Online Subpath Profiling

by
David Oren

A THESIS
Submitted in Partial Fulfillment of the
Requirements of the Degree of
Master of Science
in Computer Science

June 2002

Online Subpath Profiling

By David Oren

Abstract

We present an efficient online subpath profiling algorithm, OSP, that reports hot subpaths executed by a program in a given run. The hot subpaths can start at arbitrary basic block boundaries, and their identification is important for code optimization; e.g., to locate program traces in which optimizations could be most fruitful, and to help programmers in identifying performance bottlenecks.

The OSP algorithm is online in the sense that it reports at any point during execution the hot subpaths as observed so far. It has very low memory and runtime overheads, and exhibits high accuracy in reports for benchmarks such as `JLex` and `FFT`. These features make the OSP algorithm potentially attractive for use in just-in-time (JIT) optimizing compilers, in which profiling performance is crucial and it is useful to locate hot subpaths as early as possible.

The OSP algorithm is based on an adaptive sampling technique that makes effective utilization of memory with small overhead. Both memory and runtime overheads can be controlled, and the OSP algorithm can therefore be used for arbitrarily large applications, realizing a tradeoff between report accuracy and performance.

We have implemented a Java prototype of the OSP algorithm for Java programs. The implementation was tested on programs from the Java Grande benchmark suite and on the `SPECjvm` benchmark suite and exhibited a low average runtime overhead.

ACKNOWLEDGMENTS

I would like to thank Evelyn Duesterwald, Jim Larus, David Melski, Ran Shaham and Eran Yahav for their helpful comments and Alex Warshavski for his assistance in using Soot.

Last, but not least, I would like to thank Célia Alexander for her support and encouragement and for the occasional threats when it seemed things were getting nowhere.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

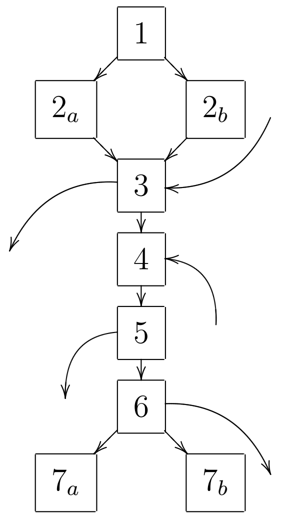
Introduction

A central challenge facing computer architects, compiler writers and programmers is to understand a program's dynamic behavior.

This understanding can be achieved by profiling. Profilers can operate on several different levels. The most basic form is *vertex profiling*, where information is recorded on the number of times each instruction (or each basic block) is executed. In *edge profiling* we are interested in the transition between basic blocks – whether the branch is taken or not. Finally, in *path profiling* the whole path the program takes during program execution (or the paths taken inside procedures) are examined.

In addition, profilers can be categorized as either *offline* or *online*. In offline profiling, results are collected during program execution, and are presented to the user after the program has stopped. In online profiling, results are either presented to the user during program execution, or directly acted upon. The classical example of online profiling are JIT compilers, where hot spots are compiled and optimized while the program is running.

In this paper we develop the first profiling algorithm with the following properties: (i) it is online, and thus well suited for JIT-like compilation and dynamic optimizations, where decisions have to be made early in order to control the rising cost of missed opportunity that results from prediction delay [**?**]; and (ii) profiling information is recorded for subpaths that start at arbitrary program points. Related works are described in Section 6.



Subpath	*Count*
[12_a34]	1724
[12_b34]	268
[2_a345]	1724
[2_b345]	268
[3456]	6042
[4567_a]	1008
[4567_b]	1864

Figure 1.1: Several cold paths sharing a common hot subpath, [3456]. This code segment may be part of a loop, or may be called numerous times from other functions.

1.1 Hot Subpaths

Considering arbitrary subpaths presents a considerable performance challenge. As the number of subpaths under consideration could be in the *hundreds of millions*, maintaining a full histogram of all subpaths is prohibitively expensive both in runtime and in memory overheads.

Figure 1.1 presents a situation where several cold paths include a common section of code [3456]. This common section is hot, even though the paths that contain it are cold.

1.2 Main Results

In this paper, we present a new online algorithm for Online Subpath Profiling, OSP, that records hot subpaths which start at arbitrary basic block boundaries. The OSP algorithm can report an estimate of the k hottest subpaths in a given program on a given run. This can be used by a programmer, an optimizing compiler or a JIT compiler to locate “hot” areas where optimizations pay off. Whereas other profiling algorithms are typically limited to certain path types, the OSP algorithm identifies arbitrary hot subpaths in the program.

The OSP algorithm is online in the sense that it reports at any point during program execution the hot subpaths as observed so far. It has very low memory and runtime overheads, and it exhibits high accuracy in reports. For example, consider the `JLex` [?] program for generating finite state machines from regular expressions. The OSP algorithm accurately identifies the 5 hottest subpaths when profiling this program on the provided sample input. The memory overhead is 45 kilobytes, compared to 170 kilobytes used by `JLex`. The runtime overhead is 64%, and could be as low as 17% with an appropriate implementation of the profiler.

The online nature of the OSP algorithm is demonstrated for the `FFT` program. At every point during its execution, the hottest subpaths observed so far are reported with high accuracy. This feature makes the OSP algorithm very attractive for use in JIT-like compilers, in which profiling performance is crucial and it is essential to locate hot subpaths as early as possible.

The `JLex` program generates approximately 22 million subpaths of length up to 1024 basic blocks. From this input a sample of about 2000 subpaths is sufficient to correctly identify the 5 hottest subpaths. Results for `FFT` are even more favorable, as elaborated in Section 4.

The OSP algorithm is based on an adaptive sampling technique presented by Gibbons and Matias [?] that makes effective utilization of memory with small overhead. Both memory and runtime overheads can be controlled, and the OSP algorithm can therefore be used for arbitrarily large applications, realizing a tradeoff between accuracy and performance. The accuracy depends on the skew level of

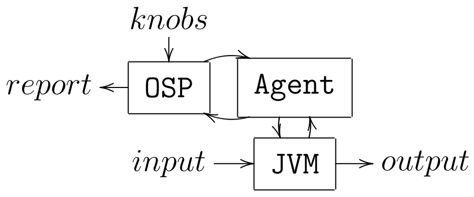


Figure 1.2: The OSP Architecture

the distribution of the subpaths. The higher the skew the better the performance, which is an attractive feature as the importance of the profiler is greater for skewed distributions.

1.3 Prototype implementation

We have implemented a simple prototype of the OSP algorithm in Java for Java programs, using the Soot [?] framework for program instrumentation. The architecture of the implementation is described in Figure 1.2. The OSP algorithm is called by a profiling agent, sitting on top of the JVM. It may accept input parameters such as available memory and a limit on runtime overhead; it continuously reports hot subpaths that can be fed back into the JVM for optimization.

We tested the algorithm on 4 programs from the Java Grande benchmark suite [?], on `JLex` [?] and on the `SPECjvm` benchmark suite [?].

