ASTracer: An Efficient Tracing Tool for HDFS with Adaptive Sampling

Yang Song, Yunchun Li, Shuhan Wu, Hailong Yang, Wei Li

ASTracer: An Efficient Tracing Tool for HDFS with Adaptive Sampling

Yang Song, Yunchun Li, Shuhan Wu, Hailong Yang, and Wei Li

School of Computer Science and Engineering,
Beihang University, Beijing 100191, China
{yangsoon,lych,wushuhan,hailong.yang,liw}@buaa.edu.cn

Abstract. Existing distributed tracing tools such as HTrace use static probabilistic samplers to collect the function call trees for performance analysis, which may fail to capture important but less executed function call trees and thus miss the opportunities for performance optimization. To address the above problem, we propose *ASTracer*, a new distributed tracing tool with two adaptive samplers. The advantage of adaptive samplers is that they can adjust the sampling rate dynamically, which is able to capture comprehensive function call trees and in the meanwhile maintain the size of trace file acceptable. In addition, we propose an auto-tuning mechanism to search for the optimal parameter settings of the adaptive samplers in *ASTracer*. The experiment results demonstrate the adaptive samplers are more effective in tracing the function call trees compared to probabilistic sampler. Moreover, we provide several case studies to demonstrate the usage of *ASTracer* in identifying potential performance bottlenecks.

Keywords: HDFS · Distributed Tracing Tool · Adaptive Sampling.

1 Introduction

With the rapid development of the computing technologies, cloud computing has been widely adopted in large scale applications. Understanding the behavior of distributed systems and tracing the performance bottlenecks is becoming more complicated in the scenario of cloud computing. This is because services are deployed on different nodes, which is particularly difficult to locate abnormal behaviors within the massive volume of log files. Therefore, the distributed tracing tools are proposed to solve the above problems, which can be used to trace function calls in distributed systems to help users understand the system behaviors and analyze performance bottleneck. Currently, distributed tracing tools are widely used inside the large Internet service providers.

Moreover, popular big data analyzing frameworks such as Spark and Hadoop universally use distributed file systems such as HDFS [5] to store the large amount of data. Targeting HDFS, Htrace [1] is a distributed tracing tool for guiding the performance analysis and optimization of HDFS. Although tracing every function call within HDFS seems ideal for performance analysis, the huge

volume of trace data generated would make the data analysis infeasible. Therefore, Htrace relies on probabilistic samplers to collect a subset of all possible traces. The sampler used in Htrace determines the way how the function calls are collected based on probability.

The drawback with Htrace probabilistic sampler is that it determines to sample a call tree at the root node based on probability, therefore it decides either to sample the entire call tree or nothing. In some cases, such design of probabilistic sampler leads to low sampling rate, and thus fails to provide enough information of the function calls for the developers, especially the information of the abnormal functions. For instance, Table 1 shows the execution statistics of several functions for *nweight* in Hibench [8]. Some functions (e.g., *DFSOutputStream#writeChunk*) are executed for a large number of times, but take a quite short time to execute. Whereas, some functions (e.g., *FileSystem#createFileSystem*) are executed for only a few times, but take a long time to execute, which are more likely to be the performance bottlenecks. However, when using probabilistic samplers in Htrace, the low sampling rate is more likely to ignore these functions. At the same time, some function calls may be called more frequently than others, which may generate very large the trace file that buries the abnormal behaviors with tremendous less useful information. For instance, Table 2 shows the number of calls of several function for *kmeans* in Hibench. The function *DFSInputStream#byteArrayRead* has been executed for a large number of times, which greatly increases the size of the trace file.

Table 1. The execution statistics of several functions in *nweight*.

Function Name	Number of calls	$Time_{mean}(ms)$	$Time_{std}(ms)$
DFSOutputStream#writeChunk	3489	0.042	0.350
DFSOutputStream#write	400	1.520	1.882
BlockSender#sendPacket(transferTo)	177	31.717	75.610
BlockSender#doSendBlock	48	121.145	156.751
DFSOutputStream#close	40	267.175	229.670
FileSystem#createFileSystem	20	1244.350	641.715

Table 2. The number of calls for several functions in *kmeans*.

