2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

PADLA: A Dynamic Log Level Adapter Using Online Phase Detection

Tsuyoshi Mizouchi*, Kazumasa Shimari*, Takashi Ishio†, Katsuro Inoue*
* Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
Email: {m-tys, k-simari, inoue}@ist.osaka-u.ac.jp
† Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan
Email: ishio@is.naist.jp

***Abstract*—Logging is an important feature for a software system to record its run-time information. Although detailed logs are helpful to identify the cause of a failure in a program execution, constantly recording detailed logs of a long-running system is challenging because of its performance overhead and storage cost. To solve the problem, we propose PADLA (Phase-Aware Dynamic Log Level Adapter) that dynamically adjusts the log level of a running system so that the system can record irregular events such as performance anomalies in detail while recording regular events concisely. PADLA is an extension of Apache Log4j, one of the most popular logging framework for Java. It employs an online phase detection algorithm to recognize irregular events. It monitors run-time performance of a system and learns regular execution phases of a program. If it recognizes a performance anomalies, it automatically changes the log level of a system to record the detailed behavior. In the case study, PADLA successfully recorded a detailed log for performance analysis of a server system under high load while suppressing the amount of log data and performance overhead.**

***Index Terms*—Dynamic Analysis, Log Level, Log4j, Phase Detection, Performance Anomaly**

I. Introduction

Logging is a common practice to record run-time information of program execution [7]. Developers insert logging statements to record run-time information such as error messages and actual values of variables. The information is often used in failure diagnosis [10], [11], [17], program comprehension [15], and so on.

Logging in detail incurs system run-time overhead (e.g. CPU consumption and I/O operations), while logging too little may miss necessary run-time information [5]. Long-running systems like a web service may produce a huge amount of logs and consume a huge amount of disk space. For instance, a service system of Microsoft can produce dozens of Terabytes of logs per day [6]. Such a high overhead might cause adverse effects such as service delays.

To address the trade-off, popular logging libraries such as Apache Log4j [3] provide log levels to control the amount of logs recorded in a storage. Developers assign a log level to each of logging statements in a system. A logging library filters log messages by comparing the log levels with a dynamic log level specified by a user of the program. In case of Log4j, six log levels are available by default: `fatal`, `error`, `warn`, `info`, `debug`, and `trace`. Those levels represent the degree of importance of messages. The `error` level represents error messages, the `info` level represents important but normal event, and the `trace` level represents detailed information for tracing a program execution, respectively [21].

The log levels are easy to use but ineffective to analyze irregular behavior such as performance anomaly of long-running systems. Such behavior infrequently occurs during daily operations. Since the `info` level does not record detailed logs, reproducing those behavior is difficult [20]. On the other hand, logging the entire execution of a system with the `trace` level is unrealistic due to the huge amount of logs and performance overhead.

To solve the problem, we propose PADLA, an extension of Apache Log4j, that dynamically adjusts the log level of a running system. It employs an online phase detection algorithm that divides a program's execution into a sequence of phases according to their dynamic properties or traits [14]. PADLA monitors run-time performance metrics of a system and learns regular phases of the system. It recognizes a performance anomaly as an unknown phase. It automatically changes the log level of the running system to record detailed logs for the irregular events while suppressing the amount of logs for regular events.

To evaluate PADLA, we have conducted a case study using a web application running on Tomcat and a load testing tool. As a result, PADLA successfully recorded the information necessary for performance analysis from the server system under high load with a small performance overhead compared with the `trace` level logging.

In the remainder of the paper, Section II explains the detail of the tool. Section III shows our case study that demonstrates the effectiveness of the tool. Section IV describes related work. Section V describes the conclusion and future work.

The PADLA tool is publicly available on GitHub: https://github.com/244-penguin/PADLA.

II. PADLA Log Level Adapter

PADLA is an extension of Apache Log4j that dynamically adjusts the output log level of a running system according to the run-time behavior of a system. The tool keeps the logging interface of Log4j. Developers can activate the tool by adding PADLA to their `log4j.xml` file and arguments for a program execution. It does not need to modify the source code of a target system.