We measured the runtime overhead, the memory overhead and the accuracy of the results. The runtime overhead averages less than 20%, and the memory overhead ranges from 40 to 65 kilobytes, compared to 100 to 170 kilobytes used by the programs. The OSP algorithm identifies the hottest subpaths in each of the tested programs. This shows that even for low memory and runtime overhead we can obtain very accurate reports of the program behavior.

1.4 Outline of the rest of this paper

Section 2 describes the online subpath profiling algorithm. Section 3 describes a simple prototype implementation and Section 4 the experimental results we have obtained. Possible extensions to the algorithm are described in Section 5. Related works are discussed in Section 6. Conclusions and further work are discussed in Section 7.

CHAPTER 2
The Online Subpath Profiling Algorithm

The OSP algorithm avoids the full cost of counting all subpaths by: (i) sampling a fraction of the executed subpaths, (ii) maintaining the sample in a concise manner, obtaining a sample that is considerably larger than available memory, and (iii) identifying hot subpaths and deriving a highly accurate estimate of their count from subpaths frequencies in the sample.

2.1 The Algorithm

The OSP algorithm is based on the hot-list algorithm presented in [**?**]. Given a sequence of items the hot-list algorithm maintains a uniform random sample of the sequence items in a concise manner, namely as pairs of (id, count). The sampling probability depends on the actual skewness of the data, and is adapted dynamically during execution. We extend the hot-list algorithm for subpaths, and maintain a concise sample of subpaths. At every sample point the OSP algorithm determines the length of the subpath to be sampled according to a predetermined distribution. The sampled subpath is encoded into a subpath id, and is either inserted into the resulting histogram (if it was not there already), or the subpath's count is incremented. If necessary, the sampling probability is adapted, and the elements in the sampled set are resampled.

Using concise samples ensures efficient utilization of memory. Instead of maintaining a multiset of ids, each id has a corresponding counter, and thus a frequently occurring element will not have a large memory footprint. With an allowed memory footprint m, and an average count G, the effective sample size is

$m \times G$. Thus, G can be defined as the gain obtained from using concise samples. The exact gain depends on the distribution of the elements in the input set.

The OSP algorithm's pseudo-code is given in Figure 2.1. The method `enterBlock` is triggered for each basic block and determines whether or not `sampleBlock` needs to be invoked. The `sampleBlock` method — the core of the algorithm — is executed for a very small fraction of the basic blocks, namely those which are part of a subpath selected to be in the sample. The algorithm maintains two variables: `skip`, which holds the number of basic blocks that will be skipped before the next sampling begins; and `length`, which holds the length of the subpath we wish to sample.

At the beginning of each basic block the `enterBlock` method is called. If a path is currently sampled, this method calls `sampleBlock`. Otherwise, if the next block is to be sampled (`skip` is 0), the length of the next sampled subpath is selected at random using a predetermined probability distribution.

The `sampleBlock` method appends the current basic block to the subpath which is currently sampled, using an implementation specific encoding. When this subpath is of the required length, the sampled set is updated by calling the `updateHotList` method. The `updateHotList` method is responsible for maintaining the hot-list.

The sampling probability determines the selection of `skip` in the `chooseSkipValue` method.

Note that the probability selections of `skip`, `length` and the resampling parameters are chosen so that at any given point the maintained histogram consists of a random sample representing the subpaths observed so far. The sampling can be uniform, or it can be appropriately biased, e.g., the probability of a subpath being sampled can be a function of its length.

```
void enterBlock(BasicBlock b) {
  if (sampling)
    sampleBlock(b);
  else {
    if (--skip == 0) {
      length = choosePathLength();
      sampling = true;
    }
  }
}
```

```
void sampleBlock(BasicBlock b) {
  subpath.appendBlock (b);
  if (--length == 0) {
    updateHotList(subpath.id);
    skip = chooseSkipValue();
    subpath = new SubPath();
    sampling = false;
  }
}
```

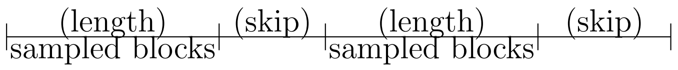


Figure 2.1: The basic OSP algorithm

Let us consider an example of the algorithm in action on the fragment of the control flow graph shown in Figure 1.1. At program startup, the OSP algorithm decides how many basic blocks should be skipped before sampling begins (using the `chooseSkipValue` function), and assigns this value to the `skip` variable. Let this value be 2. The algorithm is then called at the beginning of basic blocks 1 and 2_a, each time decreasing the value of `skip` by one.

When `skip` becomes 0, at the beginning of block 2_a, the algorithm decides how long a path should be sampled (using the `choosePathLength` function), and goes into sampling mode. Let us assume the algorithm has decided to sample a path of length 4. The next four times it is called (blocks 3, 4, 5 and 6), the algorithm will append the identifier of the current basic block to the identifier of the path being generated. Once the identifier for path [3456] has been generated, the algorithm will update the sampled set with this new subpath id. Finally, the algorithm will decide how many blocks are to be skipped before sampling begins again, and will switch to skipping mode.

Every time subpath [3456] is sampled, its count in the sample is incremented. Note that it will be sampled at a rate about 3 times the rate of subpath [4567_b], about 6 times the rate of subpath [4567_a], and over 20 times the rate of subpaths [12_b34] and [2_b345]. Also note that even for a sampling probability of about $\frac{1}{40}$, it is expected to be sampled approximately 150 times, enabling a very accurate estimate of its count.

2.2 Complexity Analysis and Runtime Overhead

The skipping overhead, in the `enterBlock` method, is $O(1)$ operations per block, with a constant depending on the exact implementation of the skipping process. The sampling overhead, in the `sampleBlock` method, is $O(1)$ operations per sampled block. The cost of table resampling is controlled by setting the new sampling probability, and can be made to be amortized $O(1)$ per sampled block [**?**]. Since the number of sampled blocks is a small fraction of the total number of executed blocks, the total sampling overhead is $o(n)$, where n is the number of executed blocks, and is $o(1)$ amortized per executed block.

Runtime overhead can be divided into two different parts: skipping overhead and sampling overhead.