Function Name	Number of calls
DFSInputStream#byteArrayRead	1644123
DFSOutputStream#writeChunk	4963
BlockReaderRemote2#readNextPacket	251
ClientNamenodeProtocol#getFileInfo	219
DFSInputStream#fetchBlockAt	131

To solve the above problems, we propose a new tracing tool *ASTracer* for HDFS. The *ASTracer* extends *HTrace* with two adaptive samplers, which records the number of function calls at the root node of the call tree in the sampler, and generates sampling decisions for different root nodes based on the recorded in-

formation. For instance, *ASTracer* limits the sampling rate of the call tree that is executed frequently, and ensures that the call trees that are executed less frequently have at least the minimum number of samples. Because the sampling decision is made for each call tree, it guarantees to capture the execution information of more functions. Moreover, *ASTracer* reduces the number of samples from the frequently executed call trees, which is effective to compress the size of the trace file. In addition, we propose several metrics from various aspects such as efficiency, storage and sampling quality to evaluate the effectiveness of the proposed samplers. Compared to the probabilistic samplers, *ASTracer* is able to capture more function call relationships while maintaining a small size of trace file.

Specifically, the main contributions of this paper are as follows:

- We propose *ASTracer*, a new distributed tracing tool with two adaptive samplers for increasing the coverage of function call sampling, as well as maintaining the size of the trace file acceptable.
- We design an auto-tuning mechanism to search for the optimal parameter settings within *ASTracer*, which eliminates the overhead of human effort and time cost of exhaustive search.
- We present several important metrics from various apsects, including efficiency, storage and sampling quality to evaluate the effectiveness of the proposed samplers in *ASTracer*.
- We provide a case study by applying *ASTracer* to analyze representative workloads, which identifies potential performance bottlenecks and gives guidance for performance optimization.

The rest of this paper is organized as follows: Section 2 introduces the background of distributed tracing tools as well as the motivation of this paper. Section 5 presents the related work on the samplers of distributed tracing systems. We present the design and implementation of our *ASTracer* with two adaptive samplers, as well as the automatic tuning method for the sampler parameters in Section 3. We evaluate the effectiveness of *ASTracer* in Section 4, and conclude this paper in Section 6.

2 Background and Motivation

2.1 HDFS

HDFS is a distributed file system proposed in Hadoop, but it is also used in other distributed computing frameworks such as Spark. HDFS is highly fault-tolerant and suitable for deployment on commodity clusters. It provides functionalities such as error checking and automatic data recovery. The HDFS cluster adopts the master-slave model, which consists of a *NameNode* and several *DataNodes*. The *NameNode* is responsible for managing the namespace, storing metadata, etc., whereas the *DataNode* performs operations such as creating, deleting, and copying the blocks under the scheduling of the *NameNode* in order to meet the requests from the *Client*.

2.2 Distributed Tracing Tool

To cope with the complicated tracing demand in the distributed systems, Google proposes Dapper [14] that builds the tracing tool based on call tree and span. Another typical tracing tool is Xtrace [7], which is able to provide a comprehensive view of the system service behaviors. However, it is incapable to handle distributed systems at very large scale. Currently, the widely used distributed tracing tools include Zipken [2], Jaeger [3] and Htrace [1]Among them, Htrace is a tracing tool specially designed for HDFS. The design of Htrace is based on the following concepts: *1)* a *Span* object represents a function being traced. *2) TraceScopes* manages the life time of *Span* objects, and the *Tracers* are responsible for creating a *TraceScope*. *Tracer* determine whether to sample a function call by calling *Sampler*. *3) Spanreciver* is a collector, which is responsible for receiving *Span* objects sent from *Tracer* and serializing trace data. In this paper, we leverage the *LocalFileSpanReciver* to periodically write sampling data to trace files.

2.3 Motivation

Certain call trees in HDFS application may be executed frequently. Sampling such call trees is not only unnecessary, but also consumes significant computation and storage resources. In addition, generating large trace files could severely degrade the performance of the running application. Moreover, the huge volume of the trace data is also difficult to analyze. However, there are few research works focusing on the design of adaptive samplers in distributed tracing tools, especially in the field of big data application. The samplers in Dapper [14] all adopt a global sampling rate. Zipkin [2] supports more samplers such as counting sampler and boundary sampler, however it fails to consider the execution behaviors of different call trees. Jaeger [3] also misses the dynamic sampling functions in its current implementation [4]. Htrace [1] only provides probabilistic sampling and equidistant sampling that are infeasible to change during the tracing.