978-1-7281-1519-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPC.2019.00029

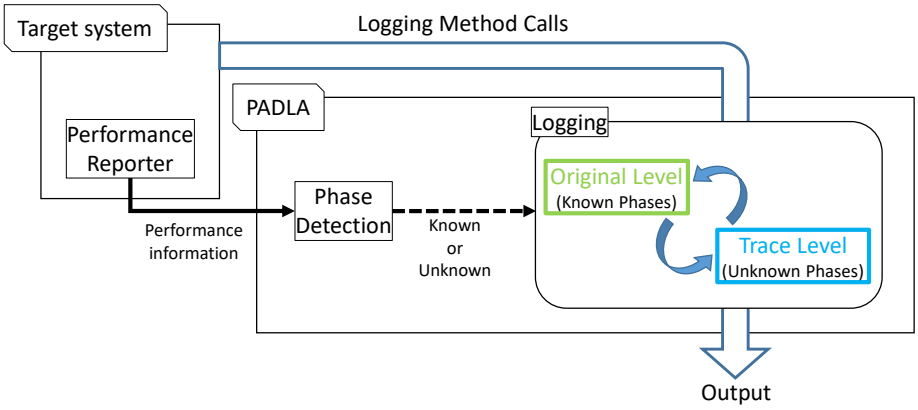

Fig. 1. An overview of PADLA

The tool takes as input a log level for recording regular behavior, e.g. `info` level. The tool also takes execution scenarios for learning regular behavior. If unavailable, the tool learns regular behavior on the fly. The tool produces application logs in the same format as Log4j using the specified level for regular events and the `trace` level for irregular events, respectively.

The tool comprises two components: Phase Detection and Logging. Figure 1 shows an overview of these components. The phase detection component monitors a program execution and divides it into a sequence of phases according to their performance characteristics. It classifies each phase as either known or unknown. The logging component buffers recent log messages and filters them using the original level specified by a user for regular phases and the `trace` level for unknown phases, respectively. The following subsections explain each of components in detail.

A. Phase Detection

PADLA employs an online phase detection method proposed by Reiss [14]. The method firstly divides a program execution to fixed time intervals. It then translates the t-th interval into a performance vector v_t. Two intervals belong to the same phase if the cosine similarity of their vectors v_s and v_t is greater than a threshold. Otherwise, the intervals belong to different phases.

We employ a performance vector used in an existing performance profiling tool [12]. Each element in a vector represents the actual execution time consumed by a corresponding method in the interval. To observe performance vectors, PADLA uses dynamic byte-code instrumentation to inject a performance reporter to a target application. The injected code reports the time spent for each executed method. PADLA obtains a performance vector every 0.5 seconds.

PADLA maintains a set of known vectors V_k to classify an interval as either known or unknown. The t-th interval is classified as a known phase if there exists a vector $v \in V_k$ that is similar to v_t. The interval is classified as an unknown phase if no vector in V_k is similar to v_t. The classification result is passed to the logging component that dynamically adjusts the log level of the running program.

To construct the initial V_k for a program, PADLA requires executions of the program that cover regular behavior. If those scenarios are available, V_k is a set of performance vectors observed in the executions. If unavailable, PADLA starts with $V_k = \phi$.

The component updates V_k to learn unknown phases as regular behavior if they repeatedly occur during an execution. The learning is necessary because an unknown phase may continue for a long time due to various reasons. For example, a permanent change of a run-time environment may affect the performance of a system. The log level change of PADLA also may affect performance characteristics. If the component records detailed logs of repeated occurrences of an unknown phase, the amount of log data and performance overhead will be huge. So it records detailed logs of only the first few intervals of a long-duration phase for analysis. In the current implementation, we add a vector v_t to V_k if two consecutive time intervals (i.e. one second) belong to the same unknown phase. In other words, the unknown phase is likely a new regular behavior for the system.

B. Logging

The logging component writes log messages to a storage according to the classification of the latest phase. Since an unknown phase is recognized after its occurrence, the component internally buffers recent N log messages ($N = 300$ in our current implementation) on memory using the `trace` level. During known phases, the logging component filters the detailed messages out but delays the output of messages. If an unknown phase is detected, all the buffered log messages are written to log files, and then the component starts to record logs using the `trace` level. The buffered messages provide detailed behavior of a few intervals before the detected unknown phase. They enable developers to investigate what actually happened in an unknown phase.

