Master Thesis

Title

Cost-Effective Detailed Logging Using Phase-Aware Dynamic Log Level Adaption

Supervisor
Professer Katsuro Inoue

Author
Tsuyoshi Mizouchi

2020/2/5

Software Engineering Laboratory Department of Computer Science
Graduate School of Information Science and Technology, Osaka University

令和元年度 Master Thesis

Cost-Effective Detailed Logging Using Phase-Aware Dynamic Log Level Adaption

Tsuyoshi Mizouchi

Abstract

Logging is an important feature of a software system useful for recording its run-time information. Although detailed logs are helpful to identify the cause of a failure in a program execution, constantly recording them in a long-running system is challenging because of performance overhead and storage cost. To solve the problem, we propose a method named PADLA that dynamically adjusts the log level of a running system so the system can record irregular events such as performance anomalies in detail, while recording regular events concisely. It employs an online phase detection method to recognize irregular events, which monitors run-time performance and learns regular execution phases of a program. If this method recognizes an irregular behavior excluding the regular phases, it automatically changes the log level threshold of a system to record its behavior in detail. PADLA aims at cost-effective logging that suppresses the entire log file size while keeping necessary information. We evaluated the logging quality of PADLA by using a benchmark tool. In our case study, PADLA successfully recorded information useful for debugging a real-case bug. The case study also showed that PADLA can greatly reduce the log file size from detailed logging while keeping much more information than ordinary logging.

Keywords

Dynamic Analysis
Log Level
Phase Detection

Contents

1 Introduction

Logging is a common practice used for recording run-time information of program execution [12]. 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 [3, 8, 14], program comprehension [11], 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 [4]. 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 [5]. Such a high overhead might cause adverse effects such as service delays.

To address the trade-off, popular logging libraries such as Apache Log4j[1] provide log levels to control the amount of log messages recorded on disk. Developers assign a log level to each logging statement in a system, then a logging library filters log messages by comparing the log levels with a threshold specified by the program user. In the case of Log4j, six log levels are available by default: *fatal*, *error*, *warn*, *info*, *debug*, and *trace*. These levels represent the messages degree of importance. The *error* level represents error messages, the *info* level represents important but normal events, and the *trace* level represents detailed information for tracing a program execution, respectively [16].

The log levels are easy to use but ineffective at analyzing 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 log messages, reproducing those behaviors is difficult [15]. For example, in our case study, we could find the cause of a bug in a server system only from *debug* and *trace* level log messages. For the case study, we use Bug 59813[2] of Apache Tomcat 8.5.3. This bug causes an infinite loop when the server system starts up. Two library files of the server system refer to each other in the Classpath causing an infinite loop. Log messages of these behavior only exist in *debug* and *trace* level, so we cannot know the cause of the infinite loop from *info* level log messages. This case is an example of *info* level messages being insufficient for debugging. On the other hand, logging the entire execution of a system with the *trace* level is unfeasible due to the huge amount of log messages and performance overhead. We conducted some load tests against a web application on Tomcat. The log file size was 12.5 MB when the log level threshold was set to *info*, however, the log file size was 1,525 MB on *trace*. The execution time of

[1]`http://logging.apache.org/log4j`

[2]`https://bz.apache.org/bugzilla/show_bug.cgi?id=59813`

the threshold *trace* is 4.84 times greater than that of the threshold *info*. The execution time of these load tests was about several tens of seconds, so we can infer that in large-scale and long-running system, the log file size and performance overhead of the threshold *trace* will increase much more.

To solve the problem, we propose a cost-effective detailed logging method that dynamically adjusts the log level threshold of a running system. The method employs an online phase detection method that divides a program's execution into a sequence of phases according to their dynamic properties or traits [10]. PADLA monitors run-time performance metrics of a system and learns regular phases of the system, then it recognizes an irregular phase like a performance anomaly and automatically changes the log level threshold of the running system to record detailed log messages only while the irregular phase is happening. PADLA aims at cost-effective logging that suppresses the entire log file size while keeping detailed log messages of the irregular behavior. The implementation of PADLA is an extension of Apache Log4j2.

This paper also proposes two different phase detection methods (time-based method and number-based method). Time-based method uses the execution time of each method to detect phases. This method attaches importance to detect performance anomaly (e.g. method execution time too long). On the other hand, number-based method uses the number of method call events. The number-based method aims at "stabler" phase detection than time-based method. The execution time of each method can fluctuate even with the same execution scenario because the program execution can be affected from memory status, other processes running on the same machine and so on. For this reason, the result of PADLA with the time-based method can be unstable. The number-based method is proposed against this problem.

To evaluate PADLA, we have conducted some experiments using an open source benchmark tool, DaCapo Benchmarks[3]. As a result, PADLA successfully maintained detailed log messages of unknown behavior with a small log file size compared with the *trace* level logging. In addition, PADLA with time-based phase detection method succeeded to reduce the execution time. Moreover, we experimented PADLA with several parameter values to investigate how these parameters affect the performance of PADLA. By using the result of the experiment, we compared the characteristics of the two phase detection methods which affect the performance of PADLA. Finally we confirmed that we could collect useful debugging information from log messages output by PADLA by using a real-case bug.

The main contribution of this paper can be summarized as follows:

(1) We propose two different methods that recognize the execution phase of a program to detect

[3]`http://dacapobench.org/`

its irregular behavior.

(2) We propose a method that dynamically adapts the log level threshold of a running system according to its behavior detected from step (1).

(3) We evaluate the proposed method in a benchmark tool, and show its usefulness by comparing it against a fixed threshold.

(4) We investigate how the parameters of the proposed method affect the performance of the method itself by using the benchmark tool.

(5) We compare the characteristics of the two phase detection methods which affect the performance of PADLA.

(6) We show, by using a real-case bug, that dynamically adapting the log level threshold can collect useful information for debugging.

In the remainder of the paper, Section 2 presents related works and explains our background. Section 3 explains the tool in detail. Section 4 shows our evaluation of PADLA. Section 5 shows our case study demonstrating the effectiveness of the tool. Section 6 describes limitations. Section 7 describes the future work and concludes the paper.

The PADLA tool is publicly available on GitHub: `https://github.com/inoueke-n/PADLA`.

2 Background

2.1 Log-based anomaly detection

Log analysis is conducted when any problems occur during the execution of a program to diagnose their cause. However, it is often difficult to analyze the log manually to detect problematic portions given the vast amount of log [5]. To solve the problem, some studies focused on log-based anomaly detection [2, 5, 6, 17].

Min, et al. proposed an approach that detects irregular log sequences using deep learning [2]. At first, the approach learns "normal" log sequences by using a LSTM, then it detects log sequences that deviate from learned log sequences and labels them as "irregular". Lin, et al. proposed LogCluster that clusters log files according to weighted log messages the log files have [6]. LogCluster can reduce logs to be examined by checking whether the clusters are known or unknown by the knowledge base approach. Shilin, et al. proposed an approach that makes traditional log-based anomaly detection approach faster and more accurate [5]. This approach used information of system Key Performance Indicators to detect problems of a target system. In addition, they used cascading clustering to shorten the time consumed by clustering the log sequences. Zhang, et al. focused on the fact that log messages and sequences are unstable and proposed LogRobust an approach robust against that [17]. Traditional log-based anomaly detection approaches need to rebuild models when the log messages evolve (e.g. changed, added). LogRobust is robust against the evolved log messages because it converts the log messages to semantic vectors. These approaches can detect anomalies from logs, however, these can only collect log messages of a log level specified in advance. Like the case study sample of Section 1, sometimes it is not possible to collect useful information without *debug* and *trace* level. PADLA can collect detailed information of an irregular behavior by adapting the log level threshold. Thus, PADLA can not only detect anomalies but also collect detailed information about them.

2.2 Execution Phase

PADLA monitors the execution of a target system and recognizes execution phases to detect an irregular behavior of the target system. In this paper, the word "phase" represents a portion of the program that exhibits common behavior recognizable by the programmer [10].

Reiss proposed an approach that monitors the execution of a target system and obtains software metrics to detect execution phases [10]. This approach firstly converts software metrics (e.g. number of entries into methods, number of allocations of objects of class) of target system at each fixed time interval to a performance vector, and then, compares it with a previous one to determine whether the execution phase changes or not. PADLA applies this approach to its phase detection (time-based method). Time-based method recognizes the execution phase by monitoring the method execution time of a target system, and then, compares the phase with previously collected phase information to determine whether the current phase is known or unknown.