It is clear that there is still much work to do for improving the effectiveness of samplers used in distributed tracing tool. For instance, how to improve the coverage of call trees during sampling, and in the meanwhile reduce the size of the trace file. With detailed function call trees sampled, especially when abnormal behaviors happened during the execution, the developers can effectively identify the performance bottlenecks and optimize accordingly. All the above needs motivate this paper.

3 The Design and Implementation of ASTracer

3.1 The Design Overview

The design overview of *ASTracer* is shown in Figure 1. First, the HDFS application is instrumented. When the application is executed, *ASTracer* decides whether to sample certain call trees. The samplers in *ASTracer* make sampling

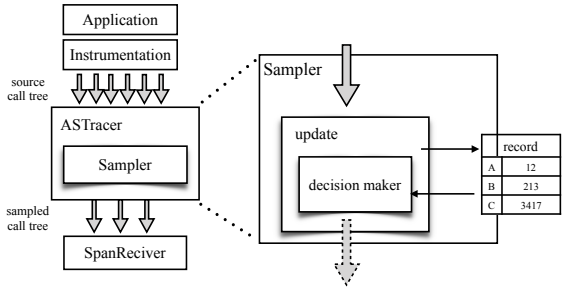

Fig. 1. The design overview of *ASTracer*.

decisions for the root nodes of each call trees, which can be approximated as sampling the call tree. In *ASTracer*, we use the *record* table to record how many times each call tree has been called. The sampler determines whether to sample a call tree based on the number of occurrences of the call tree.

To solve the problem of missing call tree with global sampling rate, *ASTracer* adjusts the sampling rate dynamically according to the number of occurrences of the call trees. the workloads that contain a large number of iterations, the number of occurrences of different call trees could differ by even 5 to 6 orders of magnitude. Sampling such workloads requires dynamically adjusting the sample rate in order to capture enough call trees without generating too large trace files.

The sampler works in the following way within *ASTracer*. The sampler is consulted for sampling decision when the root node of a function call tree is traced by the *Tracer* in *ASTracer*. The sampler updates the record and then generates a sampling decision based on the record.

3.2 Bump Sampler

Bump sampler uses the bump function to generate the sampling decision with probability. The advantage for using the bump function is that the sampling probability changes significantly when the input variable exceeds a certain threshold. With this property, we can guarantee that each function has a high probability of being sampled before a specified threshold. However, after exceeding the threshold, the sampling rate drops dramatically.

The bump function used in the bump sampler is shown in Equation 1, where x represents the number of times a function is being called. The property of the bump function is that when the number of occurrence of a function is small, the sampling probability is almost 1. However, when a function is being called more often, the sampling rate starts to decrease rapidly. In order to avoid the non-sampling problem with the functions that are being called for a large number of times in the later, we set a minimum sampling rate. Moreover, in order to prevent the frequently executed functions being sampled too less, we create a new thread when instantiating the sampler, and reset the number of function calls in the record table to be 0 every second.

$$f(x) = 1 - e^{-\frac{\lambda^2}{x^2}} \tag{1}$$

The bump sampler works as follows. It first checks whether the record of the function already exists in the record table. If not, a new entry is created, in which the number of function calls is initialized to 0. If there is a record of the function, the bump function is used to generate a new sample rate based on the number of function calls. Then, the number of calls to this function is increased by one and the record is updated. The algorithm determines whether the sampling rate is lower than the lowest sampling rate. If so, the sampling rate is set to the lowest. The threshold for the number of function calls as well as the lowest sampling rate can be customized by the users.

3.3 Token Bucket Sampler

Token bucket sampler is based on the idea of token buckets [9]. The design of token bucket sampler is to maintain a bucket with a certain number of tokens. The number of tokens in the bucket only vary within the range of 0 and bucket capacity. Each time a function is called, the tokens in the bucket are decremented by one. The tokens are replenished to the bucket at a certain rate.

