IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 20, 2019, accepted July 25, 2019, date of publication August 20, 2019, date of current version August 30, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936439

A Gaussian Set Sampling Model for Efficient Shared Cache Profiling on Multi-Cores

YI ZHANG[1], ZHANWEI LING[2], MINGSONG LV[2], AND NAN GUAN[3]
[1]Department of Sino-Dutch Biomedical and Information Engineering School, Northeastern University, Shenyang 110169, China
[2]Department of Computer Science and Engineering School, Northeastern University, Shenyang 110169, China
[3]Department of Computing, The Hong Kong Polytechnic University, Hong Kong

Corresponding author: Yi Zhang (zhangyi@bmie.neu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61602104, Grant 61772123, and Grant 61701099, and in part by the Ministry of Education Joint Foundation for Equipment Pre-Research under Grant 6141A020333.

ABSTRACT The last level cache (LLC) has significant impact to system performance on modern multi-core processors. But as cache sizes reach several megabytes and more, the overhead of exploring performance on LLC greatly increases as well. To improve the efficiency of performance analysis, we propose a set-sampling-based cache profiling model for the performance analysis on multi-core LLC. We first explore the memory access distributions on LLC by developing a low-overhead stress-application-based method. The results show that memory access distributions can be approximated by Gaussian distribution function. Based on this observation, a Gaussian-distribution-based set sampling model is proposed which can predict program performance with limited representative samples. We evaluate our model on a contemporary multi-core machine and show that 1) the proposed method can precisely predict program performance on LLC under different contention intensities and 2) our method can achieve similar precision with less samples compared to widely adopted set sampling methods such as the random sampling and the continuous address sampling.

INDEX TERMS Gaussian distribution, multi-core, shared cache, set sampling.

I. INTRODUCTION

Modern multi-core processors rely on large on-chip caches to reconcile the gap between core and memory speeds. The software performance strongly depends on how well the capability of cache is utilized. Therefore, understanding the cache access behavior is crucial for predicting and optimizing the performance of multi-core software. Typically, different cores share an on-chip multi-megabyte last level cache (LLC). On the shared LLC, accesses from one core suffers interferences from other cores, which makes the modeling and analysis of LLC behavior a challenging problem.

There are different types of techniques to analyze LLC behavior, such as analytical model [1], [2], trace-driven simulation [3]–[5] and profiling-based performance prediction [6]–[8]. Analytical models typically provide insights on program behavior for performance prediction. However, most models oversimplify the factors to facilitate analysis [9]. Simulation-based methods have the advantages of flexibility, but suffer from long simulation time. Profiling-based approaches directly obtain real performance information from system runs. However, profiling-based approach still needs to address efficiency problem. In cache performance analysis, since LLC is typically very large, analysis efficiency is low. It is desirable to design efficient cache profiling techniques, especially for iterative design space exploration of high performance and embedded systems, which need to analyze the system behavior repeatedly and adjust system design to achieve the design goal.

Set sampling can improve the cache profiling efficiency [10], [11] but at the cost of degraded accuracy. Existing works [11] mainly performs random set sampling, and to achieve satisfiable accuracy, a considerable number of cache sets has to be selected, and thus the corresponding analysis efficiency is still low [12], [13].

In this paper, we propose a novel set sampling technique for efficient and accurate performance prediction. First, we obtain the cache access behavior of a target program by simultaneously running a specially designed stress program with the target program so that they compete for cache space at runtime. From the cache miss counts of the stress program, we can deduce access distribution or other performance metrics of the target program. The key insight is that a program's cache access distribution on different cache sets can be well fit into Gaussian distribution. Based on this observation, we developed an efficient set sampling technique. The main

The associate editor coordinating the review of this article and approving it for publication was Stavros Souravlas.

idea is that very few key sampling points can represent a Gaussian distribution function, and thus it suffices to use only a small number of cache sets (corresponding the key sampling points) to represent the cache behavior of the whole program. Experimental results show that our proposed method only samples a much smaller number of cache sets to achieve the same accuracy as by the state-of-the-art random sampling methods.

II. RELATED WORKS

There has been a lot of work on investigating the system performance on shared cache. One kind of this technique is developed with simulation. The methods based on the full system simulation [14], [15] can provide the most detailed investigation, but are with the most expensive overhead. Trace driven methods [5], [16]–[18], which only simulate/model parts of the system, can provide faster, but less detailed approaches. The major intrinsic problem for simulation technique is that the accurate simulation normally runs for a very long time, which leads to poor scaling.

Another kind of technique is the profiling-based method which is developed on cache usage monitoring on real hardware [7], [8], [19]–[21]. This kind of technique leverages the existing performance monitoring unit on commodity hardware, which makes it more appropriate to be deployed on real system.

