ADLER: Adaptive Sampling for Precise Monitoring

Arnamoy Bhattacharyya(✉) and Cristiana Amza

Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
{arnamoyb,amza}@ece.utoronto.ca

Abstract. In this paper, we present ADLER, a tool for profiling applications using a sampling frequency that is tuned at program runtime. ADLER can not only determine the adaptive sampling rate for any application, but also can instrument the code for profiling so that different parts of the application can be sampled at different frequencies. The frequencies are selected to provide enough information without collecting redundant data. ADLER uses performance models of program *kernels* and prepare the kernels for sampling according to their complexity classes. We also show an example use case of real-time anomaly detection, where using ADLER's execution models, the anomalies can be detected 23% quicker than static sampling.

1 Introduction

Application sampling is widely used for a number of scenarios: (1) application phase detection [15], (2) anomaly detection [5] (3) improving energy efficiency [13]. Choosing an appropriate sampling frequency to correctly capture the behaviour of an application is quite important. Choosing a high frequency may give rise to redundant data thus incurring unnecessary storage and analysis overhead, while sampling at a low frequency may fail to capture enough information. Moreover, different applications have parts of code that show different execution behaviour. Therefore, setting a static sampling frequency is not the right choice for correctly capturing the behaviour of an entire application.

Correctly capturing data though application profiling at an optimal sampling frequency is also necessary for other use cases, for example, anomaly detection in the cloud [5]. In large-scale cloud systems like Cassandra, HBase, stateful components are expected to be many, and failures are expected to be the rule rather than the exception; for example, one hardware failure per data center, per day is commonly reported. Moreover, the necessary maintenance activities for monitoring, diagnosis, inspection or repair can no longer be handled through frequent human intervention. New approaches that predict the resource consumption of cloud applications [1] and provide automatic solutions for anomaly detection are more applicable today. For fast and effective anomaly identification in real time, an adaptive strategy for monitoring application execution and resource usage

© Springer Nature Switzerland AG 2019
L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 86–100, 2019.
https://doi.org/10.1007/978-3-030-35225-7_7

is very important. Adaptive sampling provides a balance between the storage overhead of the profiled data and the processing time of the profiled data to detect anomalies.

In this paper we propose a compiler based tool called ADLER (ADaptive sampLER) that instruments the application for adaptive sampling. ADLER takes application bytecode and different input configurations. It then builds performance models for program kernels and cluster them according to their performance complexity classes. The output from ADLER is application code that is instrumented to set the sampling frequency on the fly as the application runs with a particular input. We show the effectiveness of ADLER in reducing the storage overhead from a high static sampling frequency sampling while still keeping enough information to correctly identify anomalies. We present results for a wide range of database server applications written in multiple programming languages. We show that ADLER is able to efficiently switch the sampling frequency at minimum performance penalty. We also show the effectiveness of adaptive sampling an example use case of real-time performance anomaly identification in database servers running in the cloud.

2 Motivational Experiment

In this section, we provide a motivational experiment to show that a proper sampling frequency is necessary for correctly capturing a program's runtime behaviour. For this experiment, we set up an Hbase server. We monitor the CPU utilization of the HBase process running the Yahoo! Cloud Serving Benchmark (YCSB) [5] workload over time.

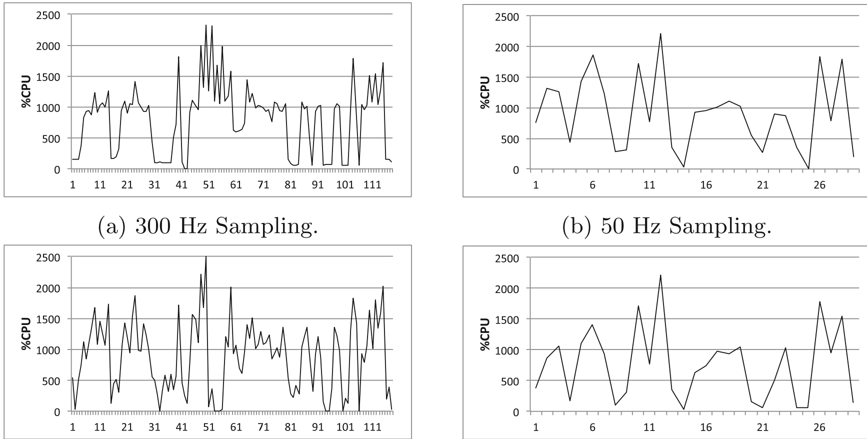
Figure 1 shows the CPU utilization of a HBase server when the YCSB workload is run on it. The sampling frequency for the CPU utilization is set at 300 HZ. There is a *busy* phase when the workload is run in the server. Also before the beginning and at the end of the service, there are *setup* and *cleanup* phases that the application uses to set up and clean up tables for running the workload.