In our token bucket sampler, we set a bucket for the root node of each call tree during workload execution. When the sampler is consulted, it first looks up the bucket to see if there are any tokens left, updates the tokens according to the policy of the token bucket, and decides whether to sample. Instead of using the static sampling rate, it decides whether to sample based on the remaining tokens in the bucket. The advantage of this sampler is that frequently occurring call trees are suppressed, and the call trees that occur less frequently are almost always taken. In particular, when a function call occurs in a burst for a short time period, the sampler can effectively compress the number of samples taken.

The token bucket sampler works as follows. It first checks whether there is an entry for the function in the table. If not, a new entry is created and initialized. Based on whether there is at least one token for the function remained in the token bucket, the token is updated according to the time elapsed from the last execution, however without exceeding the bucket capacity. The bucket capacity as well as the rate for replenishing tokens can be customized by the users.

3.4 Auto Tuning the Sampler Parameters

Since the optimal parameter settings for the samplers vary across different applications as well as distributed systems, it is more effective to use an auto-tuning mechanism to search for the optimal parameter settings for the samplers in *AS-Tracer*. Therefore, we propose an auto-tuning mechanism using the simulated annealing algorithm [10].

The objective function $f(\boldsymbol{x})$ as shown in Equation 2. For bump sampler, $\boldsymbol{x} = (\lambda, threshold)$, whereas for token bucket sampler, $\boldsymbol{x} = (bucket\_size, increase\_step)$. The $entropy(\boldsymbol{x})$ represents the information entropy of the sampling result. The

larger the entropy is, the more information the trace collects. The $dist(\boldsymbol{x})$ measures the similarity between the sampled results and the full instrumented results, which uses the Euclidean distance. The smaller the Euclidean distance is, the higher the similarity is.

$$f(\boldsymbol{x}) = \frac{dist(\boldsymbol{x})}{entropy(\boldsymbol{x})} \tag{2}$$

The constraints to the objective function $f(\boldsymbol{x})$ is shown in Equation 3, where $S_{p0.1}$ indicates the trace size sampled using 0.1 probability, and S represents the trace size sampled by the adaptive sampler after the parameter auto-tuning. That is, while ensuring a small size of compressed trace file, it will not lose too much information.

$$0.1 \cdot S_{p0.1} \leq S \leq S_{p0.1} \tag{3}$$

The parameter auto-tuning using the simulated annealing algorithm works as follows. First, it generates a random initial solution $\boldsymbol{x}$ and calculates its objective function $f(\boldsymbol{x})$. A new solution $\boldsymbol{x'}$ is then proposed by adding a perturbation, and then a new objective function $f(\boldsymbol{x'})$ is calculated. If the constraint is not met, a new solution $\boldsymbol{x'}$ is re-proposed. In order to choose a better solution, let:$\delta f = f(\boldsymbol{x}) - f(\boldsymbol{x'})$, if $\delta f \leq 0$, replace $\boldsymbol{x}$ with $\boldsymbol{x'}$. However, in order to prevent the algorithm trapping in a local optimal solution, it is necessary to accept a sub-optimal solution with certain probability. The simulating annealing algorithm accepts $\boldsymbol{x'}$ with probability $p = e^{-\frac{\delta f}{T}}$, where T is the current temperature to control the acceptance probability of a sub-optimal solution. The above process iterates until the upper limit is reached. Then the temperature T is decreased and the number of iterations is reset. The above procedure is repeated until the condition is met.

The optimal parameter settings of the samplers after auto-tuning using the simulated annealing algorithm are shown in Section 4.1.

4 Evaluation

4.1 Experimental Setup

Our experiments are conducted on a cluster with five nodes, which includes one master node, three slave nodes, and one client node running HDFS v2.8.3. Each node is equipped with 2 Intel Xeon E5-5620 processors and 16GB DDR3 memory. The operating system on each node is 64 bit CentOS v6.5. We collect trace file from the *Client* and *Namenode* for result analysis. Representative workloads are selected in Table 3 to demonstrate the robustness of *ASTracer*. To the best of our knowledge, there is no public tracing tool available on HDFS except for Htrace. Therefore, we compare with the static samplers in Htrace with the sampling rate set to 0.1 and 0.01, which is commonly used in literature [14]. The parameter settings for the samplers in *ASTracer* are also shown in Table 3.