Set sampling is an effective method to reduce the overhead in simulation and profiling. Kessler *et al.* [10] compared set sampling and time sampling in trace-driven simulation and showed that set sampling has better precision than time sampling. Zhao *et al.* [11] proposed CacheScouts which leverages set sampling to implement low-overhead occupancy and interference monitoring of the shared caches. This method has been implemented on Intel commercial multi-core chips [6]. Qureshi *et al.* [3] used a set sampling method to implement Set Dueling which can dynamically switch between two cache replacement policies. A Gaussian-based cache access distribution is assumed, based on which random set sampling is analysed to derive the bounds for Set Dueling. In our work, we demonstrate that cache misses of most program do follow Gaussian distribution, and we proposed a more efficient cache sampling method based on the key feature of Gaussian distribution.

Besides improving the efficiency in simulation and profiling, set sampling can meanwhile reduce the interference of the profiling program to the system. For example, in the profiling methods [7], [8], [22], [23], they typically use all the cache sets. As shown in Eklov's work [7], their profiling method introduces about 5% runtime overhead to the tested application. Since set sampling can only use a small number of cache sets for performance profiling, the interference is comparably small.

III. OBTAINING MEMORY ACCESS DISTRIBUTION

The main objective of this work is to find a small number of cache sets which can accurately represent the overall cache access behavior. To achieve this goal, we need to first understand how cache accesses distribute over the cache sets.

A. OVERVIEW

Generally, it is hard to directly obtain the runtime cache access behavior of a target program. A common solution is to run the target program on a simulator which simulates the hardware and reports cache access details. However, the enormous time consumption and the un-revealed details of the target processor make this solution less appropriate for the real applications.

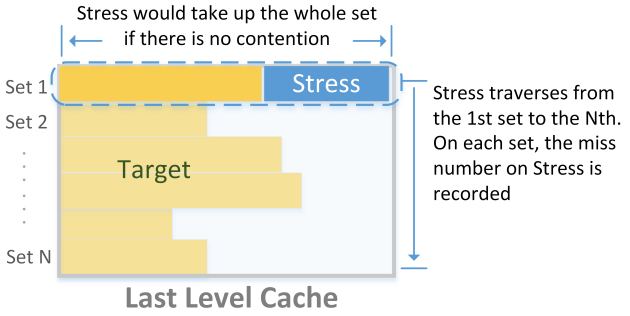


FIGURE 1. Stress program method for obtaining a program's memory access distribution on LLC.

To know how memory accesses of a program distribute on the cache sets of a real processor, we develop a stress-program-based method that allows us to obtain this information by simultaneously running a specially designed stress program with the target program. FIGURE 1 provides an illustration of this method. On each of the cache sets, we let the stress program generate the same kind of access sequence and collect the cache miss number of the stress program through performance counters provided by the processor. The more cache occupancy taken by the target program on a set, the more misses the stress program will receive on this set. As stress program issues the same kind of access sequence to each cache set, the contention between target program and the stress program on the cache sets can be used as a proxy to profile the target program's cache access distribution on the cache.

B. DESIGN AND IMPLEMENTATION

As illustrated in Figure 1, we co-run the target program with stress program and collect the cache miss numbers caused by the contention between target program and stress program to infer the cache access distribution of the target program. Given a cache with N cache sets, we accomplish the cache miss collection by co-running the target program with the stress program for N times. In each running, the target program conducts the same part of the program and the stress program is controlled to only access a distinct cache set. Thus, after performing the same stress over N cache sets, we can obtain the access information on each cache set.

To generate the accesses to fall into the target cache set, we allocate the accessed items with known physical

memory mappings. Thus, by reading a specific item that locates in a specific physical address, we can create an access to a target cache set. We obtain the memory mapping information through/proc/self/pagemap interface on Linux, which provides us with the known physical location of each page. To provide the available memory mapping that can cover all cache sets, we use the huge pages to allocate memory for the stress program. Another advantage of using huge pages is to eliminate cache misses caused by the TLB miss, which further reduces imprecision in the obtained cache miss counts.

Another issue of implementing stress program is to generate the access sequence which is able to accurately profile the misses caused by contention between target and stress programs. On each cache set, we achieve the access sequence by repeatedly reading M distinct memory blocks from the 1st one to the Mth in the same access speed, where M is the number of LLC associativity and the M memory blocks are all located on the target cache set. The main ideas in this design are as follows.

- Since the associativity of LLC is commonly much larger than the associativity of the upper level caches, the iterative accesses on the M blocks won't hit on the upper level caches and all the accesses will fall onto the LLC cache set.
- If on the target cache set there is no access issued from the target program, the accesses of the stress program will hit on these blocks and no miss will occur.
- When accesses from target program arrive on this set, the stress program can not monopolize the cache set and cache misses occur in the stress program when the replaced blocks are accessed. The higher access rate the target program introduces on the target set, the more cache misses will occur on the stress program. The cause for such cache miss behavior lies in the observation that the same techniques, such as recency and protection, are broadly used on LLC replacement policy, although the proposed LLC replacement policies differ in their mechanisms [24].

