Online Subpath Profiling

David Oren, Yossi Matias*, and Mooly Sagiv**

School of Computer Science, Tel-Aviv University, Israel
{doren,matias,msagiv}@post.tau.ac.il

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 exhibited a low average runtime overhead.

1 Introduction

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

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 [7]; and (ii) profiling information is recorded for subpaths that start at arbitrary program points. Related works are described in Section 4.

* Research supported in part by an Alon Fellowship and by the Israel Science Foundation founded by The Academy of Sciences and Humanities

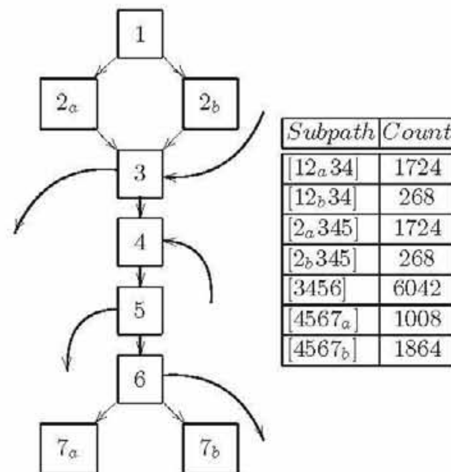
** Research supported in part by the Israel Science Foundation founded by The Academy of Science and Humanities

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 78–94, 2002.
© Springer-Verlag Berlin Heidelberg 2002

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 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.



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

Fig. 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

1.2 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 exhibits high accuracy in reports. For example, consider the `JLex` [5] 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 3.

The OSP algorithm is based on an adaptive sampling technique presented by Gibbons and Matias [8] 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 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 [17] framework for program instrumentation. The architecture of the implementation is described in Figure 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 [9], on `JLex` [5] and on Sun's java `javac` compiler [15].

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 most of the hottest subpaths in

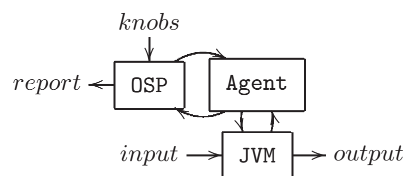


Fig. 2. The OSP Architecture

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 experimental results. Related works are discussed in Section 4. Conclusions and further works are discussed in Section 5.

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 [8]. 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 require 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 3. 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 from 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 sampling probability determines the selection of `skip` in the `chooseSkipValue` method. The `updateHotList` method is responsible for maintaining the hot-list.

```
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;
    }
}
```

```
|    (length)    | (skip) |    (length)    |  (skip)  |
| sampled blocks |        | sampled blocks |          |
```

Fig. 3. The basic OSP algorithm

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.

Let us consider an example of the algorithm in action on the fragment of the control flow graph shown in Figure 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 into 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

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 [8]. 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. A more detailed analysis is given in [12,13].

2.3 Special Considerations

Sampling and Skipping The sampling and counting are performed using a hot-list algorithm [8]. 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

$$f = \frac{n}{m \times G} \quad (1)$$

Instead of deciding for each subpath whether it should be sampled or not, a *skip value* is computed [18]. 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.

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.

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 [10] 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.

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.

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 $P_2 = b_j \cdots b_n b_1 \cdots b_{j-1}$.

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 [8].

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.

3 Prototype Implementation

We have implemented a prototype in Java, using the Soot framework [17].

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 to 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 profiler 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.

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 is 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 description 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 efficiently check whether a subpath is already part of the sampled set, increase the count of an existing subpath, and add a new subpath. Using tries will require a way to convert paths to a canonical form, to make sure the trie is not sensitive to rotation. More details can be found in [12,13].

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. 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.

The implications of this encoding and other possible encodings are presented in [12,13].

3.1 Results

We have run the profiler on four programs from the Java Grande benchmark suite [9], on the `JLex` utility [5] and on the `javac` Java compiler [15]. 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 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).

Table 1. For each program we show the number of basic blocks encountered during execution, the number of subpaths of length 2^n where $2 \leq n \leq 5$ 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

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
javac	9,838,697	49,191,773	462,813

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`.

Runtime Overhead Table 2 shows the runtime overhead of the profiler. The total runtime overhead ranges from 31% to 286%. The sampling overhead (the overhead generated by the `sampleBlock` method) is much smaller, ranging from 3% 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 Java Grande benchmark suite. These benchmarks check raw performance of the JVM, by measuring how many operations of various kinds are performed per second. 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.