Table 3. The parameter settings in *ASTracer*.

	Probability Sampler	Bump Sampler		Token Bucket Sampler	
	sampling rate	λ	threshold	bucket size	replenish rate
dfsioe_read	0.1	128	0.022	1047	21
dfsioe_write	0.1	215	0.020	2595	9
terasort	0.1	552	0.014	3728	12
wordcount	0.1	1034	0.013	5002	14
kmeans	0.1	3490	0.010	21035	19
pagerank	0.1	102	0.021	1083	116

4.2 Evaluation Metrics

To better evaluate the samplers in *ASTracer*, we propose the five metrics including execution time (ET), trace file compression ratio (TFCR), sampling coverage (SC), sample similarity (SS) and information entropy (IEn), to measure the effectiveness of the samplers from different aspects. We provide a brief description about SS and IEn in the following subsections.

Sample Similarity represents the similarity to the trace results with call trees all sampled. The calculation of SS is as follows: for a sampler B, assume that it samples function m and function n. Then we use the feature vector $\boldsymbol{F_B} = ((mean_m, std_m), (mean_n, std_n))$ to represent the sampling characteristics of the sampler, and $\boldsymbol{F_A}$ represents the feature vector with call trees all sampled. After that, we calculate the Euclidean distance between $\boldsymbol{F_A}$ and $\boldsymbol{F_B}$ as shown in Equation 4, where n represents the number of all functions. A closer Euclidean distance means higher similarity.

$$\begin{aligned} SS &= dist(\boldsymbol{F_A}, \boldsymbol{F_B}) \\ &= \sqrt{\sum_{k=1}^{n} \left[(A_k.mean - B_k.mean)^2 + (A_k.std - B_k.std)^2 \right]} \end{aligned} \tag{4}$$

Information Entropy can be used to describe the information uncertainty in a system [13]. The higher uncertainty means higher information entropy. IEn is calculated using Equation 5. The three properties of information entropy are monotonicity, non-negativeness and additivity. According to monotonicity, the more likely a sample occurs, the less information it carries. In other words, the samples with low probability to occur are more valuable to us. Whereas the non-negativity and additivity ensure that we should focus on high-value samples.

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x) \tag{5}$$

The calculation information entropy is as follows: for a sample result, it calculates the execution time ($count_x$) for each function as well as the total number of calls for all functions ($C = \sum_{x \in \mathcal{X}} count_x$,). Then, it calculates the frequency of each function $p(x) = count_x / C$ and applies $p(x)$ to Equation 5.

4.3 Sampler Evaluation

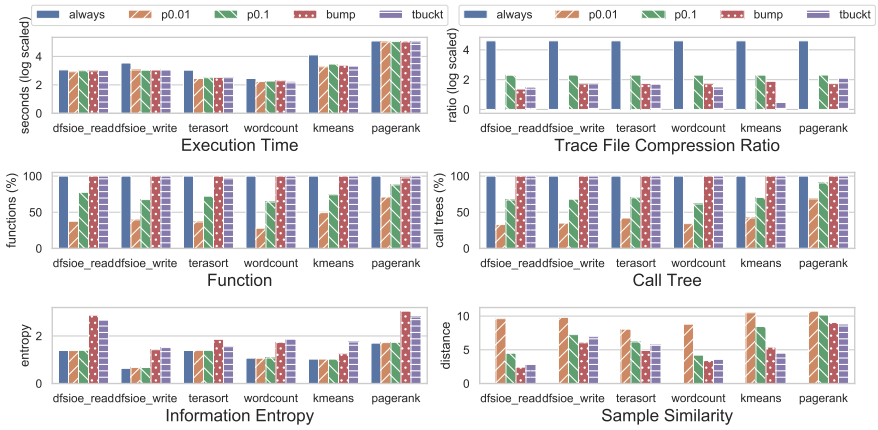

Fig. 2. The evaluation results with different samplers under different metrics. The legend *always*, *p0.01*, *p0.1*, *bump* and *tbuckt* mean the methods of sampling all functions, with probability 0.01, with probability 0.1, bump sampler and token bucket sampler respectively. The execution time, trace file compression ratio, function and call tree of all sampling methods are normalized to *always*.