  Generally, recency technique priorities recently used cache blocks over old ones, and protection technique priorities the hit cache blocks against eviction. If the access sequence is with few reused blocks, then recency will be the main technique that operates. The higher access rate means the accesses are issued more recently and therefore take up more cache space. If the access sequence is with reused blocks, then both recency and protection techniques could work. In this case, the items with higher access rate mean they are not just issued more recently but are also with more hits, and therefore they could occupy more cache space as well. Thus we can conclude that when profiling with our stress program, the cache set receiving higher rate accesses from target program could reserve more cache blocks for the target program and cause more cache misses on the stress program.

As the target programs can have various cache access rates and our stress program could intensively access each cache set during profiling, we also need to adaptively tune the stress program's access rate such that it won't excessively take up the cache space. We tune the cache access rate by adding a specific number of null operations after each access of the stress program. Since the null operation is very trivial and can be kept on the upper level cache, the null operations only take time but don't increase the misses on LLC.

C. EXPERIMENTAL RESULTS

Experiments were conducted on a quad-core Intel 2600 processor running Linux 3.16.7. The processor has 8MB LLC organized into 2048 cache sets. We randomly chose 9 benchmark programs from SPEC CPU 2006 as target programs. The cache access results are shown in Figure 2. In each chart, the x-axis is the cache set ID ranging from 1 to 2048, and the y-axis is the collected cache miss counts of the stress program on each cache set, when it co-runs with each target program. Note that cache misses of stress program can indicate the cache access behavior of the target programs.

From the figures it seems that the target programs behave quite differently from each other regarding the cache miss distribution over cache sets. In our work, we found that most programs exhibit similar probability distribution of cache misses, from which we can select a few cache sets to compactly and precisely represent the cache access behavior of all cache sets, when trying to obtain the cache occupancy of the target program. This will be detailed in the next section.

IV. MODELING DISTRIBUTION

A. PROBABILITY DISTRIBUTION OF CACHE MISSES

To understand the probability distribution of cache misses, we use a bar graph (Figure 3) to represent the probability density over different cache misses. Let N denote the number of cache sets ($N = 2048$ in our experiments); let $miss_k$ denote the miss count of the stress program collected on each cache set, where $k \in \{1, 2, \ldots, N\}$; let $[a, b]$ denote the range of miss counts occurs for all cache sets.

The interval $[a, b]$ is evenly divided into a group of m sub-intervals, so the size of a sub-interval is $(b-a)/m$. In practice, $m \approx 1.87(N-1)^{0.4}$.

For each sub-interval, we count the number of cache sets whose miss counts fall into the sub-interval and denote it as υ_i, where $i \in \{1, 2, \ldots, m\}$. Let f_i denote the frequency of occurrence in each sub-interval, then:

$$f_i = \frac{\upsilon_i}{N} \tag{1}$$

Let the median number x_i denote a sub-interval; let y_i denote the probability density within the interval x_i, then

$$y_i = \frac{f_i}{(\frac{b-a}{m})} \tag{2}$$

The bar graph of probability density distributions is shown in Figure 3. In each graphics, x-axis is the cache miss number and y-axis is the value of probability density.