In Figs. 1(c) and (d) we introduce a disk anomaly during the setup phase (at around time 51 in Fig. 1(c)). This anomaly can be detected using a real-time anomaly detection technique using sampled data about resources [5].

Figures 1(b) and (d) show the same scenario but with a lower frequency of sampling (50 Hz). Here the CPU utilization patterns with and without the presence of an anomaly are not clearly distinguishable due to the sparse collection, therefore, the anomaly is not detected.

The motivational experiment clearly shows the necessity of a good choice of sampling frequency for understanding the behaviour of servers running in the cloud. The sampled data can not only be used for resource anomaly detection, but also for debugging the code, phase analysis, application optimization, VM migration decisions [9].

The sampling frequency should not be very high as well, because that may give rise to a lot of redundant data that incurs both storage and analysis complexity overhead. Therefore, a technique of adaptive sampling, where the frequency



(a) 300 Hz Sampling.

(b) 50 Hz Sampling.

(c) 300 Hz Sampling with anomaly introduced.

(d) 50 Hz Sampling with anomaly introduced.

Fig. 1. Motivational Experiment with different sampling frequencies for a HBase server serving a YCSB workload. A higher sampling frequency captures more information, helping in Anomaly detection (at around time 51), but a lower sampling frequency, though can save space, fails to detect the same anomaly.

changes depending on the overall application structure is necessary. In the next section, we provide our methodology for adaptive sampling that can correctly capture the behavior of applications without incurring too much storage and analysis overhead.

3 A New Method for Adaptive Sampling

In this section, we provide a detailed description of our adaptive sampling technique. Our methodology consists of two main steps:

- Estimate a sampling frequency based on the execution time models of program *kernels*.
- Modify the sampling frequency on the fly according to the complexities of the *kernels* during the program execution for a given input.

3.1 Execution Time Modeling

The first step of the adaptive sampling methodology is to build precise performance models of *kernels*. An execution time model of a program *kernel* is a function that can estimate the execution time of a kernel based on the program inputs. We provide the definition of kernels and describe how we generate execution time models of those kernels below.

Program Kernels. We identify *loops* and *functions* in the program as program *kernels*. We represent the performance M of a program through the execution time models m of n *kernels*:

$$M = \{m_1, m_2, \ldots, m_n\} \tag{1}$$

We define the execution time model m of each kernel as a linear regression function of a set of *predictors* $p = \{p_1, p_2, \ldots, p_p\}$.

$$m = \sum_{i=1}^{|p|} \alpha_i \cdot p_i + \beta \text{ where } p_i \in p \tag{2}$$

A predictor p_i is a function of one or more program input parameters ι. If there are r input parameters that influence the performance of a *kernel*, the predictor set is formed by applying a set of transformations $\tau_1, \tau_2, \ldots \tau_v$ on those input parameters.

$$p = \{\bigcup_v \bigcup_r \tau_v(\iota_r)\} \tag{3}$$

Values of Model Parameters. The first task is to assemble a list of all input parameters that significantly influence the runtime of the application. We call such parameters *critical (input) parameters.*

Critical parameters should be scalar values such as sizes of dimensions, number of iterations or the percentage of reads and writes during a workload. If the execution time of the program is determined by an input file or a vector, then it should be condensed into the smallest number of scalar critical parameters (e.g., if the input file is a sparse matrix, the critical parameter could be the number of non-zero elements in the matrix). A domain expert has to determine the complete set of parameters and supply them. We identify the set of parameters as $P = (p_1, p_2, \ldots, p_n)$.

