Evaluating Sampling Based Hotspot Detection[⋆]

Qiang Wu and Oskar Mencer

Department of Computing, Imperial College London,
South Kensington, London SW7 2AZ, UK
`{qiangwu,oskar}@doc.ic.ac.uk`
`http://comparch.doc.ic.ac.uk`

Abstract. In sampling based hotspot detection, performance engineers sample the running program periodically and record the Instruction Pointer (IP) addresses at the sampling. Empirically, frequently sampled IP addresses are regarded as the hotspot of the program. The question of how well the sampled hotspot IP addresses match the real hotspot of the program is seldom studied by the researchers. In this paper, we use instrumentation tool to count how many times the sampled hotspot IP addresses are executed, and compare the real execution result with the sampled one to see how well they match. We define the normalized root mean square error, the sample coverage and the order deviation to evaluate the difference between the real execution and the sampled results. Experiment on the SPEC CPU 2006 benchmarks with various sampling periods is performed to verify the proposed evaluation measurements. Intuitively, the sampling accuracy decreases with the increase of sampling period. The experimental results reveal that the order deviation reflects the intuitive relation between the sampling accuracy and the sampling period better than the normalized root mean square error and the sample coverage.

Key words: hotspot detection, sampling, accuracy, performance event counters, instrumentation

1 Introduction

Sampling based hotspot detection is a common practice to locate the frequently executed part of a program in performance analysis. With the help of hardware performance event counters built in the processors, sampling can be done efficiently with a low overhead. Most performance monitor tools provide the functionality to count the Instruction Pointer (IP) addresses encountered during the sampling, revealing a runtime profile of the program [1][2][3][4]. By analyzing the collected counts of IP addresses, performance engineers can figure out which part in the program is most frequently executed, in a statistical manner. Intuitively, the more the IP address is encountered in the sampling, the likelier the IP address is a hotspot of the program.

[⋆] This work is supported by EPSRC grant - Liquid Circuits: Automated Dynamic Hardware Acceleration of Compute-Intensive Applications

However, the periodical sampling may not match with the real execution of the program due to its statistical nature. Fig. 1 shows the comparison of the sampled counts and the real execution counts of IP addresses for the SPEC CPU 2006 benchmark 403.gcc with one of the input data sets. For legibility, the counts are displayed in their percentages against the total sample number and the total instruction number respectively. From the figure, we can see that the sampled counts are different from the real execution counts of the IP addresses. Basically, the most frequently sampled IP address is not the same as nor even close to the most frequently executed IP address.

So the question arises that how well the sampled hotspot IP addresses match the real hotspot of the program. Since the hotspot detection is the starting step of the performance engineering, we don't want to be diverted too far from the real hotspot at the beginning.

In this paper, we address the issue of sampling accuracy by proposing evaluation method to compare the sampled hotspot IP addresses with the real execution hotspot of the program. Main contributions of our work are outlined as follows:

1. Three measurements, normalized root mean square error, sample coverage and order deviation, are proposed to evaluate the accuracy of sampling based hotspot detection;
2. Experiment on SPEC CPU 2006 benchmarks with various sampling periods is performed to verify the proposed evaluation measurements;
3. Based on the intuitive relation between the sampling accuracy and the sampling period, order deviation is regarded as the most appropriate measurement to evaluate sampling accuracy.

2 Related Work

Sampling based hotspot detection with hardware performance event counters is widely used in performance analysis. [5] introduces the method of using performance event counters to analyze the performance of running programs. In addition, [5] investigates the accuracy of event counting with multiplexing of performance event counters, which is also studied in [6]. They compare the event counts with those estimated from the incomplete samples with performance event counter multiplexing, to figure out how much error is introduced by the multiplexing technique.

Another accuracy issue is indicated by Korn W., Teller P.J. and Castillo G. in [7], which discusses the error introduced by the overhead of the event counting module itself. They compare the event counts collected by the performance event counters with the predicted ones. [8] and [9] look into this issue further. Both counting and sampling modes of the performance event counters are tested, and the measured counts are compared with the predicted ones. Statistical error in sampling mode is mentioned in [8], with a brief description of the difference between the counts obtained from sampling results and the real execution counts.