To reduce the impact of system noise, we run each workload for 10 times under each evaluating metric and report the mean of the results. The results are shown in Figure 2.

In terms of execution time, because our sampler uses *ConcurrentHashMap* to store information for parallel accesses, it has less impact on the performance of the workload. Compared to the workload execution time without the sampler, the average sampling latency with bump sampler and token bucket sampler is 6.99% and 5.49% respectively across different workloads, whereas the average sampling latency with probability sampler (rate=0.1) is 7.92%.

In terms of trace file compression ratio, compared to collecting all samples, the trace file size generated by *ASTracer* is compressed to about 5%, probability sampler (rate=0.1) is approximately 10%. ASTracer significantly reduces the number of samples of functions that performs too many times, so the size of the trace is also reduced considerably. Note that reducing the number of samples seldom leads to insufficient information about such functions. In addition, we can adjust the sampler parameters to achieve the optimal results.

In terms of sampling coverage, our samplers can capture more functions and call trees, whereas probabilistic samplers fail to capture more functions as the sampling rate decreases. Compared to collecting all samples, both bump sampler and token bucket sampler can achieve close to 100% coverage across different workloads, whereas the average coverage of the probabilistic sampler is 72%.

In terms of information entropy, our adaptive samplers reduce the sampling rate of some high-probability functions and improves the sampling rate of some low-probability functions, therefore it can obtain more information. The average information entropy of the bump sampler and token bucket sampler across different workloads is 2.03 and 2.02, whereas the probabilistic sampler (rate=0.1) only achieves 1.21.

In terms of sample similarity, when comparing to itself, the SS is 0 when all call trees are sampled. Therefore, the sampler with SS close to 0 is better. The average SS of the bump sampler and token bucket sampler across different workloads is 5.20 and 5.40 respectively, whereas the SS of probabilistic sampler (rate=0.1) is 6.80. Therefore, the adaptive samplers preserve more statistical characteristics of the samples than probabilistic sampler.

4.4 Case Study

In this section, we provide several case studies using *AStracer* to identify several abnormal function calls with workloads in Hibench.

In general, all calls to the *DFSInputStream#byteArrayRead* function in Hibench are considered as abnormal. After analyzing the execution time distribution of this function, we observe that the 75% percentile of execution time is less than 0.1ms, however the maximum execution time is as long as 100ms. This indicates that when the workloads read data, the size of data block is extremely unbalanced.

The machine learning algorithm such as *kmeans* requires multiple iterations, and thus calls the *DFSInputStream#byteArrayRead* function frequently. Table 4 shows the sampled function information of *kmeans*. We can see that the *DFSInputStream#byteArrayRead* function is called more often and the execution time is unbalanced. Therefore, the problem of data skew has a significant impact on the performance of such workload.

Table 4. The sampled functions of *kmeans and pagerank*.

workload	Function	Number of calls	$time_{mean}$ (ms)	$time_{std}$ (ms)	$time_{median}$ (ms)	$time_{max}$ (ms)
kmeans	DFSInputStream#byteArrayRead	22635	0.041	1.042	0.1	120
	ClientNamenodeProtocol#getFileInfo	219	1.01	4.712	1	70
	ClientNamenodeProtocol#addBlock	22	12.10	9.310	12.5	52
pagerank	DFSInputStream#byteArrayRead	1713	0.149	0.804	0	24
	DFSOutputStream#write	1726	0.162	0.417	0	5
	ClientNamenodeProtocol#create	247	8.846	12.022	7	106

Pagerank is an algorithm for measuring the importance of a particular web page. In particular, *pagerank* is a computation intensive workload. A lot of work is used to build directed graphs through link relationships. The number of function calls to *DFSInputStream#byteArrayRead* and *DFSInputStream#write* is much fewer than other workloads as shown in Table 4. In addition, the execu-

tion time is quite even across function calls. Therefore, the I/O operations are unlikely to become a performance bottleneck.

We also observe that the execution time of the *Client* (e.g., *ClientNamenodeProtocol#create* and *ClientNamenodeProtocol#getFileInfo*) varies significantly as shown in Table 4, which obtains the metadata from *NameNode* via RPC. Although such function is only called for a few times, its execution time is usually long and thus could become the potential performance bottleneck. Whereas, Htrace fails to capture the above information and thus loses the opportunity for performance optimization.