Model Fitting. We use an empirical method to determine the execution time model of the kernel in terms of its input parameters. In constructing models to predict performance and put locations into clusters, we make use of "least-squares linear regression and power law regression". Regression selects model parameters that minimize some measure of error. We use the LASSO statistical method proposed by Bhattacharyya et al. [6] to determine the execution time model of the kernel. Following this approach, the *predictors* are formed by applying powers and logarithm transformations on program inputs. The search space of *predictors* is constructed from program input parameters using the following normal form:

$$p = \{\iota_i^k \log^l \iota_i^k,\ k, l \in \mathbb{R},\ \iota_i \in I\} \tag{4}$$

Here I represents the set of program input parameters. By assigning different values to k and l, the predictor set is constructed from the input parameters.

An example model from EPMNF for program input parameters ι_1 and ι_2 would be $c_1 \cdot \iota_1^2 + c_2 \cdot \iota_2 \log \iota_2$, where c_1 and c_2 are constants.

We generate execution time models for each *calling context* of a kernel. We define a calling context of a kernel following the Loop Call Graph (LCG) [4] of the program as following:

Definition 3.1 *A context C of a kernel is defined as the set of nodes of the LCG that are visited during a particular instance of execution.*

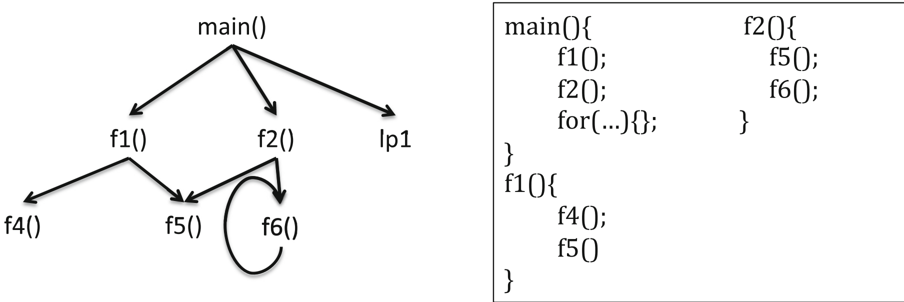


Fig. 2. Sample code with corresponding LCG.

Figure 2 shows a sample code and its corresponding LCG. According to our definition of context, the LCG will have the following contexts.

1. `main` → `f1`
2. `main` → `f1` → `f4`
3. `main` → `f1` → `f5`
4. `main` → `f2`
5. `main` → `f2` → `f5`
6. `main` → `f2` → `f6`
7. `main` → `lp1`

While profiling the kernels for the construction of execution time models, we model each calling context separately. Therefore, at the end of the execution time modeling step, we have one execution time model per kernel per calling context. It is possible for a kernel to have multiple execution time models due to execution from different calling contexts.

Measure of Fit. As our method for constructing execution time model is based on empirical method, the constructed model sometimes does not reflect the theoretical exact execution time model. Therefore, we need to measure the goodness of fit of the constructed model so that it can be effective utilized to tune the sampling rate in the following step. For measuring the goodness of

fit of the constructed model, we use the adjusted R^2 (cite pemogen) statistic, a measure of the model's goodness-of-fit that quantifies the fraction of the variance in execution time accounted for by a least-squares linear regression on the inputs:

The adjusted R-square (ARS) of the predictions by the model is calculated on the test data:

$$\mathrm{R}^2 \equiv 1 - \frac{\sum_{i=1}^{x} (y_i - f_i)^2}{\sum_{i=1}^{x} (y_i - \bar{y})^2} \tag{5}$$

$$\mathrm{ARS} = \mathrm{R}^2 - (1 - \mathrm{R}^2)\frac{m}{x - m - 1} \tag{6}$$

Where x and m are the test data batch size and number of parameters respectively.