Table 2. 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

Program	Time	Instrumented	Only-sampling	Total Overhead	Sampling Overhead
JLex	0.390	0.640	0.070	64.10%	17.95%
FFT	21.080	27.649	2.123	31.16%	10.07%
HeapSort	1.982	6.238	1.111	214.73%	56.05%
MolDyn	10.064	33.878	1.101	236.63%	10.94%
RayTrace	11.997	46.356	0.450	286.40%	3.75%
javac	1.31	3.63	0.230	177.10%	17.55%

Sampling and Efficiency Tradeoff Table 3 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 1.

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. 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 31% in the final run), and the sampling overhead was 102% (compared to 10%).

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.

Table 3. 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

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
javac	281	263	1.07	32,000

Memory Overhead Table 4 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.

Table 4. Memory usage of the different programs. The instrumented memory does not take into account the memory needed for maintaining the output of the algorithm

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

Accuracy of Results Table 5 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 multiplying its count in the sample by the sampling probability and by the a priori probability of sampling a path of that length. 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%.

Table 5. 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

Sample rank	Exact rank	Sample count	Est. count	Exact count	Error	Length
1	1	27,006	108,024,000	109,051,898	0.94%	4
2	2	6,479	103,664,000	103,782,188	0.11%	16
3	3	12,841	102,728,000	101,713,904	1.00%	8
4	4	39,545	79,090,000	79,691,780	0.76%	2
5	6	2,372	18,976,000	14,679,016	29.27%	8
6	11	4,322	8,644,000	8,388,604	3.04%	2
7	12	4,226	8,452,000	8,388,520	0.76%	2
8	10	4,200	8,400,000	8,388,608	0.14%	2
9	9	4,155	8,310,000	8,388,608	0.94%	2
10	8	4,022	8,044,000	8,388,608	4.11%	2

Table 6. 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

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

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

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 6 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`.

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.

4 Related Work

The original Ball-Larus path profiling algorithm recorded the execution frequency of intraprocedural, acyclic paths [4]. 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 [1]. They associated hardware metrics other than execution frequency with paths. They also introduced a runtime data structure to approximate interprocedural paths. In practice [10] 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 [11]. Paths in this technique do not cross loops. Interprocedural paths are assigned a unique identifier statically.

Larus [10] 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 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 [7] 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 [16]. This approach produces binaries that can to 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 generate no more overhead.

In [2], 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 [3] 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). 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 [14]. Adaptive sampling techniques have been used in related fields, such as value profiling [6].

5 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 [2]. Once the OSP algorithm is fully integrated into a JVM, its output could be used to locate possible candidates for JIT compilation.

It is possible to modify the profiler so that it will take the context of subpaths into account. For example, `enterBlock` can be modified to prefer paths starting at a back edge, or any other paths interesting to the user.

One of the main advantages of the OSP algorithm over other methods is that it can cross loop and procedure boundaries. The Ball-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 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.

The algorithm can also be extended to give a priori costs to paths, and to use this costs to affect the probability of sampling paths. For a more in-depth description see [12,13].

Acknowledgments

We 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.

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. 91
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. 92
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. 92
4. T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996. 91
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. 79, 80, 87
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. 92
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. 78, 91
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. 80, 81, 83, 85
9. JGF. The java grande forum benchmark suite. Available at http://www.epcc.ed.ac.uk/javagrande. 80, 87

10. J. R. Larus. Whole program paths. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–269, 1999. 84, 91
11. D. Melski and T. W. Reps. Interprocedural path profiling. In *International Conference on Compiler Construction*, pages 47–62, 1999. 91
12. D. Oren. Online subpath profiling. Master's thesis, Tel-Aviv University, 2002. 83, 86, 87, 93
13. D. Oren, Y. Matias, and M. Sagiv. Online subpath profiling. Technical report, Tel Aviv University, 2002. 83, 86, 87, 93
14. S. Sastry, R. Bodik, and J. Smith. Rapid profiling via stratified sampling. In *the 28th International Symposium on Computer Architecture*, July 2001. 92
15. Sun. The Java2 Platform Standard Edition. Available at http://java.sun.com/j2se/1.3. 80, 87
16. O. Taub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000. 92
17. 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. 80, 85
18. J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985. 83