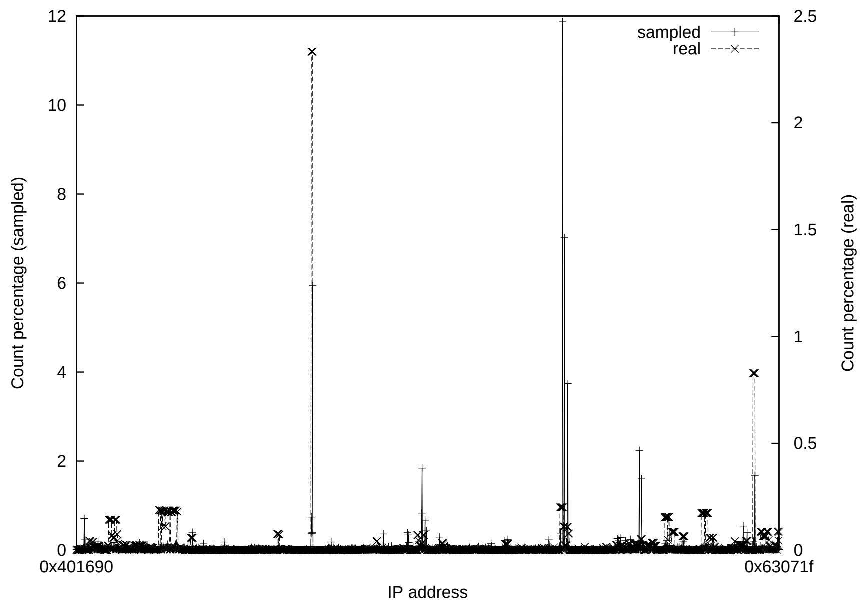


Fig. 1. Count percentages of 403.gcc with input set 1 sampled at 16M unhalted clock cycles (percentages are calculated by dividing the sampled and real execution counts with the total sample number and total executed instruction number respectively)

No further investigation of the issue is made. In [10] and [11] simulation accuracy is studied. [10] uses Euclidean distance to compare the difference of frequency vectors and calculates the error with the distance between simulated results and real execution, and the variance with the average squared distances. In [11] the accuracy of some cumulative property estimated from samples, such as CPI (Cycles Per Instruction), is evaluated based on the statistical mathematics.

3 Method

We notice that processors with built-in hardware event counters always support the CPU clock cycles (unhalted clock cycles) event. The unhalted clock cycles event is often used for determining the sampling period in sampling based hotspot detection. In the information recorded during the sampling, IP address is usually available. Taking the above observations into account, we focus on the sampling period in terms of unhalted clock cycles in this paper. And we assume the IP address is recorded during the sampling. With as few as possible assumptions, our method is generic for different architectures.

3.1 Definitions

The sampling procedure records IP address with the sampling period T, resulting in a list of paired values (t_1, IP_1), (t_2, IP_2), ..., (t_n, IP_n), where n is the number of samples obtained during the sampling. We use t_1, t_2, ..., t_n instead of T, $2T$, ..., nT for two reasons. One is that the actual time of sampling is hardly T, $2T$, ..., nT due to the uncertainty in the handling of sampling interrupts. The other is that randomization of the sampling period is often used in sampling to avoid biased sampling result.

Aggregate the sampling records (t_1, IP_1), (t_2, IP_2), ..., (t_n, IP_n) to IP addresses, we can get a count histogram of IP addresses, $S = \{(IP_1, c_1), (IP_2, c_2), ..., (IP_m, c_m)\}$, where c_i indicates the count of IP_i in the sampling, and m is the number of different IP addresses collected in the sampling.

On the other hand, we can use instrumentation tool to get the execution count of all the basic blocks of the program with the specific input set. Suppose $B = (BBL_1, bc_1), (BBL_2, bc_2), ..., (BBL_l, bc_l)$ be the counts of all basic blocks of the program with the specific input set, where bc_i is the execution count of the basic block BBL_i, and l is the number of different basic blocks. With the above basic block counts, we can find out the real execution counts of the IP addresses collected in the sampling by simply looking up the corresponding execution count of the basic block that the IP address falls in. Let $R = \{(IP_1, r_1), (IP_2, r_2), ..., (IP_m, r_m)\}$ denote the real execution counts of all the sampled IP addresses.

To compare the sampled hotspot with real hotspot of the program, we need some measurements to evaluate the difference between the two sets of values, S and R with the additional basic block counts B.

3.2 Measurements

Normalized Root Mean Square Error To evaluate the difference between the predicted and real values, the root mean square error is often employed [12]. We normalize it with the range of the values involved to balance among difference tests:

$$NRMSE = \frac{\sqrt{\sum_{i=1}^{m} \frac{1}{m}(x_{1,i} - x_{2,i})^2}}{x_{max} - x_{min}}$$

where $\{x_{1,i}\}$ and $\{x_{2,i}\}$ are two sets of values.