3.2 Adaptive Sampling

After we generate the execution time models for all the kernels in the program, we have to set execution points in the loop call graph of the program where we want to switch the sampling frequency. If we switch the sampling frequency for the execution of each kernel in each context, the overhead from sampling will be too high, resulting in a high drop in application throughput. Therefore, in this section we provide a novel approach for adaptive sampling based on complexity classes of the kernels at various calling contexts.

Complexity Classes. We cluster the execution time models of kernels as the following four main classes. This clustering of kernels helps to modify the sampling frequency switching to keep the sampling frequency switching at a minimum. Since the sampling frequency switching requires communication between the program and the sampler, a frequency switch at the beginning of execution of each kernel will produce too much runtime overhead.

1. Logarithmic Class: The kernels belonging to the logarithmic class have the following normal form of the execution time model.

$$p = \{\sum \log^l \iota_i,\ l \in \mathbb{R},\ \iota_i \in I\} \tag{7}$$

2. Linear Class: The kernels belonging to the linear class has the following normal form of the execution time model.

$$p = \{\sum \iota_i^k,\ k \in \{1\}\ \iota_i \in I\} \tag{8}$$

3. Polynomial Class: The kernels in the polynomial class has the following normal form:

$$p = \{\iota_i^k \log^l \iota_i^k,\ k \in \{2, 3\}, l \in \mathbb{R},\ \iota_i \in I\} \tag{9}$$

   It is important to note that we consider two kernels with execution time models $O(n^2)$ and $O(n^2 * \log n)$ to be in the same complexity class because their asymptotic behaviour is roughly the same.

4. Unknown Class: All kernels whose execution time models do not achieve a good fit for the training data, belong to a Unknown complexity class. We consider a value of 0.95 for the ARS a good fit.

Grouping of Kernels. Once we have identified all the different complexity classes of the kernels, we instrument the code to prepare it for adaptive sampling. The instrumentation prepares the code to communicate with the sampling tool to modify the sampling frequency on the fly during application deployment. Our goal in this grouping step is to minimize the communication between the application and the sampling tool, while still collecting enough information through sampling to capture the complete behaviour of the application.

To group the kernels, we use two information:

- The complexity class of the kernel.
- The calling context of the kernel in the LCG.

Our grouping algorithm starts from the leaves of the LCG. For each leaf of the LCG, we also check the calling context of the kernel to determine its *level*. The instrumentation adds codes for either setting the sampling for the respective kernels. The result of the instrumentation is to produce a code that after adaptive sampling, will generate the same number of data points for each kernel at each calling context level. This means that a kernel with a higher execution time will need a lower sampling frequency while a kernel with a smaller execution time will be in need of a higher sampling frequency. The setting of sampling frequency uses both the static structure and the runtime information about the kernels.

During static check, all the kernels belonging to the same complexity class is sampled against the same frequency. Therefore, code for switching is added only once for these kernels of the same group. But if two kernels belong to different complexity classes (where the input parameters in the execution time model are different), we take a look at the execution time trends of the kernels obtained during the execution time model generation. If the trend shows that the kernels do not differ from each other by more that 5% in their execution time for the different input parameter values, we do not switch the sampling frequency during the kernels switch. The number 5%, according to our experiments, provides the sweet spot between the number of sampling switches and the quality of the collected data.

The static analysis begins with kernels at the deepest calling context level (the highest number of nodes in the calling context). It processes the leaf kernels at the same level of the LCG. Once the leaves at the lowest level have been processed, the analysis moves one level up and applies the same clustering strategy. Once the processing of all the leaves at all calling context levels is done, our instrumentation for the code necessary for frequency switching per context is complete.

Setting the Sampling Frequency. Once the instrumentation of the switching of sampling frequency is done, the setting of actual sampling frequency is done during the program execution as this is input specific.