Let us denote the cost of sampling a basic block by α_1. This cost is fixed for different programs and during program execution. The probability distribution of the path lengths determines an expected path length, $\bar{l}$. Finally, let ρ be the probability of starting a subpath at a given basic block. In order to quantify the program cost, let β be the average cost of a basic block in a given program and for a given run, and n the number of basic blocks.

$$\text{Sampling overhead} = \frac{\text{Sampling cost}}{\text{Program cost}} = \frac{\alpha_1 \rho \bar{l} n}{\beta n} = \frac{\alpha_1 \rho \bar{l}}{\beta} \tag{2.1}$$

Similarily, let α_2 be the cost of skipping a basic block, then:

$$\text{Skipping overhead} = \frac{\text{Skipping cost}}{\text{Program cost}} = \frac{\alpha_2}{\beta} \tag{2.2}$$

We have measured three distinct times, the uninstrumented program time:

$$t_1 = n\beta \tag{2.3}$$

the instrumented program time:

$$t_2 = n\beta + n\alpha_2 + n\alpha_1 \rho \bar{l} = n(\beta + \alpha_2 + \alpha_1 \rho \bar{l}) \tag{2.4}$$

and the running time for an instrumented program when no sampling is performed (the profiler simply skips the correct number of basic blocks, but does not sample any subpaths):

$$t_3 = n\beta + n\alpha_2 = n(\beta + \alpha_2) \tag{2.5}$$

Using these three measurements, we can compute the total overhead, as well as the skipping and sampling overhead:

$$\mu_{total} = \frac{t_2 - t_1}{t1} = \frac{n(\beta + \alpha_2 + \alpha_1 \rho \bar{l}) - n\beta}{n\beta} = \frac{\alpha_2 + \alpha_1 \rho \bar{l}}{\beta} \tag{2.6}$$

$$\mu_{skipping} = \frac{t_3 - t_1}{t1} = \frac{n(\beta + \alpha_2) - n\beta}{n\beta} = \frac{\alpha_2}{\beta} \tag{2.7}$$

$$\mu_{sampling} = \frac{t_2 - t_3}{t1} = \frac{n(\beta + \alpha_2 + \alpha_1 \rho \bar{l}) - n(\beta + \alpha_2)}{n\beta} = \frac{\alpha_1 \rho \bar{l}}{\beta} \tag{2.8}$$

We can now calculate β in two different ways and compare the results obtained by these two methods. The first is simply to divide the program execution time by the number of basic blocks executed.

$$\beta_1 = \frac{t_1}{n} \tag{2.9}$$

The second is taking the reciprocal of the skipping overhead (which is linear in β).

$$\beta_2 \sim \frac{1}{\mu_{skipping}} = \frac{\beta}{\alpha_2} \tag{2.10}$$

The values obtained by these two methods have to be normalized in order to be meaningfully compared to each other. That is, given $\beta_{1_1} \cdots \beta_{1_n}$ and $\beta_{2_1} \cdots \beta_{2_n}$, let

$$\begin{aligned} \beta_{1_{\min}} &= \min_{i=1}^{n} \beta_{1_i} \\ \beta_{2_{\min}} &= \min_{i=1}^{n} \beta_{2_i} \end{aligned} \tag{2.11}$$

We can now normalize the computed values for β, obtaining:

$$\begin{aligned} \beta'_{1_i} &= \frac{\beta_{1_i}}{\beta_{1_{\min}}} \\ \beta'_{2_i} &= \frac{\beta_{2_i}}{\beta_{2_{\min}}} \end{aligned} \tag{2.12}$$

Thus, only the relative values of the computed βs are taken into account, and the α_2 element in β_2 is cancelled out.

2.3 Special Considerations

2.3.1 Sampling and Skipping

The sampling and counting are performed using a hot-list algorithm [?]. The hot-list algorithm is given an estimate of the input size, and a permissible memory footprint. From these values an initial sampling frequency f is computed, and each subpath is sampled with probability $\frac{1}{f}$.

Let m be the permissible memory footprint, G the expected gain and n the expected input size, then

$$f = \frac{n}{m \times G} \tag{2.13}$$

Instead of deciding for each subpath whether it should be sampled or not, a *skip value* is computed [?]. This value represents how many subpaths must be skipped before one should be sampled. The skip values are chosen so that their expected value is f, and for large values of f the performance gain can be important.

2.3.2 Subpaths

For performance reasons, we observe that it is advantageous to only consider subpaths whose length is a power of two. Since the number of subpaths increases (quadratically) with the number of basic blocks, and the number of subpaths in the input affects accuracy for a given sample size, we improve performance by limiting the input set. Our choice provides significant reduction in the noise that exists in the sample set. Moreover, for any hot subpath of length k, we can find a subpath of length at least $\frac{k}{2}$ which is part of the sample space.

2.3.3 Path Length Bias

Once we have decided a subpath should be sampled, we have to decide how long a subpath should be sampled. It has been suggested that shorter hot subpaths will yield better possibilities for optimization (see [?] and its definition of minimal hot subpaths). Thus, in the current implementation we have decided to prefer shorter paths. Paths are sampled with a geometric probability distribution, with a path of length $2^n, n \geq 1$, being sampled with probability $\frac{1}{2^n}$.

Preferring shorter subpaths also increases the probability of finding *minimal* subpaths. In the case of loops, for instance, sampling longer subpaths will often yield the concatenation of several iterations of the loop.

An important feature of the algorithm is that it can accommodate other biases towards path lengths. Path length could be selected by any probability distribution; e.g., geometric (as above), uniform, or one which provides bias towards longer paths. The random selection of length is performed by the method `choosePathLength` and the algorithm works correctly for any selected distribution.

2.3.4 Concise Samples and Resampling

The hot-list algorithm maintains a list of *concise samples* of the sampled subpaths. This list can be thought of as a histogram: for each sampled subpath we hold an identifier, and a count representing how many times it has been sampled so far. Since each sampled subpath uses the same amount of memory even if it is sampled numerous times, the use of concise samples increases the effective sample size.

The benefit resulting from the use of concise samples depends on the program being profiled. Profiling a program having a small number of very hot subpaths will benefit greatly from the use of concise samples. At the other extreme, profiling a program where the subpaths are evenly distributed will not benefit from them.

If at some point during execution the sample exceeds its allocated memory footprint, f is increased, all elements in the sample are resampled with a probability $\frac{f'}{f}$ (where f' is the previous sampling probability), and all new elements are sampled with the new probability. This ensures that the algorithm uses a limited amount of memory, which can be determined before the program starts.

2.3.5 Encoding

Each basic block can be assigned a unique integer identifier. We now need a function f that given a path $P = b_1 b_2 \cdots b_n$ where b_i are basic blocks, will generate a unique identifier for the path.

Ideally, we could find a function f that is sensitive to permutation, but not to rotation. Formally, given two paths, $P_1 = b_1 b_2 \cdots b_n$ and P_2, then $f(P_1) = f(P_2)$ iff there is some j such that $P_2 = b_j \cdots b_n b_1 \cdots b_{j-1}$.

The rotation requirement requires some discussion. Assigning different ids to 'rotated' paths may enable more aggressive optimizations, since some optimizations may involve reordering the basic blocks in order to avoid branch misprediction penalties. But since we sample the program at random intervals, a loop of length n will be captured as n different paths, even if the loop is always entered from the same place. Counting these different 'images' of the loop as different paths will make the list of sampled paths more noisy.

Thus, although we would like to differentiate between paths (by assigning a unique id to each different path), an exception in the case of 'rotated' paths may be useful.

2.3.6 Reporting the Results

At any point during program execution the subpaths in the sample can be reported. It is important to remember that not all subpaths in the sample have the same accuracy. Intuitively, the higher the count of the subpath in the sampled set, the higher the accuracy of the count, and the probability that this subpath is hot.