Considering the particulars of the sampled counts and the execution counts of IP addresses, we make two modifications to the above formula. The first is the scaling of the sampled counts of IP addresses. Since the sampled count is always far less than the execution count of an instruction at the specific IP address, directly subtracting the sample count with the execution count leads to a difference too large to evaluate. In particular, this large difference makes the variations among different sampling periods hidden behind the big numbers. So, instead of subtracting the count values directly, we calculate the difference between the count fractions. That is, we place ($\frac{c_i}{NS}$ - $\frac{r_i}{NI}$) instead of (c_i - r_i)

inside the NRMSE formula, where NS is the number of all samples and NI is the number of all instructions executed.

The second is the weighting of the count differences. In normal NRMSE formula, $\frac{1}{m}$ is used for all squared differences, which means the same weight for all differences. However, in hotspot detection, the larger the count is, the more important it is. So we replace the $\frac{1}{m}$ with the $\frac{c_i}{NS}$, where $\sum_{i=1}^{m} \frac{c_i}{NS} = 1$.

The resultant normalized root mean square error formula to compare sample counts S and real execution counts R is:

$$NRMSE_{SR} = \frac{\sqrt{\sum_{i=1}^{m} \frac{c_i}{NS}(\frac{c_i}{NS} - \frac{r_i}{NI})^2}}{\max_{i=1}^{m}(\{\frac{c_i}{NS}\} \cup \{\frac{r_i}{NI}\}) - \min_{i=1}^{m}(\{\frac{c_i}{NS}\} \cup \{\frac{r_i}{NI}\})}$$

In the above formula, c_i and r_i are the sample count and real execution count for IP address IP_i respectively. NS is the number of all samples which is equal to $\sum_{i=1}^{m} c_i$. NI is the number of all executed instructions.

Sample Coverage It is a well-known rule of thumb that 80% time of the execution is spent on 20% of the program code. This sparks the measurement of the sample coverage to evaluate the sampling based hotspot detection. In this paper, the sample coverage is simply defined to be the total real execution count of sampled IP addresses over the number of all the instructions executed.

$$SC = \frac{\sum_{i=1}^{m} r_i}{NI}$$

Since r_i represents the real execution count of the instruction at the address IP_i, the SC indicates the portion of the instructions at sampled IP addresses in the whole execution of the program.

Order Deviation In hotspot detection, we care more about the order of the counts of IP addresses than their actual values. It is a common practice in hotspot detection to pick the top IP addresses from the list sorted by the sampled count. The picked IP addresses, or the IP addresses with largest sampled count, may correspond to the most frequently executed instructions in the program, or may not, as we have seen in Fig. 1.

The difference between the order of IP addresses in the sampled count list and the order of these IP addresses in the real execution count list is an interesting issue to investigate. We propose the order deviation to measure the difference between the orders of the IP addresses in the sampled count and real execution count lists. Since we utilize the basic block counts to sort out the real execution count list, the basic block count is used to represent the real execution count in the following text.

Before computing the order deviation, it should be noted that there may be several IP addresses having the same count value. For these IP addresses, we assume them having the same order level. This means that the IP addresses with the same count value have no difference in order inside the involved list. The following is an example of the sorted list of IP addresses and sampled counts illustrating the meaning of the order level.

$$\underbrace{\overbrace{(IP_{i_1}, c_{i_1})(IP_{i_2}, c_{i_2})...}^{\text{order level 1}}}_{c_{i_1} = c_{i_2} = ... = v_1}, ..., \underbrace{\overbrace{(IP_{i_j}, c_{i_j})(IP_{i_{j+1}}, c_{i_{j+1}})...}^{\text{order level } j}}_{c_{i_j} = c_{i_{j+1}} = ... = v_j}, ...$$

Here v_1, ..., v_j, ... represent different values in the counts c_1, c_2, ..., c_m.

Suppose the $SOLevel(IP_i)$ returns the order level of IP_i in the list of IP addresses sorted by sampled count. $ROLevel(IP_i)$ returns the order level of the basic block where IP_i locates, in the list of basic blocks sorted by the execution count of the basic blocks. The order deviation is defined as:

$$OD = \frac{\sqrt{\sum_{i=1}^{m} \frac{c_i}{NS}(SOLevel(IP_i) - ROLevel(IP_i))^2}}{m}$$

Here m is the number of different IP addresses, c_i is the sampled count for IP_i, NS is the number of all samples equal to $\sum_{i=1}^{m} c_i$. The OD formula gives the weighted root mean square error of the order level per IP address.