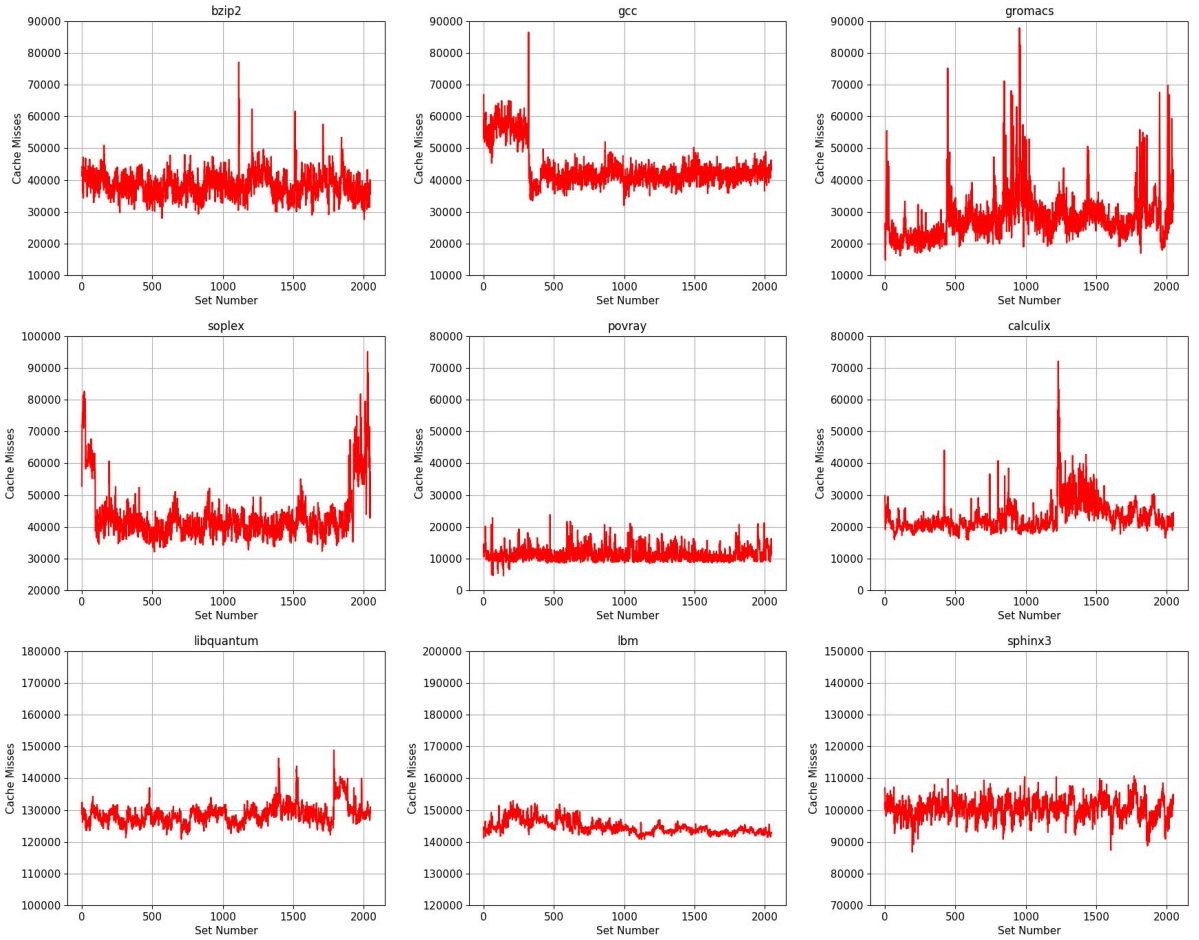


FIGURE 2. Memory access distribution over 2048 cache sets.

B. FITTING INTO GAUSSIAN DISTRIBUTION

In this section, we further explore the probability density distributions for most benchmark programs and show that they can be essentially modeled as Gaussian distribution.

1) GAUSSIAN DISTRIBUTION

Gaussian distribution (or normal distribution) [25] is an important continuous probability distribution widely used in science and engineering. If a random variable X follows a Gaussian distribution with mean μ and variance σ^2, we denote $X \sim N(\mu, \sigma^2)$. The probability density of this distribution is modeled as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{3}$$

2) MODELING INTO GAUSSIAN

If the distribution of $MISS_k$ follows a Gaussian distribution [25], we denote it as $MISS_k \sim N(\mu, \sigma^2)$, then the following formula exists:

$$f(miss_k) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(miss_k-\mu)^2/2\sigma^2} \tag{4}$$

The parameters μ and σ^2 in formula (4) are the expectation and the variance respectively. The values of the two parameters can be estimated by the Maximum Likelihood Estimate (MLE) method [26]. According to MLE, in Gaussian distribution, the expectation and the variance for gross samples can be estimated by sample mean and sample variance. Let $\hat{\mu}$ denote the estimated sample mean and $\hat{\sigma^2}$ denote the estimated sample variance, then $\hat{\mu}$ and $\hat{\sigma^2}$ can be derived from the following two formulas:

$$\hat{\mu} = \frac{1}{N}\sum_{k=1}^{N} miss_k \tag{5}$$

$$\hat{\sigma^2} = \frac{1}{N-1}\sum_{k=1}^{N}(miss_k - \hat{\mu})^2 \tag{6}$$

With formula (4), (5) and (6), we can get an estimated Gaussian distribution function for the cache miss distributions for each experiment. The fitted Gaussian curves are shown as the dotted lines in Figure 3.

C. EVALUATION OF FITNESS

To evaluate the precision of the obtained Gaussian models, the Goodness of Fit Tests [26] was applied. We use y_i calculated in formula (2) to represent the concrete probability density distribution, and use Y_i to denote the probability