It should be noted that the overhead of the buffering is small because an application using Apache Log4j always

produces log messages irrelevant to a run-time log level. An existing logger in Log4j filters the generated log messages for each logging method call. Our logging component delays the filtering step until a phase classification.

III. Case Study

To demonstrate the usefulness of PADLA, we conduct a case study that records the behavior of a web application under a load test. We analyze (1) the run-time cost of the tool in terms of the size of log files and performance overhead, and (2) usefulness of the log contents to understand irregular behavior under high load.

A. Target Application

We execute JPetStore 6 [1], an online shop demo application, on Apache Tomcat 8.5.34 [4]. To enable PADLA to learn regular behavior of the system, we firstly conduct a load test with a low load configuration. We then conduct a load test with a high load configuration as an irregular event of the system.

A load test starts the server, waits for 30 seconds, and then starts a number of threads to access the pages on the system. The low load configuration uses 100 threads that each access to the system 10,000 times. The high load configuration uses 10,000 threads. Each of them also access to the system 10,000 times. All the threads follow the same access pattern. The load test is conducted with Apache JMeter 5.0 [2] and Windows 10 Pro running on Xeon E5-1650v3 processor, 32 GB RAM, and a SSD.

We measure the baseline of an execution using Log4j with the `info` level. We use Log4j with the `trace` level to measure the cost of detailed logs. We compare the performance with PADLA using the `info` level for regular events. We measure the log file size and time three times for each configuration and compute the average cost of the tool.

B. Result

PADLA successfully changed the log level of the system. Figure 2 shows an excerpt of the produced logs. Log messages on the `debug` and `trace` levels appeared from 30 seconds after the Tomcat started. The time corresponds to the load test with the high load configuration. Since the high load test continues for a while, PADLA automatically learns the high load condition as a known phase, i.e. a new regular behavior. PADLA changes the log level of the system to the original `info` level after recording detailed logs for 10.6 seconds on average. After the change, no more unknown phases are recognized by PADLA until the end of a load test.

Table I shows the size of log files and the time for each configuration. Compared with the baseline of the `info` level, logging with the `trace` level increased the file size by 1,431 MB and the execution time by 74%. On the other hand, PADLA increased the file size only 34.1 MB and the execution time by 30%. The result shows that the tool can suppress the size of log files and performance overhead.

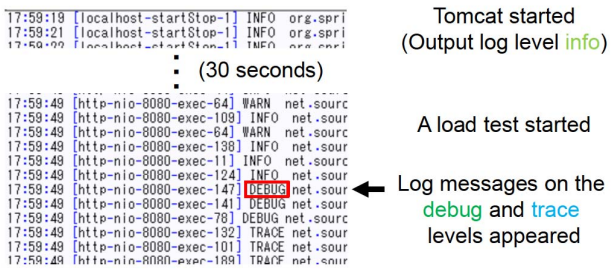

Fig. 2. Output Log file

TABLE I
Performance of PADLA

Output Log Level	Log File Size	Execution Time
Log4j `info` (Fixed)	11.9 MB	1 min 14 sec
Log4j `trace` (Fixed)	1,442.9 MB	2 min 9 sec
PADLA (Dynamic)	46.0 MB	1 min 36 sec

We also investigated the contents of logs in the log file produced by PADLA. Listing 1 shows example log messages that are likely useful to understand the behavior of the system under high load. The first line of Listing 1 shows that the `actions/Catalog.action` page was accessed. The second line shows that the `viewCategory()` method was called when the viewCategory event occurred. Those lines indicate page names and method calls that were accessed during the irregular period. They also indicate the number of accesses for a certain period of time. These information are helpful for performance analysis such as examining the performance limit of the system.

IV. Related Work

The original Apache Log4j provides two ways to dynamically change the log level of a running system. The first one is API (e.g. `Logger.setLevel` method). Using the API, developers can program their system to dynamically choose a log level. However, it may prevent users from choosing an appropriate log level for their environment. The other one is a configuration file. Log4j has an automatic reconfiguration option to update its configuration every time interval (e.g. five seconds) during run-time. The update interval is too slow to record detailed logs for irregular behavior.

Liu et al. proposed a lightweight dynamic analysis framework named DOUBLETAKE to find buffer overflows, memory use-after-free errors, and memory leaks [9]. This framework monitors memory in a lightweight manner during a normal execution and records precise information related to memory errors when abnormal behavior occurs. The tool requires two program executions, while PADLA aims to record detailed logs in a single execution.