3.3 Tool

We use pfmon in [4] as the sampling tool. pfmon provides the thread-specific sampling and the feature of the randomization of the sampling period. By default, pfmon supports the unhalted clock cycles event for different processors. It comes with a sampling module that can record the IP address during sampling and print out the count histogram of the IP addresses.

For the instrumentation, we employ the Pin [14] tool. Pin is able to instrument executable files and even the running program on the fly. It supports image level, routine level, basic block level and instruction level instrumentation. We utilize the basic block level instrumentation to record the execution counts of basic blocks.

4 Experiment

We perform the experiment on SPEC CPU 2006 [13] benchmarks with various sampling periods. All benchmarks with different input sets in SPEC CPU 2006 have been tested, totally 55 test cases.

4.1 Test Platform

The test platform is a workstation with 2 AMD opteron dual core processors running at 2210 MHz. The size of the system RAM is 2 GB.

SPEC CPU 2006 version 1.0 benchmarks are installed on the system, and are built by GCC and GFortran 4.1.2 with -O2 switch.

The operating system is Ubuntu Linux 7.10 with kernel 2.6.24.3, running in 64-bit mode. The kernel is patched with perfmon [4] to interface the performance event counters. The version of perfmon kernel patch is 2.8 which is required for pfmon [4] tool version 3.3. pfmon is used to perform the sampling and print out the counts histogram of IP addresses.

Instrumentation is done with the Pin [14] tool, which figures out the execution counts of basic blocks. The version of Pin is 2.4 for Linux x86_64 architecture.

4.2 Test Results

We use pfmon to sample the SPEC CPU 2006 benchmarks and get the histograms of the sampled counts of IP addresses. Sampling periods are set to be 9 different values: 64K, 128K, 256K, 512K, 1M, 2M, 4M, 8M and 16M unhalted clock cycles. To avoid biased sampling results, sampling period randomization provided by pfmon is used for all the tests. The randomization mask value is set to be one eighth of the sampling period. That is to say, at the time of sampling, a random value up to one eighth of the sampling period is added to the original value. The resultant value is used as the count-down value for the next sampling.

We have carried out the tests on three evaluation measurements: normalized root mean square error (NRMSE), sample coverage (SC) and order deviation (OD). An intuitive rule is adopted to evaluate the above measurements:

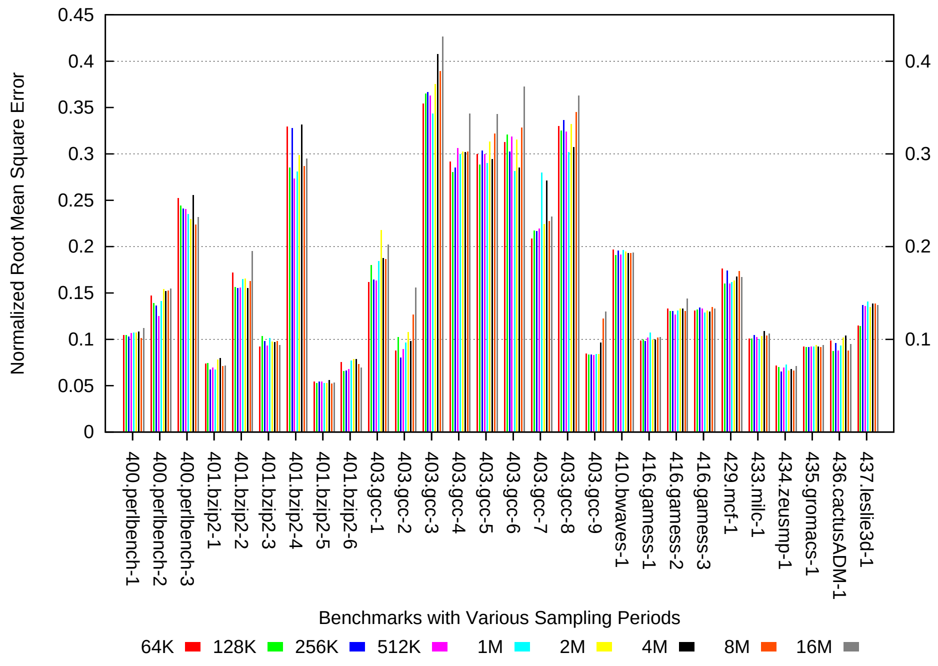
– The longer the sampling period is, the less accurate the sampling result is.