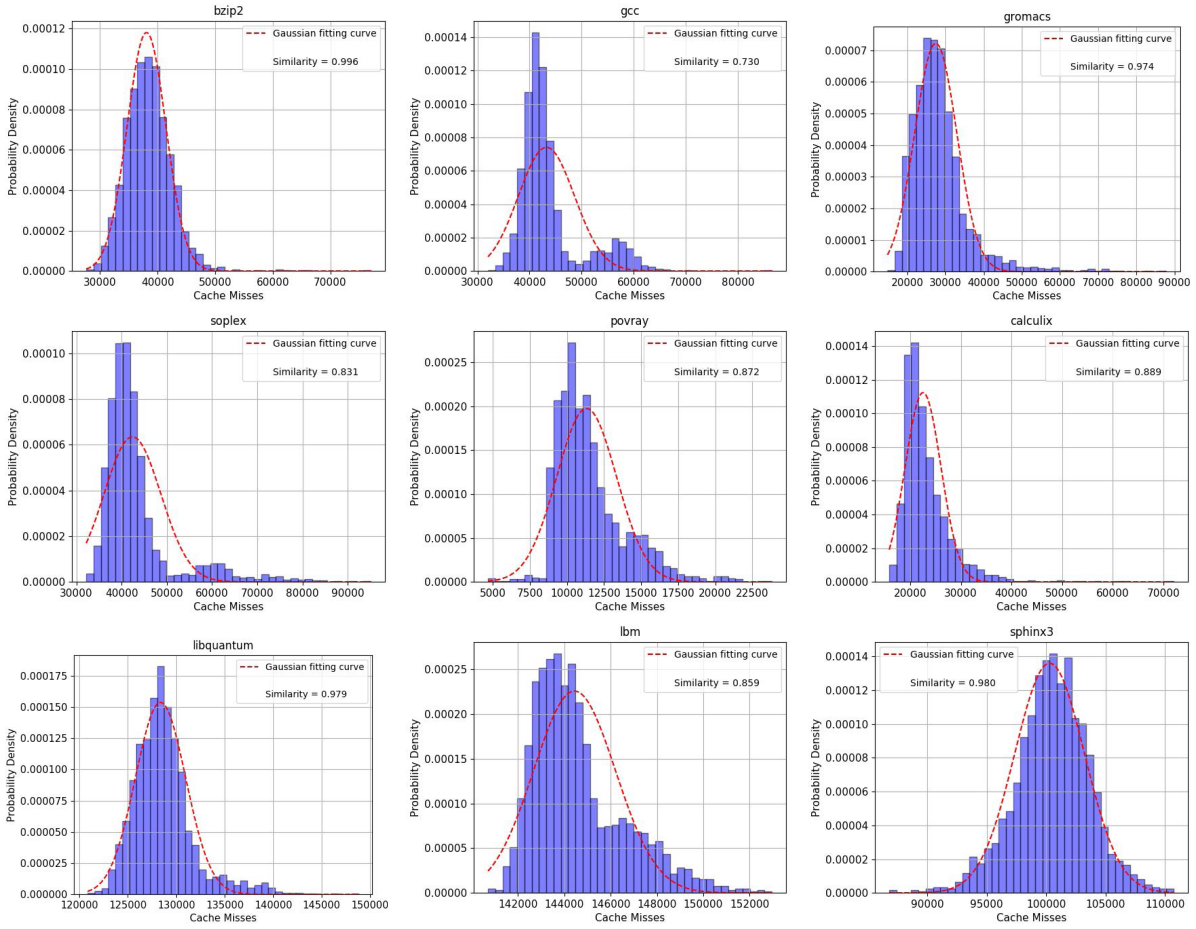


FIGURE 3. Gaussian modeling.

density distribution calculated by formula (4), where $i \in \{1, 2, \ldots, m\}$, μ and σ^2 are computed by formula (5) and (6).

The goodness of fitness between the Gaussian model and concrete result can be tested by Pearson correlation coefficient, which is denoted as R. Let $\bar{y}$ and $\overline{Y}$ denote the mean for y_i and Y_i respectively, the Pearson correlation coefficient R can be calculated by the following formula:

$$R = \frac{\sum_{i=1}^{m}(y_i - \bar{y})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{m}(y_i - \bar{y})^2 \cdot \sum_{i=1}^{m}(Y_i - \overline{Y})^2}} \tag{7}$$

The square of R (denoted as R^2) is the measurement of the similarity (the goodness of the fitness) between two curves. The range for R^2 is [0,1]. The closer to 1 the value of R^2 is, the better the two curves fit.

The similarity values between Gaussian model and concrete probability density distribution are shown in Fig. 3. Results show that the value of R^2 is larger than 0.85 for most benchmark program experiments, which indicates that Gaussian distribution can well represent the characteristics of cache miss distribution of programs.

We also explored other distributions, including gamma, Poisson, beta and t distribution, however, we find the Gaussian distribution to yield the best fit. This observation motivates an efficient cache set sampling method which will be detailed in the next section.

V. SET SAMPLING MODEL

As shown in the previous section, the cache miss behavior can be precisely modeled as Gaussian distribution. Since a Gaussian distribution function can be determined by 5 key points and each key point on the Gaussian curve corresponds to several cache sets, we can select a few corresponded cache sets and use their cache miss behavior to represent that of the whole program.

A. FEATURE SELECTION

It is shown that in Gaussian distribution function 5 representative sample points can determine a function [25]. The points are:

- Extreme value point, where $x = \mu$, the first order derivative of function $f'(x) = 0$;
- Two inflection points, where $x = \mu \pm \sigma$, the second order derivative of function $f''(x) = 0$;
- The starting point;
- The ending point.

The information carried by those points can be explained as follows: the starting and ending points define the boundary of the function; extreme value point and the inflection points determine μ and σ.