We can either report only subpaths whose count in the sampled set exceeds some threshold, or report the k hottest subpaths in the sampled set. For each reported subpath, an estimate of its accuracy is given [**?**].

The accuracy of each path in the sampled set can also be estimated by using Chernoff bounds.

$$\Pr(x \notin (1 \pm \epsilon)E(x)) \leq e^{-\frac{\epsilon^2 E(x)}{6}} \tag{2.14}$$

Using this inequality we can find, for a given confidence, an upper bound for the error.

We are not interested in the exact count of the sampled subpath. Rather, when we report a subpath as hot, we want to be reasonably sure that it is indeed hot — if the subpath is hotter than we assume, it does not matter, since it will be reported anyway. We can use a uni-directional Chernoff bound, yielding a better result:

$$\Pr(x \notin (1 \pm \epsilon)E(x)) \leq e^{-\frac{\epsilon^2 E(x)}{2}} \tag{2.15}$$

2.4 A Framework for Profilers

The description of the algorithm given here is very general. The behavior of the algorithm can be modified extensively by changing certain elements. Hence, the algorithm can serve as a framework for profiling under various preferences or constraints.

It is very important to remember that many of the decisions presented here — limiting ourselves to paths of length 2^n, giving a higher sampling probability to shorter paths, for instance — are implementation details, and do not stem from any limitation in the algorithm itself.

It would be very easy to collect information on paths of arbitrary length, or on any different subset of paths — for instance, paths of length 1.5^n. Another possibility is to modify the counting method to more accurately identify changes in the working set of the profiled program. This could be done using a sliding window that would take into account just the latest encountered subpaths, or with an aging function that would give more weight to more recent subpaths.

CHAPTER 3

Prototype Implementation

We have implemented a prototype in Java, using the Soot framework [?].

In the prototype implementation, profiling a program consists of two steps: first, the program to be profiled is instrumented. The class files are processed, and calls to the subpath profiler are added at the beginning of each basic block. Once the program is instrumented, it can be run and profiled on any given input. Instrumentation could also be performed dynamically, by modifying the Java class loader.

Multi-threaded programs are handled by associating a different subpath profiler with each running thread. This guarantees that subpaths from different threads are kept separately, and also reduces synchronization overhead between the different threads. The invocations of the `updateHotList` method are synchronized. Our initial experience indicates that this does not create synchronization overhead, since this method is rarely invoked.

Since we are not notified when a thread ends, we periodically check whether the thread associated with each subpath profiler is still active, and if not, we make the subpath profilers eligible for garbage collection.

In the prototype implementation, we did not implement JIT-like optimizations. Instead, when the JVM exits, a report is generated. For each path in the sampled set, its description and count are displayed.

In the current implementation the `enterBlock` method is part of the Java code. Hence it becomes the dominant factor in the total runtime overhead. A preferred implementation would be to have this method run in the JVM itself, in which case the sampling overhead is expected to become dominant. Therefore, in the measurements we have considered these two overheads separately.

3.1 Path Representation

For the reference implementation, we did not focus on path representation, and only implemented a simple path representation scheme. Path description is kept as a list of strings, each string describing a basic block. The lists are generated dynamically and entail some overhead, especially for long paths.

It is important to remember that these descriptions are not strictly necessary. If the OSP algorithm is used in a JIT compiler, no output is necessary, and the descriptions of the hot subpaths are of no interest — each subpath can be identified with a unique integer id.

However, even if these descriptions are required, they are not needed during program execution, but only when the report is displayed. Therefore, if memory becomes an issue, a possible solution would be to keep the path descriptions not in memory, but in secondary storage. Each path description would have to be written to the disk only once, thus maintaining time overhead at acceptable levels.

More complete solutions would involve developing a memory efficient representation of the paths: for instance, a naive subpath description could contain a description of the block where it begins, and for each subsequent branch a bit signifying whether this branch was taken or not. A path of length n would thus require $c + (n - 1)$ bits for its description, where c is the number of bits required to store the identifier of the starting basic block. Since the Java bytecode contains multiple branch instructions (used with the `switch` construct) the actual encoding would have to be more complex.

A different solution altogether would be to represent the subpaths using tries. With tries it will be possible to check whether a subpath is already part of the sampled set in an online manner, by "walking down" the trie. Once we have determined a subpath has already been sampled, increasing its count amounts to increasing the count in the node of the trie that is associated with the last block of the subpath. Adding a new path can also be done online, by walking down the trie along the longest prefix of the subpath that can be found in the try and then adding a new branch in the trie.

However, using tries will require a way to convert paths to a canonical form, to make sure the trie is not sensitive to rotation. The problem can be subdivided into two smaller ones, one involving intra-procedural paths, the other involving inter-procedural ones. For intra-procedural paths, once a subpath has been sampled, the basic blocks can be "rotated" back to the correct ordering (that is, the one where the basic block closest to the procedure's beginning is first). That way we can ensure a given path is always inserted into the trie in the same order.

For inter-procedural paths, a generalization of this method is required. The correct ordering could be defined as the ordering where the basic block that is closest to its procedure's beginning is first. Each procedure could be given a numeric identifier (possibly a hash value), to deal with the cases where two basic blocks have the same ordering value. That way paths will always be represented in a canonical way.

When inserting a path into the trie, the algorithm would have to keep track of the ordering value of the inserted blocks. If at some point a lower ranking block is inserted into the trie, the insertion process will reset, the path will restart its insertion from that basic block. The algorithm will have to keep track of what basic blocks were inserted to the trie in the beginning and are now to be inserted into the trie once sampling ends.

3.2 Encoding

The encoding of subpaths determines how subpaths are grouped together for purposes of counting and reporting. The current implementation uses an encoding consisting of the subpath length, and of running the exclusive-or operator over block identifiers. This encoding is simple, efficient, and groups together different permutations of the same path.

The exclusive-or encoding has a significant drawback: it disregards blocks that occur an even number of times. The problem is partially solved by making the length part of the path identifier. This reduces the problem, but does not solve it completely: for instance, the paths [*abbc*] and [*addc*] will be counted as one path.

Alternative Hash Functions One possibility is to use a simple linear hash-function, $f(x) = a \times x + b$. The problem is that this is sensitive to permutation, so [*abcd*], [*bcda*], [*cdba*] and [*dabc*] are counted as different paths. This method of counting paths introduces a lot of noise into the sample space.

This noise can be avoided by using a simple variant of the linear hash function. Assume we sampled a path p_1, of length n, and containing the basic blocks $b_1 \cdots b_n$. Let us sort the identifiers of these blocks, so that $b_1 \leq \cdots \leq b_n$. Now we can run the linear hash function on $b_1 \cdots b_n$.

This is not a perfect solution. If a subpath is a loop then we will calculate the same identifier, no matter how it is sampled, since we sort the ids of the basic blocks. However, information about order is lost (just like with the XOR encoding). The important thing is that the encoding will not lose information about blocks that occur an even number of times.