This intuitive rule means that the difference between the sampling result and the real execution, in our case, the normalized root mean square error and order deviation, should rise with the increase of the sampling period. On the contrary, the sample coverage should fall with the increase of the sampling period. The test results are shown and discussed in below.

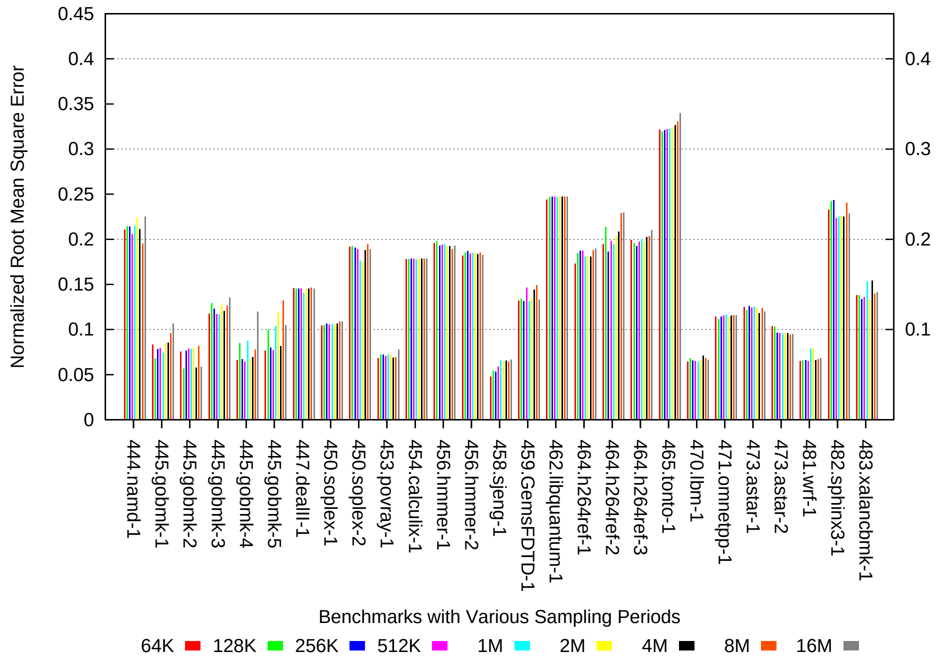
Normalized Root Mean Square Error (NRMSE) Fig. 2 shows the NRMSE values for SPEC CPU 2006 benchmarks with various sampling periods. NRMSE values are displayed with one clustered bars for each benchmark, and one color for one sampling period. Bars are lined from left to right in the ascending order of sampling periods, from 64K to 16M unhalted clock cycles.

It can be seen from the figure that most benchmarks have a normalized root mean square error (NRMSE) value around or below 0.2. NRMSE values of benchmarks 401.bzip2 with input set 4, 403.gcc with input sets 3, 4, 5, 6 and 8 and 465.tonto with input set 1 are around or above 0.3.

The more noticeable feature of the figure is the variation pattern of each cluster of bars, which represents the NRMSE values corresponding to different sampling periods of each benchmark. We expect to see a rising trend of the bars with the increase of the sampling periods. However, it should be admitted that in Fig. 2, there is no obvious trend of ascending witnessed in the clusters of bars.



(a) SPEC CPU 2006 benchmarks 400 to 437



(b) SPEC CPU 2006 benchmarks 444 to 483

Fig. 2. Normalized Root Mean Square Error (NRMSE) of SPEC CPU 2006 benchmarks with various sampling periods (NRMSE values of each benchmark are displayed from left to right with a cluster of bars in the order of sampling periods from 64K to 16M unhalted clock cycles)

Sample Coverage (SC) Fig. 3 shows the sample coverage values for SPEC CPU 2006 benchmarks with various sampling periods. Sample coverage values are displayed in a similar way as Fig. 2, with one color bar for one sampling period and a cluster of bars for one benchmark.

We can see in Fig. 3 that most benchmarks have a sample coverage (SC) value above or around 0.5. Benchmark 403.gcc with input sets 1, 2, 3 and 9, 410.bwaves with input set 1 and 465.tonto with input set 1 fall below 0.5.

For the variation inside each bar cluster, we can find a trend of descending in most of the clusters, with some exceptions such as 401.bwaves, 416.gamess with input set 3, 447.dealII, 454.calculix, 456.hmmer, and 473.astar. The descending trend means that with the increase of the sampling period, the sample coverage is decreasing, i.e. the IP addresses recorded in the sampling covers less instructions in the program execution.