B. THE EFFICIENT CACHE PROFILING APPROACH

The aforementioned feature motivates us to use 5 key sample points to represent the obtained Gaussian curve that represents the cache miss characteristics of the stress program and thus indicates the cache occupancy or other cache access characteristics of the tested target program. Thus, ideally, it suffices to select 5 cache sets (the miss counts of which correspond to the 5 key sample points on the Gaussian curve) and to precisely obtain the gross cache access characteristics of a whole program by only monitoring the cache miss behavior on these 5 cache sets.

For now, we have a complete work flow to determine a few cache sets which can precisely represent the cache access characteristics of a program. The procedure is as follows:

- First, obtain the cache miss distribution of a target program by applying the stress program method presented in section III;
- Second, derive the Gaussian distribution function for this target program with the method presented in section IV;
- Third, get the 5 key sample points and find 5 corresponding cache sets as the final set sampling result.

By monitoring the cache behavior on these 5 cache sets of the stress program, we should be able to make predictions on, for example, cache occupancy of the target program.

However, in practice, several cache sets may have close number of cache misses and fall into the same cache miss count interval in Figure 3. Taking only one cache set for each sample point may introduce a large disturbance in the prediction. In our approach, we chose 5 cache sets for each sample point (i.e., the cache miss counts of these 5 sets fall into the cache miss count interval corresponding a specific key sample point on the Gaussian curve). Of course, taking 5 cache sets for each sample point is only a design parameter and one can take different number of cache sets for each point. The experimental results reported in the next section will show that good precision can already be achieved with $5 \times 5 = 25$ representative cache sets.

VI. PERFORMANCE EVALUATION

A. EXPERIMENTAL METHODOLOGY

We conduct experiments on the 9 benchmark programs (as target programs) presented in Section III to evaluate the precision of the proposed method and compare it with other cache set sampling methods.

6 methods are compared in our experiment, which are:

- Global access: this method records the total cache misses on all cache sets and thus serves as the ''correct answer''.
- Continuous 128: a state-of-the-art method [27] which chooses N continuous cache sets; here $N = 128$.
- Random 128: a state-of-the-art method [6], [10], [11] which randomly chooses N cache sets; here $N = 128$.
- Gaussian 25: the proposed method of this paper which chooses 25 sampling cache sets according to the procedure presented in section V-B.
- Continuous 25: choosing continuous 25 cache sets.
- Random 25: randomly choosing 25 cache sets.

We choose random cache sets sampling and continuous cache sets sampling for comparison as they are still the prevalent techniques employed in the recently proposed systems. For example, the Cache Monitoring Technology (CMT), which is launched at the Intel Xeon E5 v3 product family [6] and is also enabled in the later Intel processor family [28], employs the random cache sets to monitor the cache occupancy. The continuous cache sets sampling is used on Intel Ivy Bridge chips to implement the Set Dueling mechanism [27]. We have further performed the Set Dueling detection method proposed in [27] on processors such as Intel 4790 (Haswell microarchitecture), Xeon E5-2609 v4 (Broadwell microarchitecture), Xeon E5-2620 v4 (Broadwell microarchitecture) and find that they still employ continuous cache sets to implement the Set Dueling mechanism as well.

To evaluate the sampling methods, we concurrently run a benchmark with a stress program implemented in a set sampling method. The cache misses on each stress program are recorded for evaluations. In the results, the collected cache miss count for a given number of sampled cache sets is proportionally enlarged to predict that of the whole program.

In the experiments, the stress program repeatedly reads M (M equals to cache associativity) chains of memory accesses. Each chain has the same number of items which are located in the distinct cache blocks. Thus, if there is no contention with the stress program, this program won't incur cache misses. The location for the items in the chain is corresponding to the sampling method. For example, in the Continuous 128 method, each chain contains 128 items which are located at the same continuous cache sets. To evaluate the prediction precision under different cache contention intensities, we adjust the cache access frequency of the stress program, which is achieved by inserting different number of null operations between any two consecutive chain of accesses.

B. RESULTS AND ANALYSIS

To quantify the precision for a profiling method, we use the deviation of the miss number between the profiling method and ''Global access'' as a metric, denoted by D. At a stress frequency, let G_{miss} denote the miss value of Global access, and let T_{miss} denote the miss value of a profiling method, then:

$$D = \frac{|T_{miss} - G_{miss}|}{G_{miss}} \cdot 100\% \tag{8}$$

The experimental results for each benchmark are shown in Figure 4. In each chart, the x-axis is the number of accesses