Watanabe, et al. proposed an approach that detects phase, or high-level behavioral units described in a use-case scenario by using an execution trace [13]. Their approach is based on the hypotheses that many objects are created to achieve a task and most of them are destroyed after their task. The approach loads an execution trace and keeps object information related to method call events into a LRU cache and interprets a sharp rise in the frequency of the cache update as a phase transition. It is a static approach and it aims to detect high-level behavioral units such as "login" and "show list of items". On the other hand, PADLA is a dynamic approach and it aims to detect more fine-grained phases (e.g. "initializing", "loading class files").

2.3 Detecting Irregular Behavior at Run-time

PADLA monitors the execution of a target system for online detection of irregular behavior. In existing research, some studies were conducted for detecting irregular behavior at run-time.

Liu, et al. proposed a framework named DOUBLETAKE that detects memory errors and helps to diagnose them [7]. DOUBLETAKE monitors the memory during the execution, and records detailed information of the memory state when an irregular behavior (e.g., buffer overflow, memory use-after-free error, memory leak) happens. Using these detailed information, DOUBLETAKE can reproduce the irregular behavior during debugging. This framework focuses on memory errors as irregular behavior while PADLA monitors the execution of each method and detects irregular behavior of a target system. In addition, DOUBLETAKE needs 2 executions (error detection and reproduction) in order to diagnose errors while PADLA only needs one to collect detailed log messages. Dean, et al. proposed an unsupervised anomaly detection approach for cloud environment [1]. This approach uses the self organized map method to detect performance irregularities without learning data labeled manually. PADLA can detect irregular behavior without learning data labeled manually and it can also collect detailed log messages of the irregularities by dynamically adapting the log level threshold. Ogami, et al. proposed a real time profiler named Heijo that visualizes run-time information of a target system to detect performance anomalies [9]. Heijo obtains the execution time of each method and makes a code city to visualize the performance of a target system online. To detect a performance anomaly using this tool, we have to watch the city while the target system is running. PADLA uses the information obtained by Heijo and detects irregular behavior automatically.

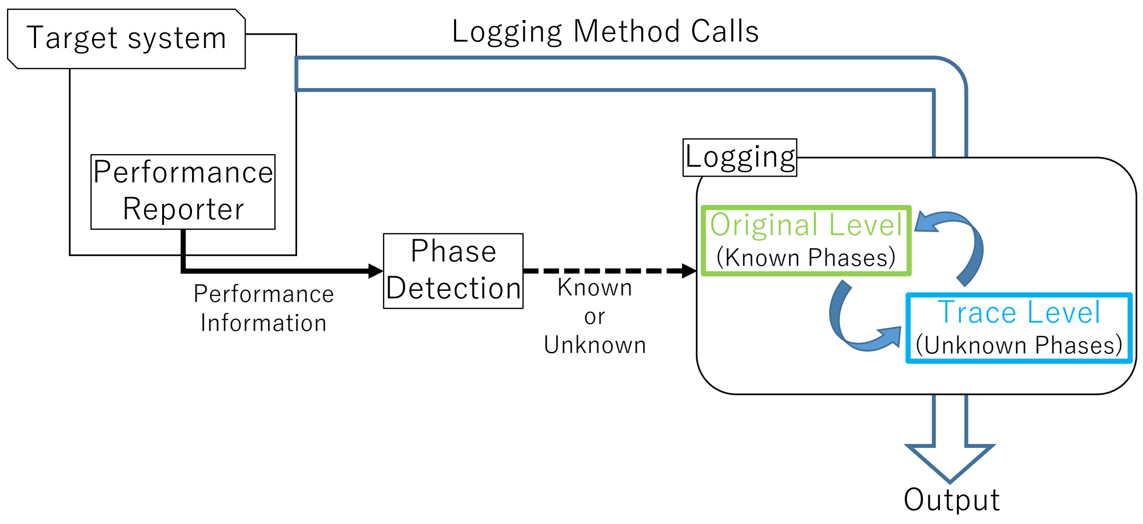


Figure 1: An overview of PADLA

3 PADLA

PADLA takes as input a log level for recording regular behavior, e.g. *info* level and takes also execution scenarios. If unavailable, regular behavior is learned on the fly. PADLA produces application logs using the specified level for regular events and the *trace* level for irregular events, respectively.

PADLA comprises two components: 1) Phase Detection and 2) Logging. Figure 1 shows an overview of these components. 1) The phase detection component monitors a program execution and divides it into a sequence of phases according to their performance characteristics, then classifies each phase as either known or unknown. 2) The logging component buffers recent log messages and filters them using the original level specified by a user for known phases and the *trace* level for unknown phases, respectively. The following subsections explain each component in detail.

The implementation of PADLA is an extension of Apache Log4j2 that dynamically adjusts the output log level of a running system according to its run-time behavior. PADLA keeps the logging interface of Log4j2 and can be activated it by adding PADLA's configuration to the *log4j2.xml* file along with arguments for the program execution. It does not need to modify the source code of a target system.

3.1 Phase Detection

PADLA has two different online phase detection methods: time-based and number-based. Both

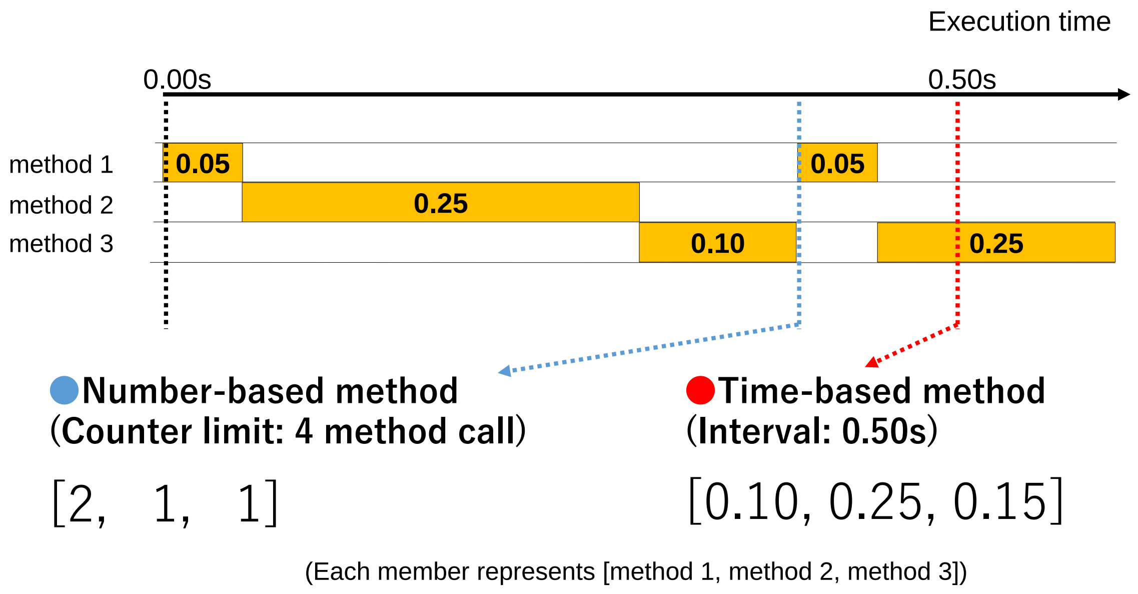

Execution time
0.00s
0.50s
method 1
method 2
method 3
0.05
0.25
0.10
0.05
0.25
●Number-based method (Counter limit: 4 method call)
[2, 1, 1]
●Time-based method (Interval: 0.50s)
[0.10, 0.25, 0.15]
(Each member represents [method 1, method 2, method 3])

Figure 2: How to make the performance vector in each phase detection method

methods translate an execution trace into a sequence of performance vectors, and then compare them to categorize known/unknown phases. Figure 2 summarizes how to create the performance vector in each phase detection method. The time-based method uses the method execution time in every fixed time interval as a performance vector. The number-based method uses the number of method executions in every certain number of method executions as a vector. The number of dimensions in the performance vector depends on the number of methods or method call events. For example, in the time-based method, the number is 2,074 in tomcat and 166,885 in kafka (tomcat and kafka are the target benchmark program of a benchmark tool we use in Section 4). In the number-based method, the number is 163,109 in tomcat and 287,135 in kafka. The time-based method attaches importance to detect performance anomaly. In case there are some performance anomalies such as freeze for a moment and when such anomaly occurs, the execution time of some method changes from the regular behavior. For this reason, monitoring the method execution time can be expected to detect such performance anomaly. The number-based method attaches importance to stabler phase detection than the time-based method. The execution time of each method can fluctuate even with the same execution scenario because of the execution environment which the program is running on (e.g. memory status, other processes). Therefore, the phase detection result of the time-based method might be unstable. On the other hand, the order and the number of method execution is relatively stable with the same execution scenario ("relatively stable" means that the order and the number may fluctuate a little when the execution scenario has some random elements such as random value or random access). For this reason, the number-based method is expected to be stabler than the time-based method.