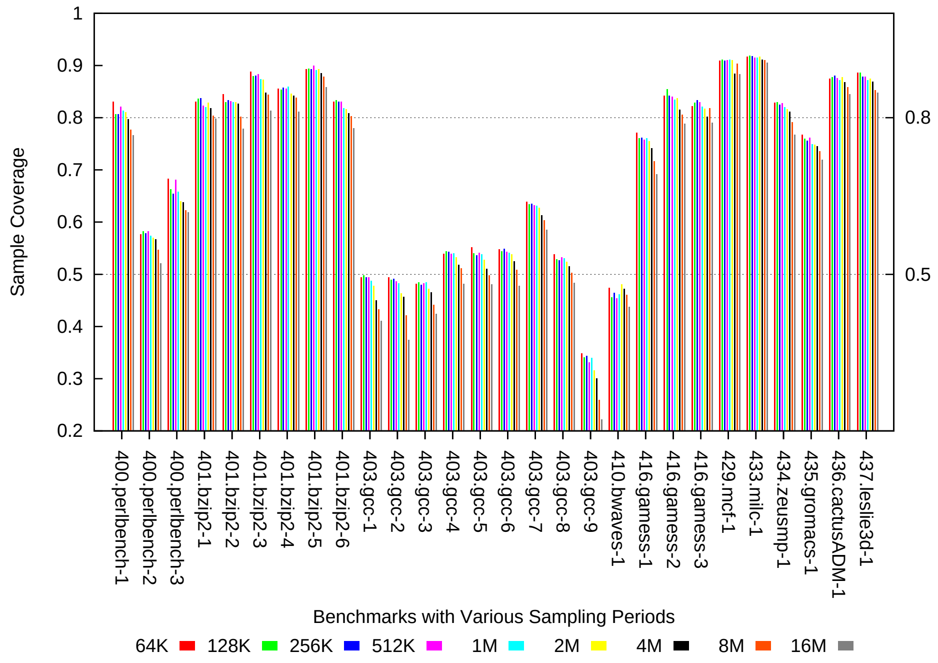
Order Deviation (OD) Fig. 4 shows the order deviation (OD) values for SPEC CPU 2006 benchmarks with various sampling periods. Each cluster of color bars corresponds to a set of OD values for one benchmark with different sampling periods. It should be noted that the order deviation values are displayed in logarithmic scale for a better visual effect.

In the Fig. 4, despite the actual order deviation values, the most interesting discovery is that nearly all the bar clusters show a trend of ascending with the increase of the sampling period. The only exception is the bar cluster corresponding to 470.lbm with input set 1, which has a descending trend. As we mentioned before, intuitively the accuracy of sampling should fall with the increase of the sampling period. Obviously, the order deviation reflects this intuitive rule better than the sample coverage and the normalized root mean square error.

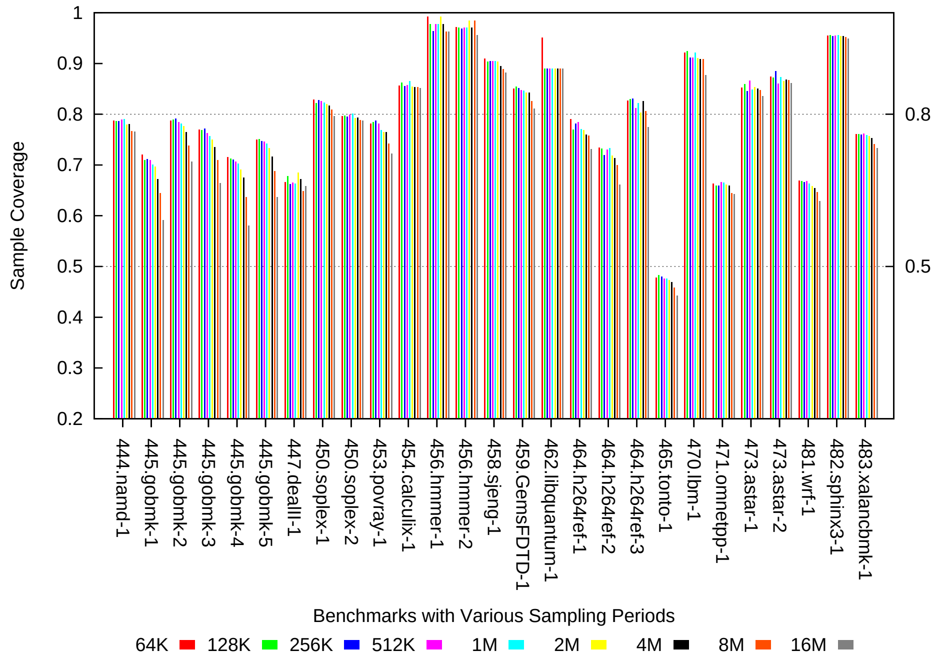
For the actual values, we can see that order deviations (OD) for most benchmarks are around or below 1.0. The outstanding ones are the OD values of benchmark 403.gcc with input sets 1 to 9, which are above 2.0. Roughly speaking, an OD value of 2.0 means that in average the order of each sampled IP address is deviated by 2 from the real execution order of the IP address in the whole program. We regard that the order deviation above 2 is not good for identifying hotspots. Considering that benchmark 403.gcc is a control dominant program, which has lots of branches in its code, this suggests that the sampling based hotspot detection has an accuracy degradation for the control dominant programs.