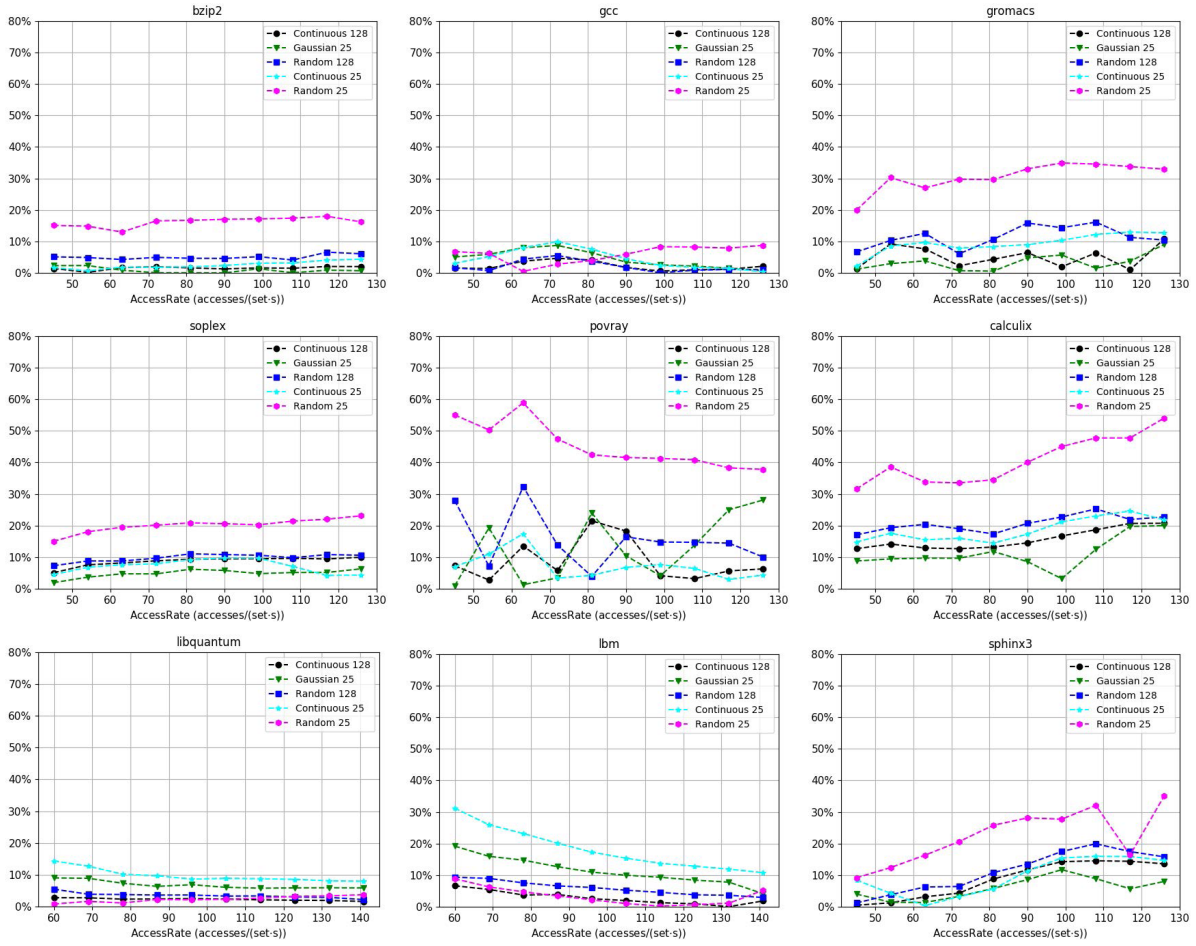


FIGURE 4. Performance deviations of each set sampling method on different cache contention intensities.

TABLE 1. Overall performance deviations for different set sampling methods.

Metrics	Methods				
	Cont128	Gauss25	Rand128	Cont25	Rand25
Average	6.2%	6.9%	9.5%	9.7%	20.8%
Worst	21.5%	28.1%	32.3%	31.2%	58.8%

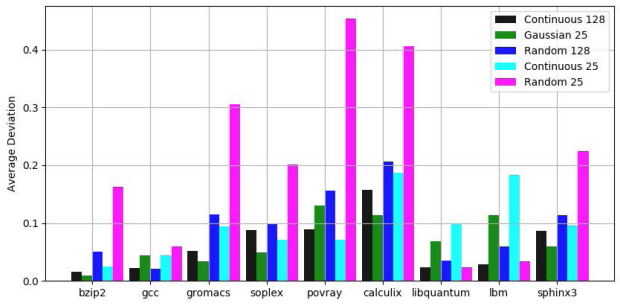
arrived on a cache set in 1 second and y-axis is the deviation value D.

To quantitatively compare the performance of each set sampling method, we compute the average and worst deviation results. Table 1 lists each set sampling method's average and worst deviation results among the overall experiments. Figure 5 shows each method's average and worst deviation on each benchmark. Note that for these metrics, all of them in our experiments are the lower-the-better ones.