At runtime, the switching code first calculates the predicted execution time of a kernel at a particular calling context based on the values of input parameters during that particular run. After calculating the execution time, based on the given number of data points necessary for capturing the program behaviour, the frequency is set. The required number of samples per kernel per calling context can be set by the analyst and that is a compromise between the resource one has vs. the amount of information one wants to collect about the program behaviour. For the kernels with *Unknown* execution time models, the execution time is conservatively predicted to be the minimum of all the execution times of that kernel during training and the sampling frequency is set according to that.

Example. In this section we give a complete example of sampling frequency switching using our kernel grouping heuristics. Listing 1.1 shows a sample code and Fig. 3 shows the corresponding LCG for the code. Figure 3 is also annotated with the execution time models of the relevant of the LCG.

Listing 1.1. Example code for Adaptive Sampling.

```
main(){
        f1();
        //non-kernel code
        for(...){
          f6();
        } //lp2
        //non-kernel code
        for(...){} //lp3
}
f1(){
        f4();
        //non-kernel code
        for(...){} //lp1
        //non-kernel code
        f5();
}
```

We first start our instrumentation for all the leaves in the LCG of the program. In the given LCG, there are five leaves: (1) `f4()` (2) loop1 (`lp1`) (3) `f5()` (4) `f6()` and (5) loop3 (`lp3`). As a first step, we have to identify all the leaves that belong to the same calling context. We can see from the graph that the three leaves (`f4()`, `lp1` and `f5()`) belong to the same calling context which is `main()` → `f1()`. Therefore, first we process them. Here let us assume that during the execution time model generation with different input parameter values, the execution time trends of `f4()` and `lp1` do not vary by more than 5%. Therefore, according to our heuristics, even though they have different input parameter values in their respective execution time models, they belong to the same linear

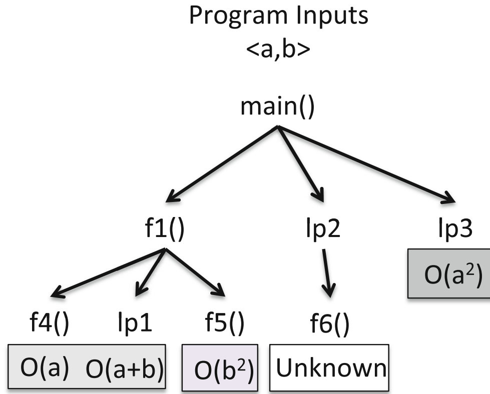


Fig. 3. Loop Call Graph and their respective sampling groups for the code in Listing 1.

complexity class. As a result, we do not have to switch sampling frequency during the kernel switch. But `f5()` belongs to a different (quadratic) complexity class. When the code switches from the execution of kernel `lp1` to `f5()` we perform a sampling frequency switch.

Next we process the next leaf node of the graph that is `f6()`. This node alone belongs to the calling context `main()` → `lp2`. Therefore, this kernel is instrumented with its own sampling frequency code. As the kernel belongs to a Unknown complexity class, the sampling frequency will be set according to the smallest execution time of this kernel during the execution time model generation phase.

Once we finish processing of all the leaf kernels at the deepest level, we move one level up and process the leaf kernel `lp3`. `lp3` has the calling context `main()` and it is the only kernel belonging to this context. Therefore, it will be instrumented with the sampling frequency according to the predicted execution time for the given input at runtime. Listing 1.2 shows the instrumented code with our adaptive sampling method.

Listing 1.2. Instrumented for Adaptive Sampling.

```
main(){
        f1();
        //non-kernel code
        for(...){
          predict_and_setfreq(f6);
          f6();
         } //lp2
        //non-kernel code
        predict_and_setfreq(lp3);
        for(...){} //lp3
}
f1(){
        predict_and_setfreq(f4);
```

```
        f4();
        //non-kernel code
        for(...){} //lp1
        //non-kernel code
        predict_and_setfreq(f5);
        f5();
}
```

4 Implementation

In this section we provide details about the implementation of our tool ADLER. As seen in Fig. 4, the tool is composed of three components.