Implications of the XOR Encoding In order to evaluate the quality of the results, we have run the profiler with a different encoding as well. These tests have shown that the results obtained by the exclusive-or encoding are correct, in spite of its drawback.

Although the XOR encoding does lose information, it has no noticeable effect on the results. First, for all subpaths of length 2, no problem arises. The problem is for longer subpaths. We have run the profiler on FFT with the linear hash function and compared results with what we obtained using XOR.

After sorting out through the noise created by the linear hash function, we found out that the results match.

CHAPTER 4

Experimental Results

We have run the profiler on four programs from the JavaGrande benchmark suite [?], on the `JLex` utility [?] and on the `SPECjvm` benchmark suite [?]. All programs were run on a computer with a 1.2GHz Athlon processor, and 512MB of memory running Sun's JDK 1.3.1 on Windows 2000.

Table 4.1 shows the sizes of those programs. It is important to remember that from the profiler's view, what matters is not the number of lines of code in the program, but the program's *dynamic* size (its trace length).

The table also displays the number of subpaths encountered during program execution, as well as the number of distinct subpaths encountered. The subpaths are those of length 2^n, where $n \leq 5$. For `JLex`, it was also possible to obtain accurate results for paths of length up to 1024. This was not done for the other programs, since extremely long runtimes would have been needed.

These results show the size of the input data set over which the OSP algorithm works. It is also interesting to note that, even for a very limited subpath length, obtaining accurate results required an extremely large amount of time — more than an hour for `FFT` and `HeapSort`, almost ten hours on `MolDyn` and `RayTrace`.

We also tried to obtain accurate results for paths up to a length of $65,536$ for `JLex`. That particular run failed to complete even after more than 100 hours of running time.

Program	Basic blocks	Subpaths	Distinct subpaths
JLex (1024)	2,212,208	22,120,044	828,772
JLex (32)	2,212,208	11,060,983	37,985
FFT	169,867,487	849,337,378	870
HeapSort	124,039,672	620,198,303	1,095
MolDyn	1,025,640,629	5,128,203,088	6,316
RayTrace	1,367,934,068	6,839,670,283	6,800
check	63,551	317,698	6,308
compress	1,408,896,503	7,044,482,458	17,801
jess	403,421,451	2,017,107,198	57,271
db	208,593,066	1,042,965,273	5,104
javac	247,350,713	1,236,753,508	649,191
mpegaudio	724,584,925	3,622,924,568	40,227
mtrt	646,188,073	3,230,940,308	32,356
jack	63,589,082	317,945,353	72,959

Table 4.1: For each program we show the number of basic blocks encountered during execution, the number of subpaths of length 2^n where $1 \leq n \leq 5$ and the number of distinct subpaths. For JLex there are two separate entries, one showing the number of subpaths of length up to 32, the other the number of subpaths of length up to 1024.

4.1 Runtime Overhead

Table 4.2 shows the average and the standard deviation of the running time of original program abd the instrumented program and the time spent in skipping mode.

Table 4.3 shows the runtime overhead of the profiler. The total runtime overhead ranges from 20% to 360%. The sampling overhead (the overhead generated by the `sampleBlock` method) is much smaller, ranging from 1% to 56%.

Most of the runtime overhead is created by the skipping process. If the profiler is incorporated into the JVM — for instance, in order to use it for JIT compiling — the skipping process will have much lower overhead. In such a case, the total runtime overhead will be similar to the sampling overhead presented here.

Further understanding of the overhead created by the profiler can be gained by examining the first section of the JavaGrande benchmark suite. These benchmarks check raw performance of the JVM, by measuring how many operations of various kinds are performed per second (Table 4.4). For instance, a loop containing additions of `int`s will see a ten fold slow-down. On the other hand, a loop containing divisions of `long`s will slowdown only by a factor of 1.18. Creating an array of 128 `long`s will have an even smaller slowdown factor of 1.04. This is in line with Equations 2.1 and 2.2.

As was mentioned earlier, the analysis of runtime overhead can be verified by computing the average cost of a basic block using two methods and comparing the results. Table 4.5 shows two values of β, the average cost of a basic block. β_1 was computed by dividing the execution time by the number of basic blocks, while β_2 was computed using the skipping overhead (as can bee seen in Equations 2.9 and 2.10) as well as the computed error.

Program	Orig.	Stdev	Inst.	Stdev	Skipping	Stdev
JLex	0.418	0.004	0.807	0.030	0.610	0.000
FFT	21.424	0.026	29.185	0.030	25.823	0.050
HeapSort	2.141	0.008	6.503	0.051	5.310	0.004
MolDyn	10.635	0.016	36.220	0.120	32.822	0.008
RayTrace	10.829	0.013	49.893	0.084	45.443	0.009
check	0.390	0.000	0.590	0.030	0.474	0.005
compress	14.248	0.058	53.280	0.051	51.591	0.078
jess	5.135	0.004	16.335	0.008	15.826	0.008
db	24.633	0.058	29.582	0.163	29.289	0.043
javac	10.815	0.019	19.329	0.234	18.548	0.020
mpegaudio	9.069	0.005	29.370	0.026	28.615	0.017
mtrt	5.558	0.021	25.565	0.533	22.993	0.048
jack	4.788	0.004	6.697	0.016	6.572	0.022

Table 4.2: The average running time and the standard deviation of the original program, the instrumented version and the time spent in skipping mode.

Program	Time	Instr.	Only-sampling	Tot. Overhead	Sampl. Overhead
JLex	0.418	0.807	0.610	93.06%	47.13%
FFT	21.424	29.185	3.362	36.23%	15.69%
HeapSort	2.141	6.503	1.193	203.74%	55.72%
MolDyn	10.635	36.220	3.398	240.57%	31.95%
RayTrace	10.829	49.893	4.450	360.74%	41.09%
check	0.390	0.590	0.116	51.28%	29.74%
compress	14.248	53.280	1.689	273.95%	11.85%
jess	5.135	16.335	0.509	218.11%	9.91%
db	24.633	29.582	0.293	20.09%	1.19%
javac	10.815	19.329	0.781	78.73%	7.23%
mpegaudio	9.069	29.370	0.755	223.85%	8.32%
mtrt	5.558	25.565	2.572	360.01%	46.29%
jack	4.788	6.697	0.125	39.87%	2.61%

Table 4.3: The running time in seconds of the original and the instrumented programs, and the time the algorithm spent in sampling mode. The two last columns display the total runtime overhead, and the overhead generated by the sampling process itself, without taking into account the cost of deciding when to sample a path.

Benchmark	Ops/sec	Instrumented Ops/sec	Performance Ratio
Add:Int	7.8e8	7.4e7	10.48
Add:Float	7.7e7	4.1e7	1.87
Mult:Int	2.6e8	6.5e7	3.98
Mult:Double	10e7	6.4e7	1.55
Assign:Same:Scalar:Local	1.4e9	8e7	17.75
Cast:IntFloat	8e7	5.4e7	1.48
Create:Array:Long:1	1.5e7	1.2e7	1.23
Create:Array:Long:128	3.8e5	3.7e5	1.04
Loop:For	2.3e8	1.8e7	12.35

Table 4.4: Varying Runtime Overhead. This table shows the number of operations per second performed by the original and by the uninstrumented programs, for various kinds of operations. For each kind of operation, a ratio of the numbers is given, and this number gives us an estimate of the runtime overhead of the instrumentation for this kind of operation.