3.1.1 Time-based Method

The time-based method employs an online phase detection method proposed by Reiss [10]. This method firstly divides a program execution into fixed time intervals. Then it translates the t-th interval into a performance vector v_t. Two intervals (e.g. s-th interval and t-th interval) belong to the same phase if the cosine similarity between their vectors v_s and v_t is greater than a threshold θ. Otherwise, the intervals belong to different phases.

The time-based method employs a performance vector used in an existing performance profiling tool [9]. Each element in a vector represents the actual execution time spent by a corresponding method in the interval. To observe performance vectors, the time-based method uses dynamic bytecode instrumentation to inject a performance reporter into a target application. The injected code reports the time spent for each executed method. The time-based method obtains a performance vector every fixed time interval I.

The time-based method 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 or unknown phase if there exists or not a vector $v \in V_k$ that 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 an execution of the program that covers regular behavior. Once a V_k is constructed, it can be available in subsequent executions as an execution scenario. If the scenario is unavailable, PADLA starts with $V_k = \phi$.

The component updates V_k to learn unknown phases as known phase if they repeatedly occur during an execution. 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 its performance. The log level change of PADLA also may affect performance characteristics. If the component records detailed log messages of repeated occurrences of an unknown phase, the log file size and performance overhead will be huge. So it records detailed log messages of only the first few intervals of a long-duration phase for analysis. In the current implementation, the method 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 known phase for the system.

3.1.2 Number-based Method

The number-based method is similar to the time-based method. The different point is how to make the performance vector. The number-based method has a method call counter and it counts the number of method call events. When the counter reaches its limit L, a performance vector is made with the counter information. Each member of the performance vector corresponds to each method, and the value of the member is the number of method call events of the corresponding method.

The performance vector is classified as known or unknown same as the time-based method. After that, the counter is reset and starts again. The classification method of the number-based method is the same with the time-based method. It keeps past performance vectors as known vectors and compare latest performance vector with the known vectors using cosine similarity.

3.2 Logging

The logging component writes log messages to the disk 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 on memory using the *trace* level. During known phases, the logging component filters the detailed messages output but keeps them in memory. If an unknown phase is detected, the buffered log messages up to the previous interval are written to log files, and then the component starts to record log messages using the *trace* level. The buffered messages provide detailed behavior of the last interval before the unknown phase detection allowing developers to investigate what actually happened in the unknown phase.

It should be noted that the overhead of the buffering is small because an application using Log4j2 always produces log messages irrelevant to a run-time log level. An existing logger in Log4j2 filters the generated log messages for each logging method call. Our logging component delays the filtering step until a phase classification.

3.3 Usage

To apply PADLA, we need two execution files, *Agent.jar* and *log4j-core-extended.jar*. All files needed to apply PADLA are located on the GitHub repository written in Section 1. The additional disk space is at most 3MB. *Agent.jar* represents the phase detection component of PADLA. This component comprises a Performance Reporter function which collects information necessary for the Phase Detection and a Phase Detection function which detects execution phases using the result from the Performance Reporter. The phase detection component is implemented as a Java

Agent so it has to be installed into a target system when the system starts. *log4j-core-extended.jar* represents the logging component and it is implemented as an extension of a Log4j2 library file. It can be used by replacing the original *log4j-core.jar* file and editing the original configuration file of log4j2. When PADLA is applied, *Agent.jar* sends results of the phase detection to *log4j-core-extended.jar* via socket. When receiving the result, *log4j-core-extended.jar* filters log messages using the original log level threshold if the current phase is known. Instead, if the current phase is unknown, it turns off the filter. Using this method, we can collect log messages of a level higher than the original one in known phases. Therefore, in unknown phases, all log messages are collected.

PADLA has two parameters and a user can adjust PADLA's performance by changing the parameters. The two parameters will be changed with the phase detection method which a user chooses. If the time-based method is chosen, the parameters are I and θ. The parameter I represents the length of the fixed time interval. This parameter is related to the frequency of the phase detection step and grain size of a phase, so we can infer that I affects results of the phase detection. In other words, it is related to the amount of log messages created in a single interval. The parameter θ represents a phase similarity threshold that is used to judge the phase that is known or unknown. When θ is changed, the amount of intervals judged as unknown phase changes. If the number-based method is chosen as the phase detection method, the parameters are L and θ. The parameter L represents the upper limit of a method call events. Phase detection is executed every time the counter reaches L. This parameter is also related to the frequency of the phase detection step and the grain size of a phase same as I. The parameter θ is the same thing as in the time-based method.

4 Evaluation

We want to investigate the following research questions to evaluate the effectiveness of PADLA.

RQ.1 What is the logging performance of PADLA?

RQ.2 How do parameter values affect the performance of PADLA?

RQ.3 What characteristics do the two phase detection methods give PADLA?

To answer the three RQs, We apply PADLA to benchmark programs and evaluate the logging results in terms of the content quality and run-time overhead.

We use four logging configurations listed below.

- **none.** This configuration executes a program without logging. The resultant log includes nothing.
- **trace.** This configuration executes a program using the *trace* log level threshold. The resultant log records all the log messages generated in a program execution.
- **info.** This configuration executes a program using the *info* log level threshold. The resultant log excludes detailed messages of *debug* and *trace* levels.
- **PADLA.** This configuration dynamically adapt two log level thresholds (*info* and *trace*) using our phase detection.

The *none* configuration is used to investigate the overhead of log the output process itself. The *trace* log is used as baseline of the log contents. To evaluate the quality of contents in the *info* and PADLA traces, we define a *Message Coverage* $M(l)$ metric against the log content of a log l as follows:

$$M(l) = \frac{|C_{trace} \cap C_l|}{|C_{trace}|}$$

The log content C_l represents a set of source code locations that generated messages in the log l. This means that for each log message we record the caller position only once in C_l. $M(C_l)$ is calculated based on *trace* level log (C_{trace}) because C_{trace} contains log messages of every log levels. We employ a set of log messages that called from a unique position of its source code as the log content C_l because PADLA aims to record the unknown behavior. As we describe in Section 3, PADLA treats an unknown phase as a known phase after its third appearance (if it has already appeared in two consecutive intervals). In other words, it records detailed log messages at the first few occurrence having similar behavior, but after these, nothing is recorded anymore in order to save disk space and avoid performance overhead. According to that, we can say PADLA detects the unknown behavior and maintains the contents of its log if and only if PADLA can record log messages that called from an unique position of the source code.

The execution time of the benchmark is the benchmark resulting from every target program. It is not the time from the beginning to the end of the benchmark program. The log file size is the file size of a file output by log4j2. The PADLA tool outputs two log files, a log file created by the original log4j2 function and a log file for recording memory buffered log messages. These two files are combined and treated as one log file.

Our target benchmark programs are tomcat and kafka in the DaCapo Benchmark Suite[4] because they implement logging features. We replace their underlying logging library with the PADLA tool. The tomcat benchmark provides three execution scenarios: small, default, and large. We choose the large scenario because we expect PADLA to be used on server programs so we need a long time execution. For kafka, we use the only available default scenario.

The evaluation is conducted with a Windows 10 Pro machine running on a Xeon E5-1650v3 processor, 32 GB RAM, and an SSD. We execute the benchmark 10 times for each target program, and then, calculate an average. Calculating an average is also needed for *none*, *info* and *trace* because the execution time and the log file size of the benchmark are not constant between each execution. DaCapo Benchmarks uses some random numbers in its benchmarks, so the numbers of some steps slightly fluctuates between each execution. On the other hand, C_{trace} is always the same meaning that $M(l)$ is immutable in this evaluation.