1. **Execution time model Generator:** The execution time model generation engine of ADLER has all the capabilities built for generating the execution time models of the kernels in the program.
2. **Adaptive Sampler:** The adaptive sampler engine of ADLER takes the execution time models generated by the execution time model generator engine and then instruments the original code for adaptive sampling.
3. **Compiler Analysis:** The compiler component of ADLER has two compilers that support intermediate languages: (1) LLVM for C/C++ and (2) Soot [16] for Java. The execution time model generator and the adaptive sampler components both are connected to this component.

ADLER takes as input source code files and produces an instrumented version of the source code ready for adaptive sampling. If the source code is not available, ADLER can work with intermediate representations of the code as well. For the intermediate representation of C and C++ programs, we use the LLVM's intermediate language. The LLVM compiler is widely used by programming language research these days the intermediate representation of the code gives the flexibility to work across microarchitectures.

Similar to C/C++ applications, for Java ADLER supports both source code files and class/ jar files that are essentially intermediate representation of source code in the Java language.

The analyst supplies the source code files, the language of analysis and the values and names of the input parameters for the given code. ADLER first performs static analysis of the source code and instruments the source code for execution time model construction. This produces an instrumented version of the source code, which, when run by the user with different input parameter values, produces profile files with timing information per run.

Next, the profile files are fed back into the execution time model generator component of ADLER to learn the execution time models of the kernels inside the code. The execution time models of the kernels are written to files by the model generator engine.

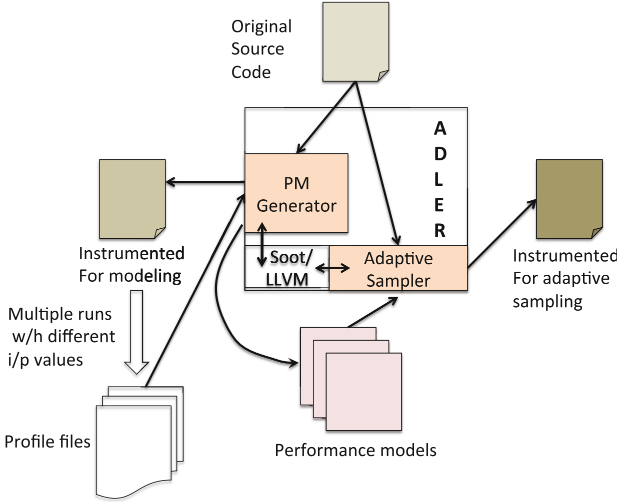


Fig. 4. Different components of ADLER and the complete workflow.

In the next step, the adaptive sampler engine of ADLER uses the execution time models learned at the previous step to perform instrumentation for preparing the code for adaptive sampling. The instrumentation in this step does not go on top of the previous instrumentation because our tool buffers the original code.

5 Experimental Evaluation

In this section, we present the effectiveness of adaptive sampling using ADLER for a number of popular cloud database servers written in both C/C+ and Java. Though our method for adaptive sampling is versatile and can be used in any use case where sampling needs to be performed, we focus our use case on real time resource anomaly detection on the cloud. We choose the YCSB [1] workloads for running on the cloud servers. The various parameters of the YCSB workloads give us different input values to train the execution time models of kernels. We first present the study on the kernel characteristics of the servers and how adaptive sampling is effective in grouping the kernels based on their complexity classes. Then we present a detailed study on an anomaly detection use case.

5.1 Execution Time Modeling

In this section, we present the results from the execution time modeling engine of ADLER. We report the total number of kernels in each of the databases we use for experiments and their complexity classes. We also report how many sampling frequency switching points are created by ADLER to show the effectiveness of

Table 1. The total number of kernels and their complexity classes for the codebases of our experiments.

Codebase	Language	# kernels	Log	Linear	Polynomial	Unknown	# Switching
Hbase-1.1.0	Java	488953	45	488529	279	100	2536
Cassandra-3.0	Java	133331	44	133141	89	57	4789
Elasticsearch-2.3.3	Java	134581	10	134416	90	65	2987
MongoDB-3.0.14	C/C++	298822	12	298735	45	30	546
ArangoDB-4.3.61	C++	1539	30	1473	20	16	656
Memcached-1.4.37	C	120	3	108	6	3	54

the grouping strategy. We run each database with 100 different input parameter combinations, each for 10 times.