Ogami et al. proposed a three-dimensional cities visualization to show performance in real-time [12]. It visualizes the execution time for each method so that developers can recognize irregular behavior of a system. PADLA adopts the

Listing 1. Example log messages of JPetStore

```
1 15:03:46 [http-nio-8080-exec-108] TRACE net.
      sourceforge.stripes.controller.
      StripesFilter - Intercepting request to
      URL: /actions/Catalog.action
2 15:03:46 [http-nio-8080-exec-132] DEBUG net.
      sourceforge.stripes.controller.
      DispatcherHelper - Resolved event:
      viewCategory; will invoke:
      CatalogActionBean.viewCategory()
```

time information as a performance vector for phase detection and automatically adjusts the log level of a running system to record further details of irregular behavior.

Yao et al. proposed an adaptive logging approach for database systems [18], [19]. The approach combines two types of log messages to optimize the I/O cost of transaction logs while keeping the cost of data recovery from a failure. Although our method also uses two levels of log messages to reduce the I/O cost, our method focuses on debugging and failure diagnosis activities.

PADLA is dependent on appropriate log messages produced by a program. To enable developers to write a log statement properly, Yuan et al. proposed a method to detect inappropriate log levels based on source code version history [21]. Li et al. proposed an automated approach to help developers to determine the most appropriate log level when adding a logging statement [8]. Yuan et al. also proposed a tool named LogEnhancer that modifies each log statement to collect additional information (e.g. actual values of variables) to ease failure diagnosis [22].

PADLA is also dependent on a phase detection algorithm. PADLA cannot recognize irregular events if they do not affect performance vectors. Although existing behavior-based phase detection techniques [13], [16] are promising to detect other kinds of irregular events, they are defined as offline analysis performed after a program execution. Investigating an online phase detection algorithm for recognizing various irregular events is our future work.

V. Conclusion

This paper presents PADLA, a tool for adapting the log level of a running system dynamically. Using the phase detection algorithm, PADLA monitors behavior of the system and adjust the log level that is appropriate for the situation. In the case study, PADLA successfully recorded a detailed log for performance analysis of a server system under high load while suppressing the amount of log data and performance overhead.

PADLA has two parameters: the length of a time interval and the size of a buffer. In future work, we are planning to investigate effective parameter configuration. We also would like to investigate online phase detection techniques to effectively recognize various kinds of irregular events.

Acknowledgments

This work has been supported by JSPS KAKENHI Nos. JP18H03221 and JP18H04094.

References

[1] "Jpetstore demo 6," http://www.mybatis.org/jpetstore-6/.
[2] Apache Software Foundation, "Apache jmeter," https://jmeter.apache.org/.
[3] ——, "Apache log4j 2," http://logging.apache.org/log4j.
[4] ——, "Apache tomcat," http://tomcat.apache.org/.
[5] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.
[6] S. He, Q. Lin, J.-G. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 60–70.
[7] S. Kabinna, W. Shang, C. Bezemer, and A. E. Hassan, "Examining the stability of logging statements," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 326–337.
[8] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.
[9] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 911–922.
[10] J. Lou, Q. Fu, J. Li, and Y. Wang, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 149–158.
[11] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, 2010, pp. 24–24.
[12] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto, "Using high-rising cities to visualize performance in real-time," in *Proceedings of the 2017 IEEE Working Conference on Software Visualization*, 2017, pp. 33–42.
[13] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah, "Exploiting text mining techniques in the analysis of execution traces," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, 2011, pp. 223–232.
[14] S. P. Reiss, "Dynamic detection and visualization of software phases," in *Proceedings of the Third International Workshop on Dynamic Analysis*, 2005, pp. 1–6.
[15] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.
[16] Y. Watanabe, T. Ishio, and K. Inoue, "Feature-level phase detection for execution trace using object cache," in *Proceedings of the 2008 International Workshop on Dynamic Analysis held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[17] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 117–132.
[18] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging for distributed in-memory databases," *CoRR*, vol. abs/1503.03653, 2015.
[19] ——, "Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1119–1134.
[20] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," *SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 143–154, 2010.
[21] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 102–112.
[22] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 3–14, 2011.