As we can see, the values of β computed by the two methods match, except for the two shortest programs (`JLex` and `check` from the `SPECjvm` suite). The difficulty with these two programs is that since they are both quite short, the measurements are affected by initialization time.

4.2 Sampling and Efficiency Tradeoff

Table 4.6 displays the number of sampled subpaths as recorded by our implementation of the OSP algorithm. The second and third columns are the number of sampled subpaths with and without repetitions. The Gain column displays the average count of a subpath in the sampled set, i.e., the gain obtained by using concise samples. The f column shows the sampling frequency, as defined in Equation 2.13.

We impose a minimum limit on f, since low values of f generate high overhead and do not contribute to the accuracy of the results being obtained.

Program	β_1	β_2	Error
JLex	23.87	6.96	70.85%
FFT	15.93	15.57	2.29%
HeapSort	2.18	2.16	0.96%
MolDyn	1.31	1.53	16.97%
RayTrace	1.00	1.00	0.00%
check	696.18	14.57	97.91%
compress	1.15	1.20	4.34%
jess	1.44	1.51	4.35%
db	13.40	16.60	23.91%
javac	4.96	4.39	11.54%
mpegaudio	1.42	1.46	2.51%
mtrt	1.00	1.00	0.00%
jack	8.54	8.42	1.43%

Table 4.5: The average cost of a basic block, computed by dividing the execution time by the number of basic blocks (β_1) and using the skipping overhead (β_2). The error is computed relative to β_1. The values of β have been normalized for the two groups of tests separately.

Program	# subpaths	# distinct subpaths	Gain	f
JLex	2,183	891	2.45	1,000
FFT	168,885	314	537.85	1,000
HeapSort	10,217	475	21.50	12,304
MolDyn	2,530	353	7.17	400,000
RayTrace	5,276	443	11.90	260,000
check	1907	175	10.90	25
compress	1202	357	3.37	1127116
jess	465	317	1.47	806842
db	2023	213	9.50	104296
javac	517	421	1.23	494704
mpegaudio	736	346	2.13	966112
mtrt	477	381	1.25	1279782
jack	472	326	1.45	127178

Table 4.6: The number of subpaths in the sample with and without repetition, the gain obtained by using concise samples (the ratio between columns two and three), and the sampling frequency f at the end of the program.

This was important for the FFT program, where the gain is very high. In the original FFT run, for instance, the sampling probability was one in 40. The results were similar, but the total runtime overhead was 145% (compared to 36% in the final run), and the sampling overhead was 102% (compared to 15%).

As has already been mentioned, the OSP overhead does not depend only on the sampling probability. The HeapSort program performs very simple operations on integers (comparisons and assignments). Since the cost of sampling, relative to these simple operations, is high, the sampling overhead is higher for this program than for others.

4.3 Memory Overhead

Table 4.7 shows the memory overhead of the profiler. The programs' memory footprint (for both the instrumented and the uninstrumented versions) was measured at the end of the execution. The programs' memory footprint varies between 100 and 200 kilobytes, and the profiler's is about 50 kilobytes. For simplicity, we used a straightforward representation of sampled subpaths. Thus, the actual memory required during a profiling run may be higher. With a different implementation this can be avoided, as suggested earlier in this section.

4.4 Accuracy of Results

Table 4.8 compares the results obtained by the OSP implementation with results obtained for a profiler, that collects information about all subpaths (with no sampling). For brevity, we only show the results for FFT. Similar results were obtained for JLex.

For each subpath, an estimated count was computed, by dividing its count

Program	Program footprint	Instrumented footprint	Overhead
JLex	169,728	213,032	43,304
FFT	107,416	147,168	39,742
HeapSort	107,400	156,360	48,960
MolDyn	111,800	152,664	40,864
RayTrace	108,016	173,816	65,800

Table 4.7: Memory usage of the different programs. The instrumented memory does not take into account the memory needed for maintaining the output of the algorithm.

Sample rank	Exact rank	Sample count	Est. count	Exact count	Error	Est. Error	Length
1	1	27,006	108,024,000	109,051,898	0.94%	2.58%	4
2	2	6,479	103,664,000	103,782,188	0.11%	5.27%	16
3	3	12,841	102,728,000	101,713,904	1.00%	3.74%	8
4	4	39,545	79,090,000	79,691,780	0.76%	2.13%	2
5	6	921	14,736,000	14,679,016	0.39%	13.98%	16
6	11	4,322	8,644,000	8,388,604	3.04%	6.45%	2
7	12	4,226	8,452,000	8,388,520	0.76%	6.53%	2
8	10	4,200	8,400,000	8,388,608	0.14%	6.55%	2
9	9	4,155	8,310,000	8,388,608	0.94%	6.58%	2
10	8	4,022	8,044,000	8,388,608	4.11%	6.69%	2

Table 4.8: For the hottest paths in the sample we show their true rank as obtained by counting all subpaths, their count in the sample and in the full results, their estimated count and the error in the estimation. For each path we also show its length. The table is sorted by estimated count.

in the sample by the sampling probability and by the a priori probability of sampling a path of that length.

For instance, the hottest subpath in the sample was sampled $27,006$ times. We divide this number by the a-priori probability of sampling a path of length 4 (that is, by $\frac{1}{4}$), and by the probability of sampling a path ($\frac{1}{1000}$). This yields the count estimate.

The table shows, for each of the ten hottest subpaths in the sample, its rank in the accurate results. We can see that the estimated count is very close to the accurate one. For example, the count of the hottest subpath was estimated with a precision of 0.94%, and of the second hottest with a precision of 0.11%.

In addition to the actual error we have computed an estimated error, using Chernoff bounds (see Equation 2.14). According to the Chernoff bounds, the error should be lower than the one computed, with a probability of over 95%. We can see that in almost all cases, the Chernoff bound is pessimistic, and that the actual error is much lower than expected.

In spite of the profiler's preference for short paths, we can see that the hottest paths were of non-trivial length.

Table 4.9 shows the error estimates obtained using Chernoff bounds on the hottest subpath identified for each program. From the FFT example we can assume that the actual error is much lower.

σ_1 is an upper bound for the error, with a confidence level of 95%. σ_2 is an upper bound for the error, for the same confidence level, but using a uni-directional Chernoff bound. σ_3 and σ_4 are upper bounds for the error (bi-directional and uni-directional), with a confidence level of 80%.