5 Related Work

In the design and optimization of samplers for distributed systems, the Dapper experience from Google [14] emphasizes the dynamic adjustment of the sampling strategy for different workloads, which reduces the sampling rate under high load conditions, and increases the sampling rate under low load conditions to ensure that the coverage of the trace. In addition, Liu et. al. [11] use Htrace to analyze the performance of HDFS, and propose a compressed tree algorithm to reduce the size of the trace file, however their algorithm can only be used for offline compression.

Jaeger [3] is a distributed tracing system developed by Uber. It is used to monitor the health of the system. Its implementation is based on Dapper. Jaeger is mainly composed of *jaeger-client*, *jaeger-agent* and *jager-collector*. The *jaegar-agent* is responsible for forwarding the recorded data to the *jaegar-collector*. And it can dynamically adjust the sampling frequency.

Adaptive features are widely studied in performance analysis and tracing systems [6, 15, 12]. The main idea of these works is based on the runtime information, dynamically adjusting the pre-set parameters to achieve a certain purpose. However, there is little research work on sampling. Therefore, this paper attempts to introduce adaptive sampling into the tracing system in order to achieve better sampling results.

Different from existing works, this paper proposes adaptive samplers by extending the tracing system Htace. Each time the sampler is called, the number of calls to the function (the root node of the call tree) is recorded. According to this record, the sampler can adjust its sampling rate according to the dynamic strategies.

6 Conclusion

In this paper, we propose a new distributed tracing tool *ASTracer* with two adaptive samplers that adjusts the sampling rate dynamically to improve the effectiveness of function tracing from various aspects. The experiment results show that our proposed samplers are better than the probabilistic sampler under various evaluating metrics. Moreover, we provide several case studies to

apply *ASTracer* in identifying the performance bottlenecks with representative workloads.

Acknowledgement

This work is supported by National Key Research and Development Program of China (Grant No.2016YFB1000304) and National Natural Science Foundation of China (Grant No. 61502019). Hailong Yang is the corresponding author.

References

1. https://github.com/apache/incubator-retired-htrace/
2. https://zipkin.io/
3. https://www.jaegertracing.io/
4. https://github.com/jaegertracing/jaeger/issues/365/
5. Borthakur, D.: The hadoop distributed file system: Architecture and design. Hadoop Project Website **11**(2007), 21 (2007)
6. Ehlers, J., van Hoorn, A., Waller, J., Hasselbring, W.: Self-adaptive software system monitoring for performance anomaly localization. In: Proceedings of the 8th ACM international conference on Autonomic computing. pp. 197–200. ACM (2011)
7. Fonseca, R., Porter, G., Katz, R.H., Shenker, S., Stoica, I.: X-trace: A pervasive network tracing framework. In: Proceedings of the 4th USENIX conference on Networked systems design & implementation. pp. 20–20. USENIX Association (2007)
8. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010). pp. 41–51. IEEE (2010)
9. Humayun, F., Babar, M.I.K., Zafar, M.H., Zuhairi, M.F., et al.: Performance analysis of a token bucket shaper for mpeg4 video and real audio signal. In: 2013 IEEE International Conference on Smart Instrumentation, Measurement and Applications (ICSIMA). pp. 1–4. IEEE (2013)
10. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
11. Liu, Y., Li, Y., Zhou, H., Zhang, J., Yang, H., Li, W.: A fine-grained performance bottleneck analysis method for hdfs. In: IFIP International Conference on Network and Parallel Computing. pp. 159–163. Springer (2018)
12. Mos, A., Murphy, J.: Compas: Adaptive performance monitoring of component-based systems. In: Proceedings of 2 nd ICSE Workshop on Remote Analysis and Measurement of Software Systems. Citeseer (2004)
13. Shannon, C.E.: A mathematical theory of communication. Bell system technical journal **27**(3), 379–423 (1948)
14. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure (2010)
15. Wert, A., Schulz, H., Heger, C.: Aim: Adaptable instrumentation and monitoring for automated software performance analysis. In: Proceedings of the 10th International Workshop on Automation of Software Test. pp. 38–42. IEEE Press (2015)