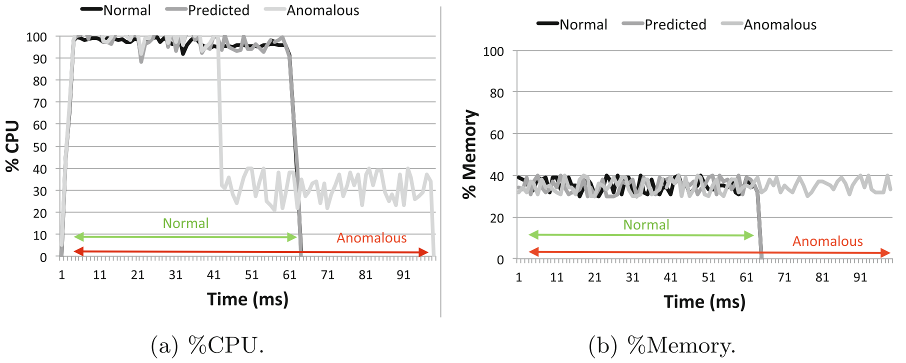
As seen from Table 1, most of the kernels can be correctly classified into complexity classes, with a few unknowns. Also most kernels belong to a linear complexity class for the databases. We see in the last column of Table 1, that the number of switching points introduced by ADLER is relatively low, which shows the effective clustering of the kernels in these applications. Grouping a large number of kernels in a smaller number of clusters also indicate the existence of recurrent phases in applications that have a significant number of kernels with similar complexity classes.

5.2 Case Study: Anomaly Detection

In this section we show an example use case of ADLER in case of anomaly detection. We use the execution time models to detect the anomaly during program runtime. Our anomaly detection technique closely relates to the method proposed by Bhattacharyya et a. [5] in the sense that we annotate raw resource usage data with semantic information (with the kernels). But unlike them, we use the predicted execution time from the execution time models and compare against the actual execution time during a program run of kernels for detecting anomalies.

For building the execution time models, we use 10 different configurations of YCSB workload for a total of 1000 runs. Then for testing the accuracy of anomaly detection, we use the *systemtap* tool to simulate a faulty disk anomaly. During the execution of a YCSB workload on Cassandra, we inject a delay of 50 s each in 50% of the reads and 50% of the writes to disk coming from the database. We keep injecting the delay for a period of 10 s.

Table 2 shows the methods from Cassandra that represent the workload processing phase. The actual and predicted execution times for a *normal* run and the actual execution time of an *anomalous* run are also shown. It is clearly seen that by comparing the predicted execution time with the actual execution time at runtime, the anomaly can be detected.



(a) %CPU. (b) %Memory.

Fig. 5. The CPU and Memory consumption by the kernel org.apache.cassandra.io.util. ByteBufferOutputStream.write() for normal and anomalous runs for Cassandra.

Table 2. Predicted and actual execution times for Cassandra kernels for anomalous runs. Pred is Predicted execution time and Anom is Anomalous run.

Kernel	Normal	Pred	Anom
org.apache.cassandra.io.util.ByteBufferOutputStream.write()	150 ms	160 ms	300 ms
org.apache.cassandra.utils.PureJavaCrc32.update()	52 ms	60 ms	150 ms
org.apache.cassandra.io.util.ChecksummedOutputStream.write()	100 ms	95 ms	234 ms

Root-Cause Analysis. With our adaptive sampling methodology, we are able to perform a root cause analysis of the anomaly by correlating the monitored usage of different resources. Figure 5 shows the CPU and memory utilization of one of the Cassandra kernels (`org.apache.cassandra.io.util. ByteBufferOutputStream.write()`) during the normal and anomalous runs.