We can learn even from relatively inaccurate measurements, like the `mtrt` program. On that particular case, the most frequently occurring subpath was

Program name	Count	σ_1	σ_2	σ_3	σ_4
JLex	109	40.64%	23.46%	29.76%	17.18%
FFT	27,006	2.58%	1.49%	1.89%	1.09%
HeapSort	825	14.77%	8.53%	10.82%	6.25%
MolDyn	279	25.40%	14.66%	18.60%	10.74%
RayTrace	393	21.40%	12.36%	15.68%	9.05%
check	191	30.70%	17.72%	22.49%	12.98%
compress	48	61.24%	35.36%	44.85%	25.90%
jess	17	102.90%	59.41%	75.37%	43.51%
db	127	37.65%	21.74%	27.57%	15.92%
javac	13	117.67%	67.94%	86.19%	49.76%
mpegaudio	42	65.47%	37.80%	47.95%	27.68%
mtrt	6	150.00%	86.60%	109.87%	63.43%
jack	12	122.47%	70.71%	89.71%	51.79%

Table 4.9: For the hottest path in the sample we show an estimate of their count error as obtained using Chernoff bounds.

sampled 6 times. As we've seen, using bi-directional Chernoff bounds with a confidence level of 95% we get an error estimate of $\sigma_1 = 150\%$. Thus, we know with very high probability that the expected value of the sampled count is probably no higher than 15. The converse is also true — had there been a subpath whose expected count was 15, its count would have been higher than 6 with very high probability. Thus, we can learn that there are probably no subpaths with a higher count.

In order to estimate the quality of the results, it is not enough to look at the hottest subpath in the sample. Table 4.10 shows the four error estimates for the fifth hottest subpath located in each program. We can see that the results are still accurate enough.

It is important to stress that better results could be obtained by sampling more elements from the input set. Obtaining more accurate results will require

Program name	Count	σ_1	σ_2	σ_3	σ_4
JLex	59	55.23%	31.89%	40.46%	23.36%
FFT	4,289	6.48%	3.74%	4.74%	2.74%
HeapSort	441	20.20%	11.66%	14.80%	8.54%
MolDyn	251	26.78%	15.46%	19.61%	11.32%
RayTrace	206	29.56%	17.07%	21.65%	12.50%
check	96	43.30%	25.00%	31.72%	18.31%
compress	30	77.46%	44.72%	56.74%	32.76%
jess	11	127.92%	73.85%	93.69%	54.09%
db	102	42.01%	24.25%	30.77%	17.76%
javac	8	150.00%	86.60%	109.87%	63.43%
mpegaudio	18	100.00%	57.74%	73.24%	42.29%
mtrt	5	189.74%	109.54%	138.97%	80.24%
jack	7	160.36%	92.58%	117.45%	67.81%

Table 4.10: For the fifth hottest path in the sample we show an estimate of their count error as obtained using Chernoff bounds.

higher overhead, however.

4.5 Incremental Results

The algorithm can, at any point during program execution, give an estimate of the hottest subpaths encountered so far. In order to test this capability, we have stopped the `FFT` example at several equally spaced points. At each of these points, we took the 5 hottest subpahts in the accurate subpath count, and checked their rank in the report of the sampling profiler. We can see in Table 4.11 that during program execution the intermediary results obtained by the sampling profiler match the "true" results obtained by a full count of all subpaths with high accuracy. Similar results were obtained for `JLex`.

True Rank	6%	12%	18%	24%	30%	36%
1	2	6	1	2	2	1
2	3	4	2	1	1	2
3	1	2	3	3	3	4
4	4	1	8	4	4	3
5	5	5	7	5	5	5

Table 4.11: Stops after every 10 millions blocks. At each stop point, we show the rank in the sample of the 5 highest ranking subpaths in the full count. Note that the 5 highest ranking subpaths are not necessarily the same at each stop point

4.6 Arbitrary Length

In order to perform a sanity check on our decision to limit ourselves to paths of length 2^n, we have run a different version of the profiler, which is able to sample paths of arbitrary lengths. The length of the paths sampled varies from 2 to 1024, with the probability of selecting a path of length n being approximately $\frac{1}{10n}$.

As expected, the results were much more noisy, with the hottest subpaths being sampled no more than 3 times. In spite of this, the results are acceptable, with the hottest subpaths corresponding to those obtained when the path lengths where limited to 2^n.

Still, the low count of the results means they are not accurate with high probability. Therefore, running the OSP algorithm with arbitrary path length would require a larger sampling probability, and a larger memory overhead, to make sure paths are sampled often enough for results to be meaningful.

In order to obtain more complete results, while still keeping the noise to a relatively low level, we have run the profiler with yet another length distribution. In addition to paths of length 2^n we have decided to allow paths of length 3 and 5 (with the distribution being $\frac{1}{2}, \frac{1}{4}, \ldots$).

In the FFT example, the profiler identified hot subpahts of lengths 3 and 5, at the expense of longer subpaths. The 15 hottest subpaths included 3 subpaths of length 3 and 1 of length 5. It is interesting to note that these additional subpaths were concatenations of several loop iterations (shorter versions of which were already identified), and in this specific case did not add any new information.

CHAPTER 5
Algorithmic Extensions

The algorithm can be adapted to different requirements by changing the way the subpaths are counted. Also, the algorithm can be adapted to report subpaths which exceed a predetermined threshold, instead of the k hottest ones.

5.1 Weighted Paths

The algorithm can be adapted to different requirements by changing the way the subpaths are counted.

As was already seen, the algorithm samples shorter subpaths with higher probability. This probability distribution represents our belief that shorter paths may be more interesting. This probability distribution may be modified to reflect other needs.

Other than changing the sampling probability of paths of different lengths, there are several other options for introducing weights into the subpath sampling algorithm.

5.1.1 Weighing Paths by Length

The algorithm as presented so far counts the number of times the different hot subpaths occur in the program. These subpaths may be of different lengths, but the algorithm does not take this into account.

It may be that we find longer subpaths more interesting, as they enable understanding of the program flow. On the other hand, as was pointed out by Larus [?], since some compilers excel at small, local optimizations, we may be interested in the shorter hot subpaths (what he terms *minimal subpaths*), which present a great target for optimization.

It is obvious that in either case we should treat differently paths of different lengths. The easiest way to achieve that goal is to modify the way the algorithm counts the sampled subpaths. Each time a subpath is encountered, its count will be increased not by one, but by $f(n)$, where n is the length of this subpath.

Despite its simplicity, this is a very general method. We could modify f and adapt it to the results we find interesting. It should be emphasized that there is no "correct" weight function and that different applications will most probably require different weight functions.

5.1.2 A priori cost of paths

Until now we have assumed that a subpaths is a sequence of basic-blocks, and that all basic-blocks have the same "cost". When we sample a path we take into consideration only the number of basic-blocks it contains, and not the length of time it takes to execute.

This is a simplification of reality. Different basic blocks may have different execution costs, for various reasons: they may be of different lengths, or they may contain instructions that have different costs themselves (for instance, consider the high cost of I/O operations).

If two subpaths happen an equal number of times during program execution, they will (with high probability) be sampled an almost equal number of times. It would be better to consider the path which has a longer execution time hotter than the other.