Recall that the two phase detection methods have their own parameters. The time-based method has the interval for the phase detection I and the phase similarity threshold θ. The number-based method has the limit of method call events counter L and the phase similarity threshold θ. For RQ1, we use the default values of our implementation: $I = 0.5$s, $L = 1,000,000$ and $\theta = 0.95$ (the default value of θ is the same between the two phase detection methods). For RQ2, we tried varying values for a parameter while maintaining the other one fixed. When time-based method is applied, we choose [0.1s, 0.5s, 1.0s] for I when θ is fixed, and values in increments of 0.01 from 0.90 to 0.99 for θ when I is fixed. When the number-based method is applied, we choose [500,000, 1,000,000, 1,500,000] for L when θ is fixed, and values in increments of 0.01 from 0.90 to 0.99 for θ when L is fixed.

Table 1: The result of RQ1. (PADLA_t represents PADLA with time-based method. PADLA_n represents PADLA with number-based method.)

Target	Configuration	File size (KB)	(division by *trace* (%))	M(l) (%)	Execution time (s)	(division by *trace* (%))
tomcat	*none*	0	(0.00)	0.00	118.49	(30.49)
	info	20	(0.01)	6.25	114.65	(29.50)
	PADLA_t ($I = 0.5s$,$\theta = 0.95$)	39,866	(0.84)	90.07	255.90	(65.85)
	PADLA_n ($L = 1,000,000$,$\theta = 0.95$)	31,465	(0.66)	86.47	780.57	(200.88)
	trace	4,752,124	(100.00)	100.00	388.58	(100.00)
kafka	*none*	0	(0.00)	0.00	6.34	(53.37)
	info	166	(0.01)	27.46	9.38	(78.95)
	PADLA_t ($I = 0.5s$,$\theta = 0.95$)	13,398	(26.11)	99.05	11.27	(94.86)
	PADLA_n ($L = 1,000,000$,$\theta = 0.95$)	11,326	(22.07)	97.29	10.19	(85.77)
	trace	51,324	(100.00)	100.00	11.88	(100.00)

4.1 RQ1: What is the logging quality of PADLA?

Table 1 shows the result of the evaluation for RQ1. In the tomcat result, the log file size of PADLA with time-based method (hereinafter referred to as "PADLA_t") is 0.84% of *trace* and 26.11% in the kafka result. $M(l)$ values of PADLA_t are 90.07% in the tomcat result and 99.05% in the kafka result. Likewise, the log file size of PADLA with number-based method (hereinafter referred to as "PADLA_n") is 0.66% of *trace* and 22.07% in the kafka result. $M(l)$ values of PADLA_n are 86.47% in the tomcat result and 97.29% in the kafka result while $M(l)$ values of *info* are 6.25% and 27.46% for tomcat and kafka respectively. $M(l)$ values of *trace* is 100% because $M(l)$ values are calculated based on the log of *trace*. Besides, the execution time of PADLA_t is 65.85% shorter than *trace* in the tomcat result and 94.86% shorter in the kafka result. Compared to *none*, we can infer that the additional log output process of *trace* takes 270.09s (the execution time difference between *none* and *trace*) in the tomcat result and 5.54s in the kafka result while PADLA_t takes 137.41s in the tomcat result and 4.93s in the kafka result. It means that, compared to *trace*, PADLA_t can reduce additional log output process to 50.88% in the tomcat result and 88.99% in the kafka result.

Indeed the log file size and the execution time of *info* are smaller than that of PADLA, however, $M(l)$ value of PADLA_t and PADLA_n greatly exceed that one of *info* in both tomcat and kafka. The results indicate that PADLA can suppress the log file size (execution time can be suppress only by PADLA_n) compared to *trace* while keeping a higher $M(l)$ value than *info*. In other words,

[4] `dacapo-evaluation-git+8b7a2dc-java8.jar`

Table 2: Parameter I: Interval for the phase detection (PADLA_t)

Target	Configuration ($\theta = 0.95$)	Execution time (s)	File size (KB)	M(l) (%)
tomcat	$I = 0.10s$	256.29	37,857	85.00
	$I = 0.50s$	255.90	39,866	90.07
	$I = 1.00s$	258.73	33,667	85.63
kafka	$I = 0.10s$	11.44	2,950	97.25
	$I = 0.50s$	11.27	13,398	99.05
	$I = 1.00s$	11.47	16,486	99.54

Table 3: Parameter L: Limit of method call events counter (PADLA_n)

Target	Configuration ($\theta = 0.95$)	Execution time (s)	File size (KB)	M(l) (%)
tomcat	L = 500,000	841.42	30,950	88.31
	L = 1,000,000	780.57	31,465	86.47
	L = 1,500,000	755.54	22,791	82.35
kafka	L = 500,000	14.01	12,372	97.36
	L = 1,000,000	10.19	11,326	97.29
	L = 1,500,000	9.47	15,493	95.56

PADLA records the first occurrence of an exceptional behavior in the *debug* and *trace* level saving disk space.

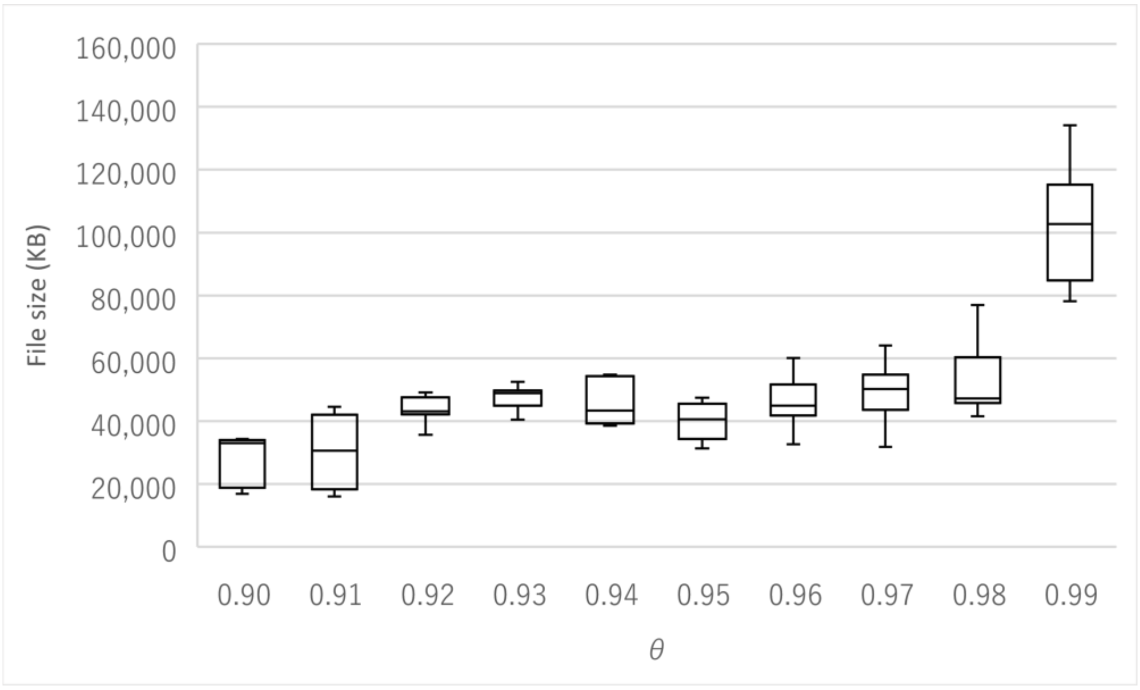
4.2 RQ2: How do parameter values affect the performance of PADLA?

Table 2 shows the PADLA_t's results when changing I in tomcat and kafka. In both results, there are little differences between the execution time for each value of I. When I decreases, the number of the phase detection step in a fixed interval increases. For example, when I halves, the number of the phase detection step doubles. So this result indicates that changing the number of the phase detection step caused by changing I does not affect the execution time by a great extent. The log file size and $M(l)$, in the result of tomcat, increase when I is between $I = 0.10$s and 0.50s, but decrease when between $I = 0.50$s and 1.00s. On the other hand, in the result of kafka, these tend to only increase with increasing I. PADLA changes the log level threshold to *trace* when it detects an unknown phase and changes the log level threshold to original level when it detects a known phase. When L increases, the time to change the log level threshold to original level increases, so the number of log messages to be output also increases. For this reason, in the result of kafka, the log file size increases when I increases making $M(l)$ larger. However, because the granularity of phase detection changes when I changes, the result of phase detection of tomcat becomes affected. When the execution time is relatively long, the number of detected unknown phases affects the log file size rather than the number of log messages output in one interval. So we can conclude that at $I = 0.50s$ in tomcat, the log file size and $M(l)$ are larger than their respective results with $I = 1.00s$ because $I = 0.50s$ detects more unknown phases than $I = 1.00s$.

