mean for the SPECjvm98, SPP reaches an accuracy of 90% with a threshold count of 1,000 for each hot method, while BPP needs to collect an order of magnitude more samples for each method to achieve the same level of accuracy. From the geometric mean of the jBYTEmark, the accuracy of SPP is well above 90% for all ranges of the threshold count, but BPP requires 5,000 profiles to reach 90% accuracy.

4.4 Profiling overhead

We evaluated the overhead of SPP with various thresholds of the total profile count for each hot method. Although we didn't show the overhead of BPP in Figure 17, it turned out to be very close to that of SPP for each program. Thus we presented the overhead of SPP alone.

Figure 17 shows the overhead of SPP for the first run of SPECjvm98 benchmark and for the 1 warehouse of SPECjbb2000 benchmark over the same run without SPP. The first run contains the time for compilation, recompilation, various profiling activities, and program execution time. We note here that no optimization based on the collected path profiles was performed for this evaluation. Thus the result includes the possible performance degradation caused by the delay of the recompilation in the profiling period as well as the overhead of SPP itself.

The performance impact for compress and db is extremely small. This is because the number of hot methods in these benchmarks is relatively small (around 10). On the other hand, javac has a flat execution profile with many equally warm methods. The profiling overhead for a program with a flat profile can be larger than that for one with a more spiky execution profile. The overhead of SPECjbb2000 is similar to that of javac because both have flat profiles and a large number of methods must be profiled during the execution.

The overhead of SPP for jBYTEmark was too small to be shown in the graph. This is because each benchmark program has a few hot loops that are iterated for many cycles, which is enough to hide the overhead of SPP.

Overall, the overhead of SPP for SPECjvm98 is between 2% and 3% on average, which is an acceptable level for JIT compilers. Even for programs with flat profiles, such as javac and SPECjbb2000, the overhead of SPP is between 6% and 8%.

5. Discussion

Profiling by SPP would take longer than BPP, because, in the current implementation, profiling proceeds in multiple steps from the outline structure graph to the inner loop structure graphs depending on the nested level of the loops in the given method. This can be improved by profiling all of the structure graphs at once, but it would require more working buffers.

The accuracy of the collected profiles would become unexpectedly lower when we use a small profile count for each structure graph. This can happen when some loops in a method are rarely executed or there are too many structure graphs in a method. This is due to the fact that the current implementation divides the total profile count equally among the structure graphs. This could be improved by employing a sampling technique to change the sampling rate or by a dynamic mechanism to assign a different profile count for each structure graph at runtime.

An alternative would be to apply path profiling selectively to those loop structure graphs whose loop nodes in the outer graphs are found to be hot ones when collecting the early profiles. While this implementation cannot detect hot loops located in a rarely executed region of the outer structure graph, it can collect path profiles for a hot region connected from the entry point of the method with lower overhead.

6. Related work

Two path profiling techniques, one by Ball and Larus [11] and the other by Young and Smith [6], were proposed for offline use with static compilers. The Ball-Larus technique treats all the loop backedges as two separate paths, one path ending at a backedge and the other path starting at the same backedge. Their technique assigns instrumentation code blocks to the chord edges of the spanning tree of the control flow graph of the target method to minimize the number of instrumentation code blocks on each path. Then it inserts a simple arithmetic operation for each instrumentation code block to calculate the state for path profiling by using a variable called a *path register*. The final state of the path register is the path index of the traced path. They demonstrated that the overhead of path profiling can be reduced to be as small as that for edge profiling.

The Young-Smith technique defines paths as the last k branches of the execution trace at each edge. It profiles paths by constructing the path CFG, where each node represents a path in the original control flow graph and each edge represents the last edge of the profiled path. While the Young-Smith technique has higher overhead than the Ball-Larus technique, it can provide more detailed information useful for some advanced optimizations [6, 10].

The instrumentation sampling framework (we call it ISF in our paper) proposed by Arnold and Ryder [13] has both advantages and disadvantages in comparison to our SPP framework. Although we have not implemented their framework on our system for a fair comparison, we roughly estimated the accuracy of ISF by providing a counter for each profile point in BPP and executing the profile collection once every sampling interval. The result is that ISF and SPP show almost the same level of accuracy on average for each threshold count. As for the profiling overhead, it is proportional to the total amount of profiles to be collected, and thus considered the same between the two frameworks. The difference lies in that SPP executes profiling in a bursty manner, while ISF works on a sampling basis. ISF is effective for keeping the maximum profiling overhead low, but it has two possible problems. First, it incurs an additional storage requirement for the code duplication. Second, it makes the profiling period much longer than SPP, and it can affect the overall performance due to the delay of further optimizations based on the profiled information.