In order to achieve this goal, basic blocks can be given an a priori cost during instrumentation. When a path of length n has been sampled, we can define its cost as the average of the cost of the basic blocks it contains, thus:

$$\beta_p = \frac{1}{n}\sum_{i=1}^{n}\beta_i$$

and path with cost β_p may be rejected *before* it is counted, with the probability of rejection being:

$$p_{reject} = 1 - \frac{\beta_p}{\beta_m}$$

where β_m is the cost of the most expensive basic block.

This will ensure that expensive paths will be sampled more often than inexpensive ones.

sThe a priori cost of a basic block can be computed during program instrumentation, and will therefore not incur any overhead during program execution.

5.2 Using Thresholds

The algorithm as described so far samples to hottest subpaths occurring during program execution. A different approach would be not to limit subpath sampling and reporting according to the rank of the subpaths.

Instead of a hot-list algorithm, an iceberg query algorithm can be used [?]. This class of algorithms can be used to find elements which occur in some input data set above some specified threshold. The algorithm's initial parameters determine the count above which elements should be reported (and their report probability), and the count below which they should not be (and the matching probability).

Using this class of algorithms would enable us to report the subpaths which exceed a predetermined threshold, instead of the n hottest ones in the program.

However, as was seen in our results, different programs may have vastly different subpath distributions, and some fine-tuning of this threshold may be needed for each program.

5.3 Subpath Correlation

One of the main advantages of the OSP algorithm over other methods is that it can cross loop and procedure boundaries. The Ball and Larus path profiler loses information about the context of a path and its correlation to other paths.

For example, consider a loop which contains an if-clause, which separates odd from even iterations. The subpath profiler will sample two hot subpaths, one for the behavior occurring for odd iterations, one of the behavior occurring for even ones. However, the subpath profiler will do more than that. Another hot subpath that will be sampled is the subpath consisting of the concatenation of these two behaviors. An optimizing compiler could use this information to create a specialized unrolled version of the loop that would not contain branching instructions.

CHAPTER 6
Related Work

The original Ball-Larus path profiling algorithm recorded the execution frequency of intraprocedural, acyclic paths [?]. The program was instrumented in such a way that each path would generate a unique identifier during program execution.

Ammons, Ball and Larus extended acylic path profiling [?]. They associated hardware metrics other than execution frequency with paths. They also introduced a runtime data structure to approximate interprocedural paths. In practice [?] these linkages were imprecise, and this method does not connect paths across loop iterations.

Another interprocedural extension of the Ball-Larus path profiling technique is described by Melski and Reps [?]. Paths in this technique do not cross loops. Interprocedural paths are assigned a unique identifier statically.

Larus [?] later described a new approach to path profiling, which captures a complete picture of the program's dynamic behavior. He introduced *whole program paths*, which are a complete compact record of a program's entire control flow. A whole program path crosses both loop and procedure boundaries, and so provides a practical basis for interprocedural path profiling. Since the whole program path can be quite large (hundreds of megabytes), it has to be compressed, and compression is achieved by representing the WPP as a grammar. The grammar is over an alphabet of symbols representing acyclic paths, but the algorithm can be adapted to run over an alphabet of symbols representing vertices or edges.

Once the WPP for a program has been collected and compacted, it is possible to run different analyses on this representation of program flow. Larus presents one such analysis, which identifies hot subpaths. The WPP approach requires two stages: data collection and analysis. Hence, it cannot be used by a JIT compiler to locate hot subpaths during program execution.

Duesterwald and Bala [?] analyze online profiling and its application to JIT compilation. Online profiling is a different challenge than offline profiling: the longer the program execution is profiled, the later will predictions be made and, consequently, the lower will be the potential benefit of the predictions. They have shown that prediction delay is a significant factor in evaluating the quality of a prediction scheme. Thus, while intuition may call for longer and more elaborate profiling, the opposite is true: less profiling actually leads to more effective predictions. We believe it would be interesting to combine hot subpath profiling with their results.

Taub, Shechter and Smith present an idea for reducing profiling overhead [?]. This approach produces binaries that periodically record aspects of their executions in great detail. It works because program behavior is predictable, and it suffices to collect information during only part of the program run-time. After a specified number of executions, the instrumentation can remove itself from the program code, and generates no more overhead.

In [?], Arnold and Ryder proposed to maintain two versions of the program in memory — one instrumented, and one almost uninstrumented. The program execution can then jump between these two versions, collecting enough data for effective profiling, but keeping the overhead low. The technique as presented there is different from the OSP algorithm in several details — back-edges return to the uninstrumented code, independently of the profiler — but their framework could be adapted for use by the OSP algorithm.

Bala, Duesterwald and Banerjia present in [?] a dynamic optimization system called Dynamo. Dynamo is implemented as a native code interpreter that runs on top of the native processor. Once hot traces are located they are aggressively optimized, and the next occurrences of those traces will run natively. Hot traces may begin only at certain predetermined points, so the results obtained by the OSP algorithm, where no such restriction exists, are more general in nature (as can be seen in Figure 1.1). It would be interesting to integrate the OSP algorithm into Dynamo, in order to evaluate its benefits and to compare both methods.

A different approach of using sampling for profiling using a combined software and hardware solution is described in [?]. Adaptive sampling techniques have been used in related fields, such as value profiling [?].

CHAPTER 7

Conclusions

In this paper we demonstrated an efficient technique for online subpath profiling, which is based on an adaptive sampling technique.

The OSP algorithm has been implemented as a prototype, and has been successfully tested on several Java programs.

If the profiler is incorporated into the JVM, the skipping process can be incorporated into the JVM as well. As was mentioned, the profiler overhead consists of two parts — the one caused by the skipping process, and the one caused by the sampling process. Once the skipping process is part of the JVM, its overhead could be lowered. For a discussion of possible optimizations when incorporating profiling into a JVM, see [?]. Once the OSP algorithm is fully integrated into a JVM, its output could be used to locate possible candidates for JIT compilation.

REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, 1997.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] E. Berk and C. S. Ananian. JLex – A lexical analyzer generator for Java. Available at http://www.cs.princeton.edu/~appel/modern/java/JLex.

[6] M. Burrows. Efficient and flexible value sampling. In *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[7] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, 2000.

[8] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD*, pages 331–342, 1998.

[9] JGF. The java grande forum benchmark suite. Available at http://www.epcc.ed.ac.uk/javagrande.

[10] J. R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–269, 1999.

[11] D. Melski and T. W. Reps. Interprocedural path profiling. In *International Conference on Compiler Construction*, pages 47–62, 1999.

[12] S. Sastry, R. Bodik, and J. Smith. Rapid profiling via stratified sampling. In *the 28th International Symposium on Computer Architecture*, July 2001.

[13] E. Segal, Y. Matias, and P. B. Gibbons. Online iceberg queries. Technical report, Tel-Aviv University, 2000.

[14] SPEC. The SPECjvm benchmark suite. Available at http://www.spec.org/osg/jvm98.

[15] O. Taub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000.

[16] R. Vallee-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34, 2000.

[17] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.