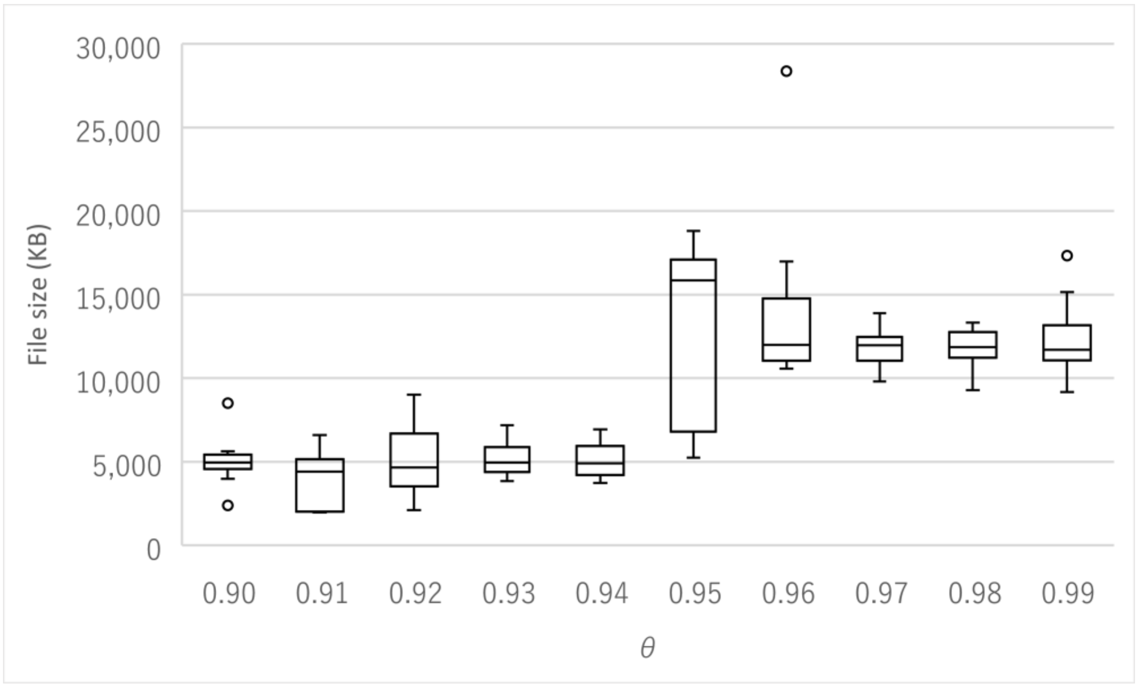
Table 3 shows the PADLA_n's results when changing L in tomcat and kafka. In both results, the execution time decreases when L increases. As with the parameter I, the number of phase detection steps in fixed interval increases when L increases. However, unlike I, the execution time increases with the number of phase detection steps. The reason for this result can be inferred as the difference between the number of phase detection steps. For example, the phase detection step was executed 8,585 times in an execution of tomcat with PADLA_n ($\theta = 0.95$, $L = 1,000,000$). On the other hand, it was executed 571 times in an execution of tomcat with PADLA_t ($\theta = 0.95$, $L = 1,000,000$). When the phase detection step of PADLA_t becomes 5 times ($I = 0.5s \rightarrow I = 0.1s$), the number is smaller than that of PADLA_n. However, when the phase detection step of PADLA_n becomes double ($L = 1,000,000 \rightarrow L = 500,000$), the number is greatly larger than that of PADLA_t. For this reason, increasing the phase detection steps increase the execution time of PADLA_n while it does not affect too much the execution time of PADLA_t. The log file sizes do not increase or decrease along with changing the value of L in both tomcat and kafka results. The parameter I controls the time length of the interval between phase detection steps, so the bigger I causes the longer interval. If the interval is an unknown phase, the amount of detailed logs output in the interval will be huge. L is the limit of method call events counter and the number of method call events in a fixed time interval is not fixed. Then, if an unknown phase is detected in an interval which has many method call events, the time during which detailed logs are output will be short. Conversely, if the unknown phase is detected in an interval which has a small amount of method call events, the time during which detailed logs are output will be long. Consequently, the log file size does not change monotonically along with changing of L, so we can infer that it depends on the target programs. In both tomcat and kafka results, the smaller value of L, the larger $M(l)$ becomes. However, we cannot conclude that smaller values of L causes larger values of $M(l)$ because the effect of changing the grain size of a phase caused by changing L depends on the target programs.

Table 4: Parameter θ: Threshold for phase detection (PADLA_t)

Target	Configuration ($I = 0.5s$)	Execution time (s)	File size (KB)	M(l) (%)
tomcat	$\theta = 0.90$	257.18	28,709	81.43
	$\theta = 0.91$	257.72	29,803	81.80
	$\theta = 0.92$	255.65	43,930	83.60
	$\theta = 0.93$	257.03	47,484	83.68
	$\theta = 0.94$	257.31	45,493	89.63
	$\theta = 0.95$	255.90	39,866	90.07
	$\theta = 0.96$	256.85	46,187	90.88
	$\theta = 0.97$	256.07	49,534	90.55
	$\theta = 0.98$	258.55	53,215	88.38
	$\theta = 0.99$	257.20	102,127	91.91
kafka	$\theta = 0.90$	11.25	5,051	98.73
	$\theta = 0.91$	11.36	4,096	97.32
	$\theta = 0.92$	11.18	5,033	98.70
	$\theta = 0.93$	11.28	5,086	98.94
	$\theta = 0.94$	11.26	5,053	98.84
	$\theta = 0.95$	11.27	13,398	99.05
	$\theta = 0.96$	11.09	13,973	99.30
	$\theta = 0.97$	11.37	11,814	99.05
	$\theta = 0.98$	11.28	11,854	98.94
	$\theta = 0.99$	11.37	12,292	99.26



(a) tomcat



(b) kafka

Figure 3: Boxplots of the resulting log file sizes (each file size is the result of 10 executions)

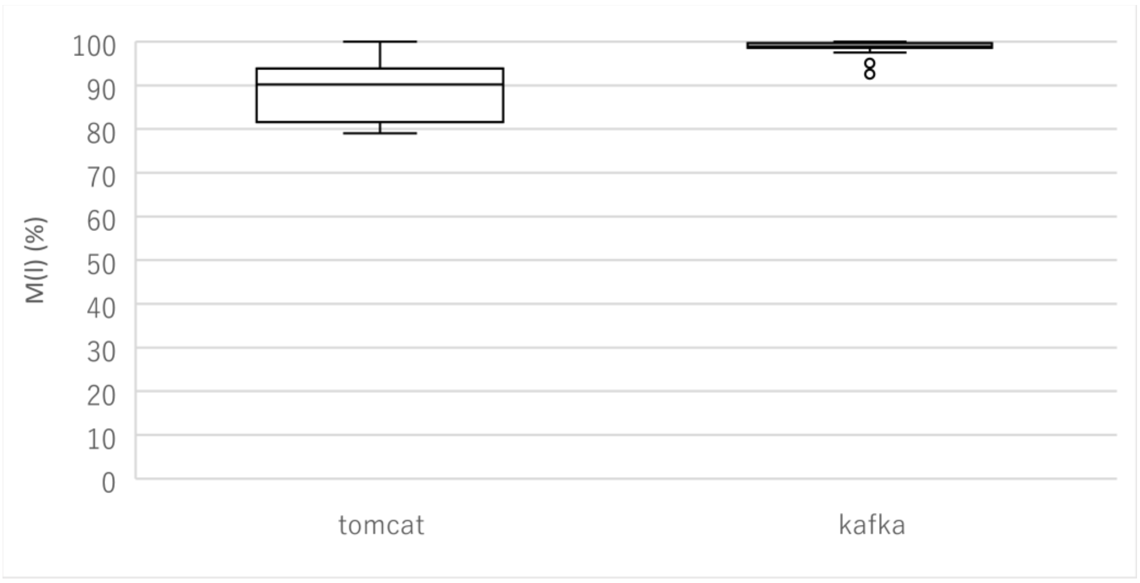


Figure 4: Boxplot of $M(l)$ values for all executions of the evaluation of θ

Table 5: Parameter θ: Threshold for phase detection (PADLA_n)

Target	Configuration ($L = 1,000,000$)	Execution time (s)	File size (KB)	M(l) (%)
tomcat	$\theta = 0.90$	769.57	26,212	86.40
	$\theta = 0.91$	773.15	25,104	85.74
	$\theta = 0.92$	774.77	25,778	85.74
	$\theta = 0.93$	774.33	27,208	85.85
	$\theta = 0.94$	773.77	29,083	86.07
	$\theta = 0.95$	780.57	31,465	86.47
	$\theta = 0.96$	786.41	30,008	86.43
	$\theta = 0.97$	783.82	34,212	88.27
	$\theta = 0.98$	783.33	40,145	87.21
	$\theta = 0.99$	780.08	49,389	89.67
kafka	$\theta = 0.90$	9.53	10,951	97.64
	$\theta = 0.91$	9.77	10,476	97.15
	$\theta = 0.92$	9.49	11,331	96.90
	$\theta = 0.93$	9.71	11,431	97.43
	$\theta = 0.94$	10.39	10,935	97.39
	$\theta = 0.95$	10.19	11,326	97.29
	$\theta = 0.96$	10.47	12,777	97.22
	$\theta = 0.97$	10.56	13,195	97.29
	$\theta = 0.98$	10.57	16,659	97.29
	$\theta = 0.99$	10.91	23,781	97.29

Table 4 shows the results of changing θ in tomcat and kafka with PADLA_t. There are little differences between the execution time for each value of θ . Since the number of phases increases as the value of θ increases, the log file size is expected to increase by detecting more types of phases. However, although the log file size of tomcat increases as θ increases, the log file size of kafka does not increase monotonically with increasing θ. Figure 3 shows boxplots of the log file sizes for each value of θ. Figure 3 indicates that the log file size is unstable in both tomcat and kafka. The result of the phase detection is greatly affected by the machine status, so the result is somewhat unstable. For this reason, as a result of changing the number of unknown phases with each benchmark execution, the log file size changes greatly with each benchmark execution.

Figure 4 shows a boxplot of $M(l)$ values for every evaluation of θ. In both tomcat and kafka, $M(l)$ has an high value in all executions. Although the log file size is unstable, the result of Figure 4 indicates that PADLA can achieve high $M(l)$.

Table 5 shows the results of changing θ in tomcat and kafka with PADLA_n. The execution time increases as θ increases (although not linearly). The result is not the same as the result of PADLA_t because the number of dimensions of the performance vector is different. PADLA calculates the cosine similarity between a performance vector and known vectors when it conducts the phase detection, so if the number of dimensions of the vector is huge, the calculation time of the cosine similarity will be long. For example, in the tomcat execution, the dimension of the performance vector of PADLA_t is 2,047 and that of PADLA_n is 163,109. When θ increases, the number of phases increase (e.g. 6 phase in $\theta = 0.90$ and 9 phase in $\theta = 0.99$ with tomcat and PADLA_n) so the number of known vectors for which cosine similarity should be calculated also increases. In case of PADLA_t, the number of dimensions of the performance vector is relatively small so the increase in calculation of the cosine similarity is also low, as well as the increase of execution time. However, the dimensions are huge in PADLA_n so the increase of calculation cost is huge and it can not be ignored at execution time. The log file size increases as θ increases because the large θ makes many unknown phases and many unknown phases implies many detailed logs. In the tomcat result, the $M(l)$ value increases as θ increases while it hardly changes in the kafka result. We can infer that new unknown phases which are detected when θ increases do not contain the first occurrence of log events in kafka.

From the above, the effects of the parameters can be summarized as follows. I and θ of PADLA_t do not affect the execution time of a target system much. The optimal value of I depends on the target program. The log file size increases as θ of PADLA_t increases, but not monotonically. The effect of changing θ of PADLA_t on the log file size and $M(l)$ is not clear because the output log file size is not stable. However, we can conclude that PADLA achieves high $M(l)$ values in both tomcat and kafka. In the current implementation, when L becomes small, the execution time increases and the $M(l)$ value gets better. We can not determine the correlation between the log file size and L in this evaluation. With PADLA_n, the log file size and the execution time increase as θ increases. The $M(l)$ value also can increase but it depends on the target program.

4.3 What characteristics do the two phase detection methods give PADLA?

For RQ3, we compare PADLA_t and PADLA_n by using the following three points of view.

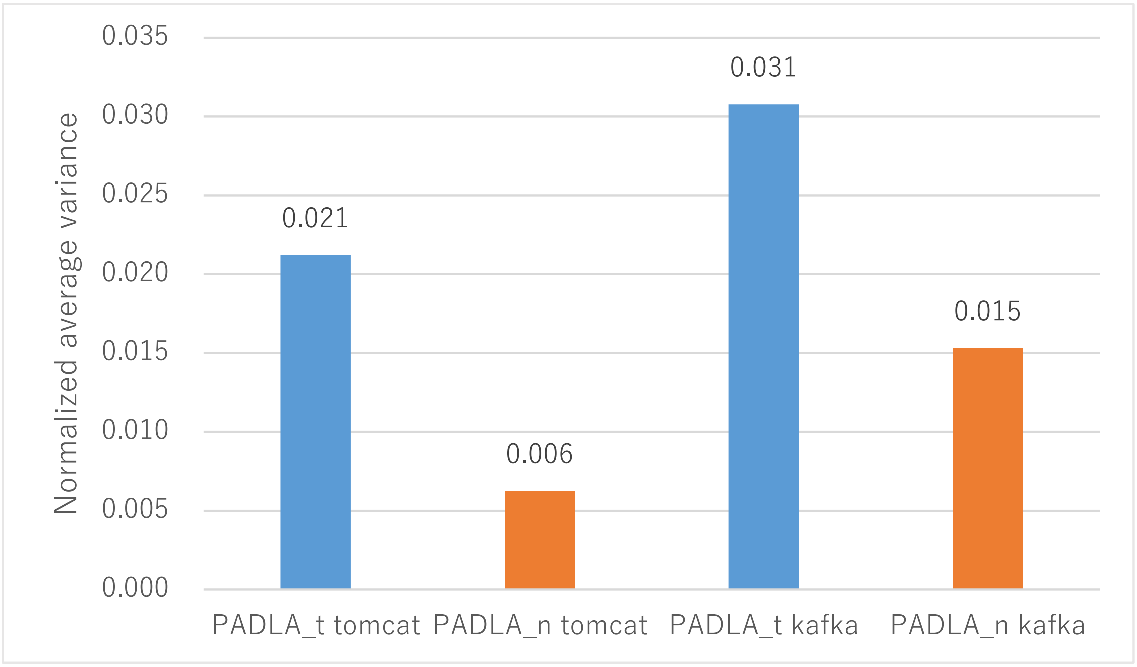


Figure 5: Normalized average variance value of the log file size

- $M(l)$
- Execution time
- Stability

The stability represents how stable the results produced by PADLA are with each execution in the same scenario and environment. Note that the log file sizes of PADLA_t and PADLA_n are not compared because, as we mentioned above, the log file size of PADLA_t is not stable in every execution so it is difficult to compare. We use the data (file size, execution time, $M(l)$ value) collected for RQ2.

As Table 2 to Table 5 show, the maximum $M(l)$ value of PADLA_t is 91.91% in tomcat and 99.30% in kafka while PADLA_n achieves 89.67% in tomcat and 97.43% in kafka. So it can be clear that PADLA_t can produce better $M(l)$ value than PADLA_n. For the comparison of the execution time, we use parameter sets that have similar $M(l)$ value because we need to compare the situation where we can get similar result. From above, the $M(l)$ value of PADLA_t tends to be higher than that of PADLA_n. So we use the parameter set which has the maximum $M(l)$ value in PADLA_n and the parameter set which has the $M(l)$ value nearest (the absolute value of the difference is the smallest) to the PADLA_n's maximum in PADLA_t. In the tomcat result, we choose [$L = 1,000,000$, $\theta = 0.99$] ($M(l) = 89.67$) for PADLA_n and [$I = 0.5s$, $\theta = 0.94$]

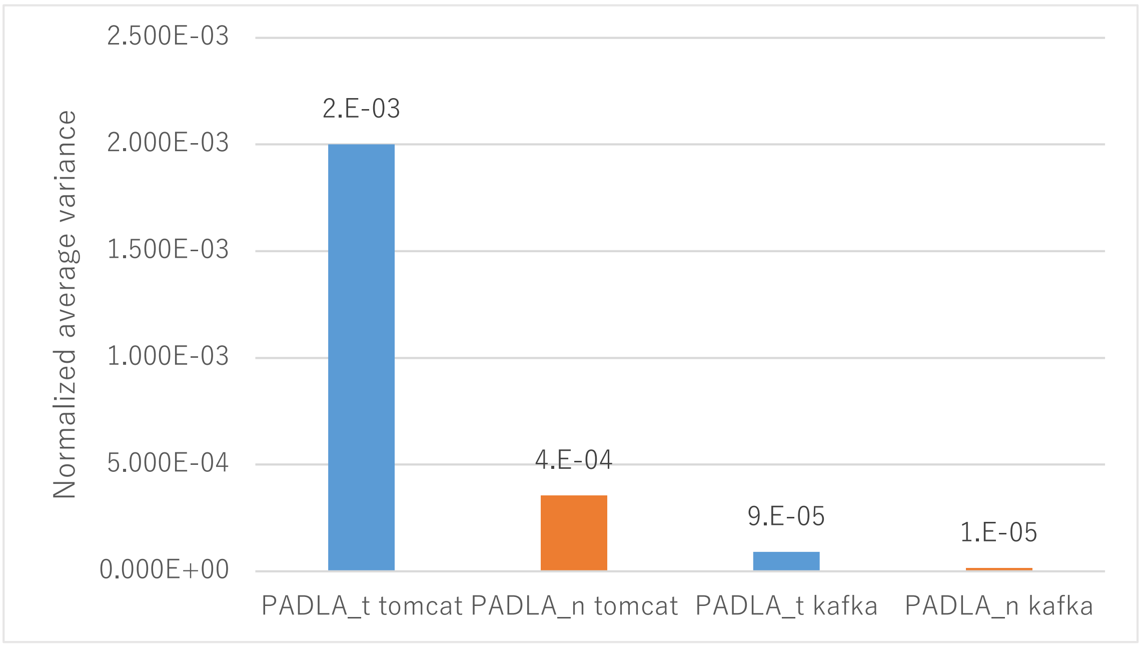


Figure 6: Normalized average variance value of the $M(l)$ value

($M(l) = 89.63$) for PADLA_t. In the kafka result, we choose [$L = 1{,}000{,}000$, $\theta = 0.90$] ($M(l) = 97.64$) for PADLA_n and [$I = 0.5s$, $\theta = 0.91$] ($M(l) = 97.32$) for PADLA_t. In the tomcat result, the execution time of PADLA_n is 2.83 times longer than PADLA_t. On the other hand, in the kafka result, the execution time of PADLA_t is 1.19 times longer than PADLA_n. The reason for this can not be concluded only from the data collected from this evaluation. However, we can infer that it is caused by the differences between implementation and target programs.

For the comparison of the stability, we calculate the variance values of the log file size collected from PADLA_t and PADLA_n. As we explained in RQ2's result, the log file size of PADLA_t is not stable in every execution even if it uses the same scenario and environment. So, we evaluate the stability of PADLA_t and PADLA_n by their variance value of the log file size. We collected the log file size 10 times in each parameter θ for PADLA_t and PADLA_n. We calculate the variance value of the log file size in each θ and then, we compare the average of the values between the two methods. Note that the log file size is normalized by using the maximum value in each θ. Figure 5 shows the normalized average variance value of the log file size in each method and target program. In both tomcat and kafka, the normalized average variance value of PADLA_n is smaller than that of PADLA_t. In the tomcat result, the value of PADLA_n is about one third of that of PADLA_t and in the kafka result, the value is about half of that of PADLA_t. Figure 6 shows that the normalized average variance value of the $M(l)$ value in each method and target program.

The value is calculated from the log files collected from the same execution of Figure 5. The normalized average variance value of PADLA_n is smaller than that of PADLA_t in both tomcat and kafka. From these facts, we can infer that PADLA_n is stabler than PADLA_t. However, ideally, PADLA_n's variance value of log file size should be zero because the number-based method observes the method call events for the phase detection and the method call events are expected to be the same in every execution. The reason for this can be inferred as some random elements of the benchmark tool and the target programs or the delay of changing the log level threshold (PADLA uses the socket for the connection between the phase detection and logging component).

From the above, the characteristics of PADLA_t and PADLA_n can be summarized as follows. PADLA_t can achieve better $M(l)$ value than PADLA_n. In addition, the execution time of PADLA_t is greatly shorter than PADLA_n. However, the result can be changed depending on the target program because in the kafka result, the execution time of PADLA_n is shorter than that of PADLA_t. From the stability perspective, PADLA_n is better than PADLA_t because PADLA_n is smaller than PADLA_t comparing the variance value of the log file size.

5 Case Study

To investigate the usefulness of PADLA, we apply PADLA to the irregular behavior of a server system to answer the following questions.

- **Question 1.** Is the log messages output by PADLA useful for debugging?
- **Question 2.** Compared to *info* and *trace*, is PADLA useful?

5.1 Use Case Scenario

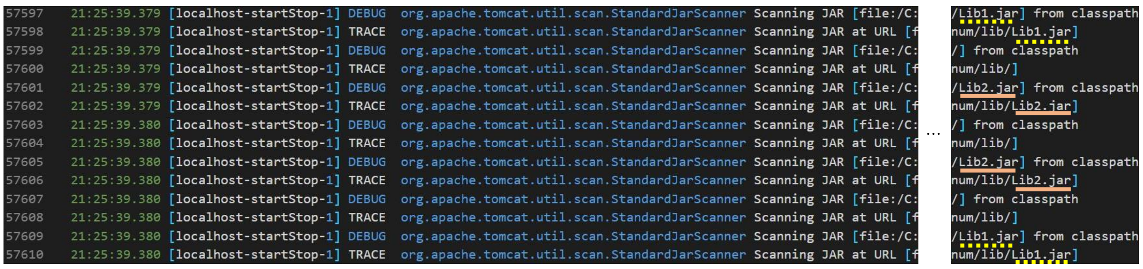
We use PADLA to analyze Bug 59813 chosen from Bugzilla[5] as a sample of real-case bugs of a server system. Bug 59813 is a bug of Apache Tomcat 8.5.3 causing an infinite loop during the starting up of the server system. This is caused by two jar library files read by Tomcat referring to each other in the Classpath inside the MANIFEST.MF file, resulting in the two libraries loaded cyclically. We apply PADLA to the bugged Tomcat, then for the Question 1 we investigate the log file that output when the server starts up. For the Question 2, we measure and compare the log file size and $M(l)$ of each configuration *info*, *PADLA* and *trace*.

To set up the experiment, we have to replicate the bug, and apply PADLA to Tomcat. For this reason, we make two additional library files and add them into the Classpath of Tomcat. The two files (*Lib1.jar* and *Lib2.jar*) refer to each other in the Classpath in the MANIFEST.MF file, so they cause the bug explained above. To apply PADLA to Tomcat, we firstly apply log4j2 to Tomcat and then replace the *log4j-core.jar* with our *log4j-core-extended.jar*. We also add a javaagent option to the *JAVA_OPTS* of Tomcat to install *Agent.jar*.

In this case study, PADLA employs the time-based method as its phase detection method (PADLA_t). The objective of this case study is collecting detailed logs of irregular behaviors of a real-case. As we compared in the Section 4, PADLA_t is better than PADLA_n in the $M(l)$ value. In addition, PADLA_t suppressed the execution time to about one-third of PADLA_n in tomcat. Although the stability of PADLA_n is better than that of PADLA_t, the larger value of the $M(l)$ is desirable for collecting detailed logs of irregular behaviors. Furthermore, PADLA_n takes double execution time of *trace* so using PADLA_n in "real-case" is not realistic. For these reasons, we use PADLA_t.

We start Tomcat and stop it after a given execution time. The execution times are 20s, 30s, 40s, 50s and 60s for each configuration. For each configuration and for each execution time, we execute Tomcat 10 times and then, calculate an average of the log file size and $M(l)$ value.

[5] `https://bz.apache.org/bugzilla/`



```
21:25:39.379 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /Lib1.jar] from classpath
21:25:39.379 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/Lib1.jar]
21:25:39.379 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /] from classpath
21:25:39.379 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/]
21:25:39.379 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /Lib2.jar] from classpath
21:25:39.379 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/Lib2.jar]
21:25:39.380 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /] from classpath
21:25:39.380 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/]
21:25:39.380 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /Lib2.jar] from classpath
21:25:39.380 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/Lib2.jar]
21:25:39.380 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /] from classpath
21:25:39.380 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/]
21:25:39.380 [localhost-startStop-1] DEBUG  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR [file:/C:  …  /Lib1.jar] from classpath
21:25:39.380 [localhost-startStop-1] TRACE  org.apache.tomcat.util.scan.StandardJarScanner Scanning JAR at URL [f  …  num/lib/Lib1.jar]
```

Figure 7: Case study: a portion of the infinite loop output log obtained by PADLA (file paths are omitted)

Table 6: Case study: the log file size and $M(l)$ value of each execution time and each configuration

Execution time (s)	Configuration	File size (KB)	(division by *trace* (%))	M(l) (%)
20	*info*	5	(0.01)	17.43
	PADLA	15,365	(20.81)	100.00
	trace	73,819	(100.00)	100.00
30	*info*	5	(0.01)	17.43
	PADLA	15,249	(14.03)	100.00
	trace	108,690	(100.00)	100.00
40	*info*	5	(0.01)	17.43
	PADLA	15,750	(11.67)	100.00
	trace	147,630	(100.00)	100.00
50	*info*	5	(0.01)	17.43
	PADLA	15,635	(8.66)	100.00
	trace	180,584	(100.00)	100.00
60	*info*	5	(0.01)	17.43
	PADLA	15,382	(7.18)	100.00
	*trace*trace	214,259	(100.00)	100.00

In the experiment explained above, when PADLA is applied, the log level threshold is *info* at the beginning. When PADLA detects an unknown phase, the log level threshold is changed to *trace*. The experiment is conducted with the same environment as Section 4. The parameters I and θ are set to 0.5s and 0.95 respectively. The parameters values follow the default values used in Section 4.

5.2 Result

Figure 7 shows a part of the log file output by PADLA. The *debug* and *trace* level log messages are output because PADLA changes the log level threshold of the target system. In the Figure, lines 57,597, 57,598, 57,609 and 57,610 are the log messages when Tomcat loads *Lib1.jar*, lines 57,601, 57,602, 57,605 and 57,606 are the log messages when Tomcat loads *Lib2.jar*, the two types of log messages repeats while the infinite loop continues. PADLA recorded these log messages in every execution of this experiment. These log messages indicate that after the *Lib1.jar* or *Lib2.jar* is loaded, Tomcat tends to load the other, so we can infer that the two library jar files are the cause of the bug. In addition, the two types of log messages exist only in *debug* and *trace* level, so if the log level threshold is set to *info*, the log messages does not appear in the log file. From these facts we can conclude that PADLA can collect useful information for debugging by adapting the log level threshold.

Table 6 shows that the log file size and $M(l)$ value of each execution time and each configuration. We can see that the log file size of *trace* increases as the execution time increases, while that one of PADLA is relatively stable. The log file size of *trace* continues to increase until Tomcat stops because *trace* continues to output the log of the infinite loop while the loop continues. On the other hand, PADLA learns a phase of the infinite loop and records detailed log messages only at the first few intervals of the loop. So the log file size of PADLA is not affected by the execution time and a reduction rate increases with the execution time. In addition, we can observe that PADLA keeps 100% $M(l)$ value in every execution time while *info* keeps only 17.43%. From these facts, in this case study, PADLA shows an higher $M(l)$ value than *info* while greatly suppressing the log file size that of *trace*.

6 Limitations

In this evaluation, the phase detection function of PADLA is not strictly evaluated. To conduct a strict evaluation for the phase detection function, we need an execution scenario labeled with phases information. The information should specify when a phase starts and whether the phase is known or unknown. However, the execution time of a program depends on its environment and we cannot know accurately when a phase starts so observing phases information is not always possible. For this reason, in this evaluation, we use $M(l)$ to evaluate PADLA.

In the evaluation, the execution time of the number-based method is greatly longer than that of the time-based method and *trace*. So the number-based method is not employed in the case study. However, this long execution time might be caused by the implementation or the parameter setting. For this reason, we can not conclude the usefulness of the number-based method before reconsidering these.

7 Conclusion

This paper presents PADLA, a cost-effective detailed logging method that dynamically adjusts the log level threshold of a running system. Using the phase detection methods, PADLA monitors the behavior of a system and adjusts the log level threshold to appropriate one for the situation. We evaluated PADLA using DaCapo Benchmarks. In the evaluation, we investigated the logging quality of PADLA and the effect of the parameters values. In addition, we compared the characteristics of the two phase detection methods. The time-based method has better $M(l)$ value and execution time than number-based method in the tomcat result, while the number-based method is stabler than the time-based method. As a case study, we applied PADLA to a real-case bug. In the case study, PADLA successfully recorded information that is useful for debugging. The case study also showed that PADLA can reduce the log file size from *trace* while keeping high $M(l)$ value compared to *info*.

Acknowledgement

First of all, I would like to express my sincere gratitude to my supervisor Professor Katsuro Inoue (Graduate School of Information Science and Technology, Osaka University Department of Computer Science) for giving me the opportunity to study and work at Inoue Laboratory. He always gave me a lot of valuable advice and insightful comments.

I would like to thank Associate Professor Takashi Ishio (Graduate School of Science and Technology, Nara Institute of Science and Technology). His guidance helped me in all the time of research and writing of my thesis. Tools which he introduced to me helped to progress my research. He also helped to make a presentation in some conferences.

My sincere thanks goes to Associate Professor Makoto Matsushita (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He always gave me appropriate comments and an opportunity to review the validity of my research.

I am grateful to Assistant Professor Tetsuya Kanda (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He provided me with a lot of insightful comments for implementing my proposed method. He also helps me to manage equipments of Inoue laboratory so I could conduct my research smoothly.

I would like to thank Specially Appointed Professor Shusuke Haruna (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He gave me a lot of valuable comments and suggestions on my research and paper writing.

I am grateful to Kazumasa Shimari (Graduate School of Information Science and Technology, Osaka University Department of Computer Science). He gave me a lot of wonderful research ideas and helpful advice. Without his help, it would not have been possible for me to complete my thesis.

I would like to thank all the members in Inoue laboratory for creating such a wonderful environment for studying and researching.

Last but not least, I would like to thank my family members for supporting me spiritually throughout writing this thesis and my life in general.

References

[1] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proceedings of the 9th ACM International Conference on Autonomic Computing*, ICAC '12, pages 191–200, 2012.

[2] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1285–1298, 2017.

[3] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, 2009.

[4] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering Companion*, ICSE '14, pages 24–33, 2014.

[5] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '18, pages 60–70, 2018.

[6] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering Companion*, ICSE '16, pages 102–111, 2016.

[7] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering*, ICSE '16, pages 911–922, 2016.

[8] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC '10, pages 24–24, 2010.

[9] Katsuya Ogami, Raula Gaikovina Kula, Hideaki Hata, Takashi Ishio, and Kenichi Matsumoto. Using high-rising cities to visualize performance in real-time. In *Proceedings of*

the 5th IEEE Working Conference on Software Visualization, VISSOFT '17, pages 33–42, 2017.

[10] Steven P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the 3rd ACM/IEEE International Workshop on Dynamic Analysis*, WODA '05, pages 1–6, 2005.

[11] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process*, 26(1):3–26, 2014.

[12] Kabinna Suhas, Shang Weiyi, Bezemer Cor-Paul, and Ahmed E. Hassan. Examining the stability of logging statements. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 326–337, 2016.

[13] Yui Watanabe, Takashi Ishio, and Katsuro Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 6th ACM/IEEE International Workshop on Dynamic Analysis*, WODA '08, pages 8–14, 2008.

[14] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, 2009.

[15] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGARCH Computer Architecture News*, 38(1):143–154, 2010.

[16] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering*, ICSE '12, pages 102–112, 2012.

[17] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 807–817, 2019.