5 Conclusion

In this paper, we investigate the accuracy of sampling based hotspot detection. Three measurements, normalized root mean square error, sample coverage and order deviation are proposed to evaluate how well the sampled hotspot matches the real hotspot of the program. These measurements are adopted in the consideration of the experiences of previous research work and our observations. We test the proposed measurements on SPEC CPU 2006 benchmarks with different

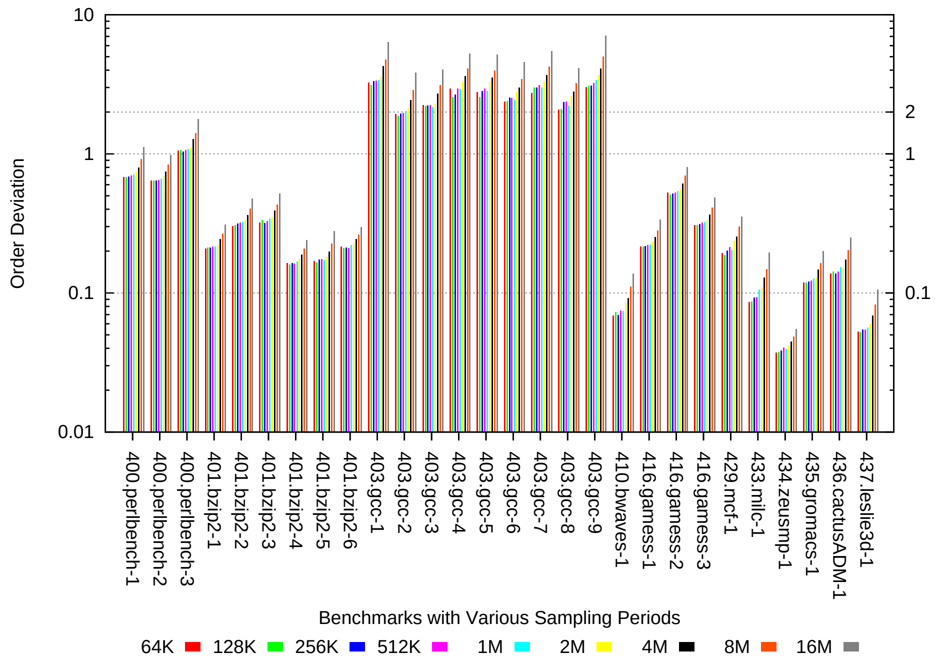


(a) SPEC CPU 2006 benchmarks 400 to 437

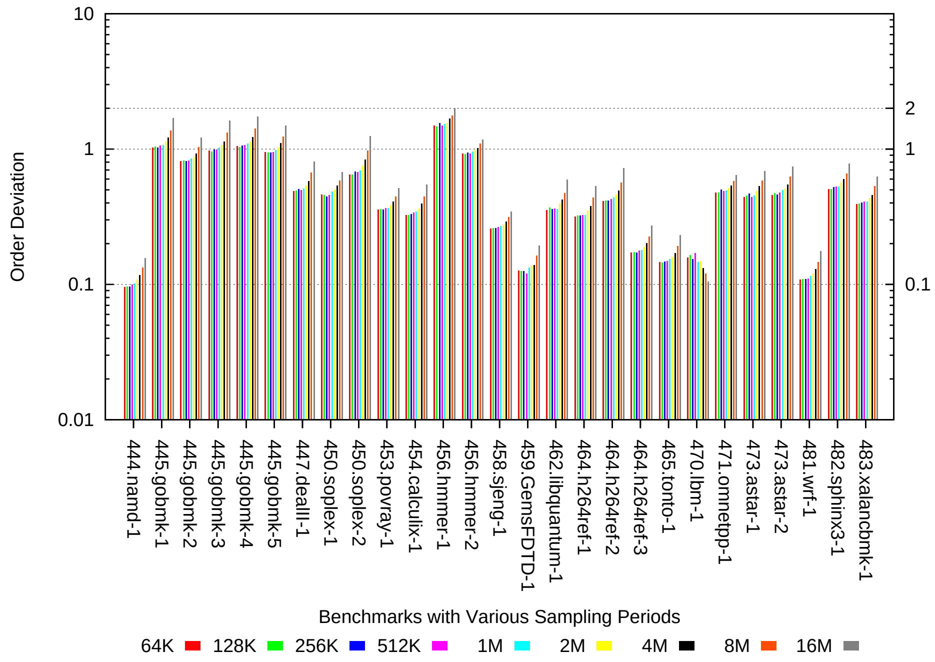


(b) SPEC CPU 2006 benchmarks 444 to 483

Fig. 3. Sample coverage (SC) values of SPEC CPU 2006 benchmarks with various sampling periods (SC values of each benchmark are displayed from left to right with a cluster of bars in the ascending order of sampling periods from 64K to 16M unhalted clock cycles)



(a) SPEC CPU 2006 benchmarks 400 to 437



(b) SPEC CPU 2006 benchmarks 444 to 483

Fig. 4. Order Deviation (OD) of SPEC CPU 2006 benchmarks with various sampling periods in logscale (OD values of each benchmark are displayed from left to right with clustered bars in the order of sampling periods from 64K to 16M unhalted clock cycles)

sampling periods. To verify and compare the proposed measurements, we exploit an intuitive relation between the sampling accuracy and sampling period. The longer the sampling period is, the less accurate the sampling result is. From the experimental results, we find that the order deviation fits the intuitive relation best, with only 1 significant exception out of 55 test cases. We also notice that the control dominant benchmark 403.gcc has relatively lower sampling accuracy indicated by the order deviation measurement which are generally above 2.0, a level of value that is regarded as acceptably different in our point of view. This suggests that control dominant programs usually degrade the sampling accuracy to some extent. Our future plan is to investigate the relation between the program code characteristics and the sampling accuracy.