Chilimbi and Hirzel proposed a technique to profile hot data streams for online use in order to apply dynamic prefetching [24]. They gathered subsequences of data references and compressed them by using the Sequitur algorithm [25] to extract the hot data streams. While their technique can detect hot executing traces, it is not straightforward as to how to use them for program-restructuring optimizations.

Dynamo [26] is a dynamic optimization system, which takes binary code as input and reoptimizes it at runtime. When it identifies a hot trace in the program as an optimization target, it employs a technique called *NET* (next executing tail) [27] to reduce the total profiling overhead. Using NET, when a counter on a start-of-trace point exceeds a threshold, the next executing trace is recorded as the hot trace. While NET can identify hot paths quickly with low overhead, it is not clear whether it can provide a similar level of accuracy to our approach for analyzing the cost and benefit of applying program-restructuring optimizations.

7. Conclusions

We have described a new path profiling technique, structural path profiling (SPP), which is practical for just-in-time compilers. Our technique partitions a given method into a set of structure graphs based on the loop hierarchy, and applies path profiling for each structure graph independently. With SPP, we can achieve low profiling overhead and high accuracy of the resulting profiles.

We implemented our technique in the IBM Java Just-In-Time compiler. Our experiments show that it can provide profile information with an accuracy of around 90% with the total profile count of 1,000 for each method in comparison to the offline complete path profiles, while we can limit the profiling overhead to within 3% on average during the active profiling phase.

Acknowledgements

We thank the staffs of Network Computing Platform at Tokyo Research laboratory for implementing our JIT Compiler. We would like to thank the anonymous reviewers for their useful comments.

References

[1] M. Arnold, M. Hind, and B. G. Ryder, "Online Feedback-Directed Optimization of Java," in *In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 111–129, 2002.

[2] M. Paleczny, C. Vick, and C. Click, "The Java HotSpot Server Compiler," in *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pp. 1–12, 2001.

[3] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani, "A Dynamic Optimization Framework for a Java Just-In-Time Compiler," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 180–194, 2001.

[4] M. Kawahito, H. Komatsu, and T. Nakatani, "Eliminating Exception Checks and Partial Redundancies for java Just-in-Time Compilers," Tech. Rep. RT0350, IBM Research Report, 2000.

[5] G. Ammons and J. R. Larus, "Improving Data-flow Analysis with Path Profiles," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 72–84, 1998.

[6] C. Young and M. D. Smith, "Static Correlated Branch Prediction," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 1028–1075, 1999.

[7] R. Bodik, R. Gupta, and M. L. Soffa, "Complete Removal of Redundant Expressions," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–14, 1998.

[8] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Dead Code Elimination Using Predication," in *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pp. 102–113, 1997.

[9] R. Gupta, D. A. Berson, and J. Z. Fang, "Path Profile Guided Partial Redundancy Elimination Using Speculation," in *Proceedings of the IEEE International Conference on Computer Languages*, pp. 230–239, 1998.

[10] C. Young and M. D. Smith, "Better Global Scheduling Using Path Profiles," in *Proceedings of 31st International Conference on Microarchitecture*, pp. 115–123, 1998.

[11] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Proceedings of 29th International Conference on Microarchitecture*, pp. 46–57, 1996.

[12] P. T. Feller, "Value profiling for instructions and memory locations," Master's thesis, University of California, San Diego, 1998.

[13] M. Arnold and B. G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 168–179, 2001.

[14] T. Ball, P. Mataga, and M. Sagiv, "Edge Profiling versus Path Profiling: The Showdown," in *Proceedings of the 25th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, pp. 134–148, 1998.

[15] R. Cohn and P. G. Lowney, "Feedback directed optimization in Compaq's compilation tools for Alpha," in *Proceedings of 2nd ACM Workshop on Feedback-Directed Optimization*, 1999.

[16] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, 1981.

[17] W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, vol. 7, no. (1,2), pp. 229–248, 1993.

[18] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral Instrumentation for Lightweight Program Profiling," tech. rep., Harvard University, 1999.

[19] P. Havlak, "Nesting of Reducible and Irreducible Loops," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 4, pp. 557–567, 1997.

[20] J. K. Hollingsworth, B. P. Miller, M. R. Goncalves, O. Naim, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 201–212, 1997.

[21] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 6–27, 1990.

[22] *Standard Performance Evaluation Corporation. SPECjvm98 and SPECjbb2000.* available http://www.spec.org/.

[23] D. W. Wall, "Predicting program behavior using real or estimated profiles," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 59–70, 1991.

[24] T. M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-Purpose Program," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 199–209, 2002.

[25] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Proceedings of the Data Compression Conference*, pp. 3–11, 1997.

[26] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, 2000.

[27] E. Duesterwald and V. Bala, "Software Profiling for Hot Path Prediction: Less is More," in *Proceedings of the 9th International Conference on Architectural Support on Programming Languages and Operating Systems*, pp. 202–211, 2000.