To learn the characteristics of normal runs, we use the method described by Bhattacharyya et al. [5]. By looking at the figure, it can be seen that there is not much change in the memory utilization for the disk fault anomaly but in CPU utilization, there is a noticeable difference. By correlating the resource utilization data with the execution time difference, we can identify this anomaly type. For a different anomaly e.g. memory leak, the difference in the memory usage pattern between a normal and anomalous run will become more significant.

Adaptive sampling can help in root cause analysis by reducing the analysis complexity of the amount of collected data. In an online setting, this is crucial. Using ADLER, we are able to perform the root cause analysis for the anomaly 23% faster after the end of the busy phase. Also, at a lower sampling frequency (e.g. the default sampling frequency of *gprof*), due to the lack of enough data points, the anomaly root cause analysis cannot be performed.

6 Related Work

Symantec i3 for J2EE [8] is a commercial tool that features the ability to adaptively instrument Java applications based on the application response time. Rish et al. [14] describe a technique, called active probing. Kumar et al. [10] apply transformations to the instrumentation code to reduce the number of instrumentation points executed as well the cost of instrumentation probes and payload. A technique to switch between instrumented and non-instrumented code is described by Arnold and Ryder [2]. Munawar and Ward [12] argue that a monitoring system should continuously assess current conditions by observing the most relevant data, it must promptly detect anomalies and it should help to identify the root-causes of problems. The magpie [3] and the Pinpoint [7] are also two well-known projects of the field. Magalhaes et al. [11] provides an approach for adaptive profiling and probably the closest to our work. But in contrary to them, our approach is not application and workload specific and it is not turned on only when anomaly is detected.

7 Conclusion

In this paper we present a tool for adaptive sampling – ADLER. ADLER can prepare applications that can self-adapt sampling frequencies on the fly based on the application input. We show an use case of ADLER in anomaly detection for web servers running on the cloud. Compared to a static sampling at high frequency, ADLER can improve the delay in anomaly root-cause analysis by 23%, making it very effective in real-time anomaly detection. ADLER can be used for any use case where sampling is necessary.

References

1. Yahoo Cloud Service Benchmarks. https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/
2. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. ACM SIGPLAN Not. **36**(5), 168–179 (2001)
3. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: OSDI, vol. 4, p. 18 (2004)
4. Bhattacharyya, A., Hoefler, T.: Pemogen: automatic adaptive performance modeling during program runtime. In: 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 393–404. IEEE (2014)
5. Bhattacharyya, A., Jandaghi, S.A.J., Sotiriadis, S., Amza, C.: Semantic aware online detection of resource anomalies on the cloud. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 134–143. IEEE (2016)
6. Bhattacharyya, A., Kwasniewski, G., Hoefler, T.: Using compiler techniques to improve automatic performance modeling. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp. 468–479. IEEE (2015)
7. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic internet services. In: Null, p. 595. IEEE (2002)

8. Symantec Corporation: Symantec i3 for J2EE - performance management for the J2EE platform
9. Jandaghi, S.J., Bhattacharyya, A., Sotiriadis, S., Amza, C.: Consolidation of underutilized virtual machines to reduce total power usage. In: Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, pp. 128–137. IBM Corp. (2016)
10. Kumar, N., Childers, B.R., Soffa, M.L.: Low overhead program monitoring and profiling. ACM SIGSOFT Softw. Eng. Notes **31**(1), 28–34 (2006)
11. Magalhaes, J.P., Silva, L.M.: Adaptive profiling for root-cause analysis of performance anomalies in web-based applications. In: 2011 10th IEEE International Symposium on Network Computing and Applications (NCA), pp. 171–178. IEEE (2011)
12. Munawar, M.A., Ward, P.: Adaptive monitoring in enterprise software systems. SysML, June 2006
13. Padmanabha, S., Lukefahr, A., Das, R., Mahlke, S.: Trace based phase prediction for tightly-coupled heterogeneous cores. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 445–456. ACM (2013)
14. Rish, I., et al.: Adaptive diagnosis in distributed systems. IEEE Trans. Neural Netw. **16**(5), 1088–1109 (2005)
15. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. IEEE Micro **23**(6), 84–93 (2003)
16. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: a java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224. IBM Corp. (2010)