References

1. Intel Corporation. Intel VTune Performance Analyzer. `http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm`
2. AMD Inc. AMD CodeAnalyst Performance Analyzer. `http://developer.amd.com/CPU/Pages/default.aspx`
3. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net`
4. perfmon project, `http://perfmon2.sourceforge.net`
5. Azimi, R., Stumm, M., and Wisniewski, R. W.: Online performance analysis by statistical sampling of microprocessor performance counters. In: Proc. of the Intl. Conf. on Supercomputing, pp. 101–110. ACM Press, New York (2005)
6. W. Mathur, J. Cook.: Towards Accurate Performance Evaluation using Hardware Counters. In: Proc. of the ITEA Modeling and Simulation Workshop. Las Cruces, NM. (2003)
7. Korn, W., Teller, P.J., Castillo, G.: Just How Accurate Are Performance Counters?. In: Proc. of the Intl. Conf. on Performance, Computing, and Communications, pp. 303–310. IEEE Press, New York (2001)
8. Michael E. Maxwell, Patricia J. Teller, Leonardo Salayandia, Shirley V. Moore: Accuracy of Performance Monitoring Hardware. In: Proc. of the Los Alamos Computer Science Institute Symposium, Santa Fe, NM (2002)
9. Shirley V. Moore: A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. In: Proc. of the Intl. Conf. on Computational Science, pp. 904–912. Springer, Berlin/Heidelberg (2002)
10. Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder: SimPoint 3.0: Faster and More Flexible Program Analysis. Journal of Instruction-Level Parallelism 7 (2005) 1-28
11. R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe: SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In: Proc. of ISCA, pp. 84–95. IEEE Press, New York (2003)
12. Husam Hamad.: A new metric for measuring metamodels quality-of-fit for deterministic simulations. In: Proc. of the Conf. on Winter Simulation, pp. 882–888. ACM Press, New York (2006)
13. Standard Performance Evaluation Corporation, `http://www.spec.org`
14. Pin - A Dynamic Binary Instrumentation Tool, `http://rogue.colorado.edu/pin`