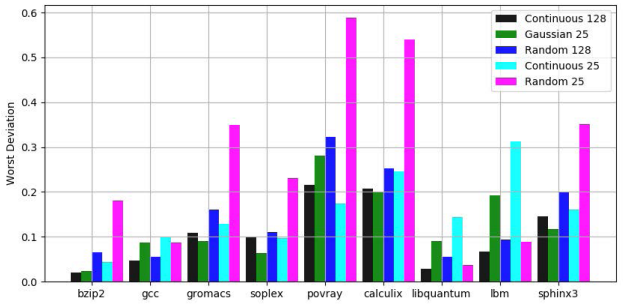
1) ANALYSIS

From the overall performance in Table 1, we can see that our proposed method produces comparable precision compared to the Continuous 128 which performs best in the average and worst deviation results. From the results on the 9 benchmarks in Figure 5, we can observe that our method and Continuous 128 achieve the most best deviation results: on the average and the worst deviations, our method achieves 5 times and 4 times best performance respectively; and Continuous 128 achieves 3 times and 4 times best performance respectively. By applying the same comparisons with "Continuous 25" and "Random 25", it can be observed that our method outperforms those two methods on the results in Table 1 and Figure 5.

Those comparisons show that our method can achieve the same level of precision compared to the two prevalent set sampling methods by using only one-fifth cache sets. The results demonstrate the effectiveness of our method that a program's cache access behavior, which can be modeled by Gaussian distribution, can be accurately represented by very few key points in its Gaussian distribution function. The cost in our method is to obtain the information to determine the Gaussian distribution function for the target program. Such information can be collected by our stress-program-based method presented in Section III. This stress-program-based method is simple and efficient, which doesn't need to modify/model the target processor and can be easily applied on current commercial multi-core processors.



(a) The average deviations for different set sampling methods



(b) The worst deviations for different set sampling methods

FIGURE 5. Performance deviations for different set sampling methods on each benchmark.

VII. CONCLUSION

In this paper, we proposed a Gaussian set sampling model for efficient shared cache profiling on multi-cores. We developed a method to probe the cache access distribution of a target program by simultaneously running a specially designed stress program and obtain the cache miss counts of the stress program. The key finding of our work is, cache miss distribution of the program can well fit into Gaussian distribution. Based on a key feature of Gaussian distribution, we were able to develop an efficient set sampling method which leverages much less set samples to achieve the same accuracy as by the state-of-the-art random sampling methods in predicting LLC cache performance.

REFERENCES

[1] R. Sen and D. A. Wood, "Reuse-based Online models for caches," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 279–292, Jun. 2013.

[2] K. Anand and R. Barua, "Instruction-cache locking for improving embedded systems performance," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, May 2015, Art. no. 53.

[3] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, May 2007.

[4] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, Jun. 2010.

[5] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 62–75.

[6] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family," in *Proc. HPCA*, Mar. 2016, pp. 657–668.

[7] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *Proc. Int. Conf. Parallel Process. (ICPP)*, Sep. 2011, pp. 165–175.

[8] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proc. MICRO*, Dec. 2011, pp. 248–259.

[9] K. Ji, M. Ling, L. Shi, and J. Pan, "An analytical cache performance evaluation framework for embedded out-of-order processors using software characteristics," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 4, Aug. 2018, Art. no. 79.

[10] R. E. Kessler, M. D. Hill, D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Trans. Comput.*, vol. 43, no. 6, pp. 664–675, Jun. 1994.

[11] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "CacheScouts: Fine-grain monitoring of shared caches in CMP platforms," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Techn.*, Sep. 2007, pp. 339–352.

[12] E. Berg and E. Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. IEEE Int. Symp. ISPASS Perform. Anal. Syst. Softw.*, Mar. 2004, pp. 20–27.

[13] N. C. Thornock and J. K. Flanagan, "Facilitating level three cache studies using set sampling," in *Proc. 32nd Conf. Winter Simulation*, Dec. 2000, pp. 471–479.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[15] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 344–355.

[16] X. E. Chen and T. Aamodt, "Modeling cache contention and throughput of multiprogrammed manycore processors," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 913–927, Jul. 2012.

[17] X. Xiang, C. Ding, H. Luo, and B. Bao, "HOTL: A higher order theory of locality," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 343–356, Mar. 2013.

[18] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer, "CRUISE: Cache replacement and utility-aware scheduling," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 249–260, Mar. 2012.

[19] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 miss rate curves on commodity systems for Online optimizations," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 121–132, 2009.

[20] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proc. Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2011, pp. 283–294.

[21] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling performance variation due to cache sharing," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2013, pp. 155–166.

[22] J. Mars, L. Tang, and M. L. Soffa, "Directly characterizing cross core interference through contention synthesis," in *Proc. 6th Int. Conf. High Perform. Embedded Archit. Compil.*. Jan. 2011, pp. 167–176.

[23] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 76–86.

[24] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 225–236.

[25] S. Ross, *First Course in Probability*. London, U.K.: Pearson, 2014.

[26] S. M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists*. New York, NY, USA: Academic, 2014.

[27] Y. Zhang, N. Guan, and W. Yi, "Understanding the dynamic caches on intel processors: Methods and applications," in *Proc. 12th IEEE Int. Conf. Embedded Ubiquitous Comput.*, Aug. 2014, pp. 58–64.

[28] K. T. Nguyen. (2016). *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family*. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology