LogReducer: Identify and Reduce Log Hotspots in Kernel on the Fly

Guangba Yu
Sun Yat-sen University
Tencent Inc.
yugb5@mail2.sysu.edu.cn

Pengfei Chen*
Sun Yat-sen University
chenpf7@mail.sysu.edu.cn

Pairui Li
Tencent Inc.
perryprli@tencent.com

Tianjun Weng
Tencent Inc.
tianjunweng@tencent.com

Haibing Zheng
Tencent Inc.
mattzheng@tencent.com

Yuetang Deng
Tencent Inc.
yuetangdeng@tencent.com

Zibin Zheng
Sun Yat-sen University
zhzibin@mail.sysu.edu.cn

***Abstract*—Modern systems generate a massive amount of logs to detect and diagnose system faults, which incurs expensive storage costs and runtime overhead. After investigating real-world production logs, we observe that most of the logging overhead is due to a small number of log templates, referred to as log hotspots. Therefore, we conduct a systematical study about log hotspots in an industrial system WeChat, which motivates us to identify log hotspots and reduce them on the fly. In this paper, we propose *LogReducer*, a non-intrusive and language-independent log reduction framework based on eBPF (Extended Berkeley Packet Filter), consisting of both online and offline processes. After two months of serving the offline process of *LogReducer* in WeChat, the log storage overhead has dropped from 19.7 PB per day to 12.0 PB (i.e., about a 39.08% decrease). Practical implementation and experimental evaluations in the test environment demonstrate that the online process of *LogReducer* can control the logging overhead of hotspots while preserving logging effectiveness. Moreover, the log hotspot handling time can be reduced from an average of 9 days in production to 10 minutes in the test with the help of *LogReducer*.**

***Index Terms*—Log Hotspot, eBPF, Log Reduction, Log Parsing**

I. Introduction

Over the years, software systems have become increasingly large and complex, which has primarily exacerbated the difficulty of maintaining them [1]–[5]. Logs, which record runtime information of systems, are the favorite data source used by Site Reliability Engineers (SREs) to check system status, detect anomalies and diagnose root causes [6]–[9]. A large system can produce a massive amount of logs to cope with a wide range of faults. As shown in Figure 1, a large real-world instant messaging application WeChat produces about 16-20 pebibyte (PB) (75-100 trillion lines) of logs per day.

Although logs are helpful, it is crucial to avoid excessive logging, as logging incurs both storage cost (§ III-B) and runtime overhead (§ III-C). To reduce logging overhead, we conduct a study on the characteristics of logs in a real-world system. What surprised us was that *most of the logging overhead is due to a very small number of log templates*, where log template is the constant part of a log statement in the code. For example, for service 1 of WeChat in Figure 6, the top1 log template consumed 95.7% of the storage, while the remaining 297 templates consumed 4.3% of the storage of service 1. We refer to these templates that take up most of the storage as *log hotspots*.

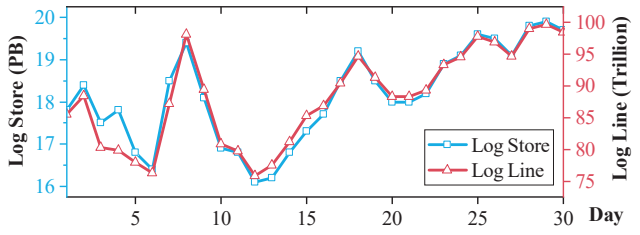


Fig. 1. In April 2022, WeChat produced about 16-20 pebibyte (PB) (around 75-100 trillion lines) of logs per day.

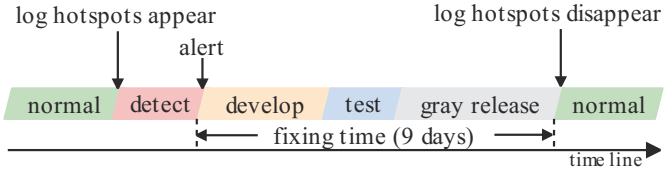


Fig. 2. Process of log hotspot identification and reduction.

Therefore, reducing log hotspots can efficiently reduce most of the logging overhead. However, as shown in Figure 2, due to the complex testing and release mechanisms (e.g., gray release [1]), developers take on average 9 days to fix log hotspots (§ III-F). Existing efforts mainly focus on where to log during the development phase [10]–[13], which cannot reduce hotspots at runtime in a timely manner. Some log compression approaches can reduce the storage overhead [14]–[16], but they cannot prevent the overhead of writing logs to disk and sending logs to the database. In other words, there is a gap between the log hotspots detection of SREs and the fixing of hotspots of developers, which leads to expensive costs for application performance and log storage.

To fill this significant gap, we first systematically study log hotspots in a real-world system WeChat, focusing on the impact of log hotspots, why they occur and how to

[1]Gray release is the process of gradually switching from the existing system to the novel system with a new version.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00151

fix them. After studying the log hotspots for 19 services and interviewing 19 experienced developers at WeChat, we obtained some insightful findings on detecting and fixing log hotspots. This empirical study also motivated us to reduce log hotspots on the fly.

To achieve that goal, we propose a non-intrusive, language-independent, and efficient log reduction framework based on eBPF (Extended Berkeley Packet Filter) [17], namely *LogReducer*, which automatically identifies log hotspots and reduces them at runtime. *LogReducer* is composed by three modules including *Log Parser*, *Hotspot Classifier* and *Log Filter*. First, Log Parser periodically queries raw logs from the database and parses them as log templates. Hotspot Classifier takes log templates and storage information as input to determine which log templates are hotspots. If log hotspots exist for a service, Hotspot Classifier notifies developers with reduction tasks offline. To reduce the impact of log hotspots during the fixing phase, *LogReducer* launches a Log Filter on each node that holds an instance of that service in the online process. Afterward, Log Filter loads the information of hotspots into the Linux kernel and efficiently filters and eliminates log hotspots in the Linux kernel space based on eBPF on the fly.

Contribution. To sum up, this work makes the following major contributions:

- We conduct a systematical study about log hotspots in a real-world system to reveal the impact of log hotspots, why log hotspots occur, and how to fix them.
- We propose a non-intrusive, language-independent, and efficient log reduction framework to identify log hotspots automatically and reduce them on the fly.
- The offline process of *LogReducer* is already adopted in an industrial system WeChat and is used by SREs daily. After two months, the storage overhead of logs dropped in WeChat from 19.7 PB per day to 12.0 PB.
- We conduct extensive experiments to validate the efficiency of the online process of *LogReducer* in reducing log hotspots at the program running. The fixing time of hotspots can be reduced from an average of 9 days in production to 10 minutes in test with the help of *LogReducer*.

II. Backgroud

WeChat System. WeChat is a large real-world instant messaging system serving billions of users globally. The backend of WeChat is constructed based on a microservice architecture, which accommodates more than 20,000 services running on over 600,000 machines. WeChat essentially needs to handle hundreds of millions of requests per second. To maintain this large and complex system, software developers insert logging statements into the source code to record necessary runtime information, such as the state of the system and error messages.

Log template and lifecycle. Although logs have brought benefits, generating, collecting, and storing such massive logs impose an expensive burden on WeChat. As depicted in Figure 1, WeChat can produce 20 PB logs per day. To reduce the

Log Format	< level> <service (pid,tid,cid,traceid)> time [code location] log info
Log Statement	MMERR("REQ %s Failed ", id);
Log Message	< 2> <Test (6,6,6,6666)> 11:48:43 66 [test.cpp:Test:6] REQ 6 Failed
Log Template	Error Test test.cpp:Test:6 REQ <ID> Failed

Fig. 3. An example of log format, logging statement, log message, and log template after log parsing in WeChat. Due to confidentiality, we do not disclose the service, file, and function names.

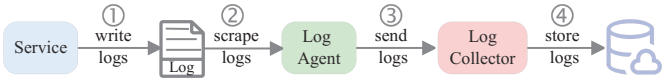


Fig. 4. The life cycle of a log from generation to persistence.

logging overhead, we first need to understand the template and lifecycle of logs. As shown in Figure 3, logs are composed of log headers (e.g., level and time) and log information, which in turn consists of two elements: 1) static descriptive words hard-coded in source code (e.g., REQ); 2) dynamic variables vary with executions (e.g., RequestID). The log level is represented as <level> at the beginning of logs. $< 1 >, < 2 >$ and $< 3 >$ in log signatures represent *Important, Error and Debug* respectively. Log headers are usually generated automatically by log formats, while developers specify log information in logging statements. A log template is the constant part of logs generated by the same log statement [18].

Figure 4 provides an overview of the logs' life cycle in relation to different stages in production. ① **Write log:** services write logs into log files continuously whenever the execution flow reaches logging statements; ② **Scrape log:** a log agent (e.g., Promtail [19]) of each node scrapes updated logs from logs files; ③ **Send log:** a log agent pushes logs to collectors (e.g., Loki [20]) via the network. ④ **Store log:** a log collector persists logs into the log database (e.g., ClickHouse [21]) and rotates them periodically. From the life cycle, we can conclude that logs not only affect the cost of storage but also consume resources (e.g., CPU) when writing logs and network bandwidth when sending logs.

Log hotspots. We formally define log hotspots as follows. Given a service A, it has n log templates, denoted $log_1, log_2, ..., log_n$, which correspond to logging statements. Suppose that logs of A occupy S GB space in the last time window, where $log_1, log_2, ..., log_n$ occupy $S_1, S_2, ..., S_n$ GB space ($S = S_1 + S_2 + ... + S_n$) respectively. A log template log_i is recognized as a log hotspot if and only if

$$\frac{S_i}{S} > \xi, \tag{1}$$

where ξ ($\xi = 0.05$ by default) is the threshold set by SREs. Considering log hotspots are non-transient and last for a long time without human intervention, we choose a moderate time window (10 minutes by default) to avoid transient noises and respond to hotspots timely. Note that the threshold can be tuned in other systems based on the tolerance to log hotspots, which is not the focus of this work.

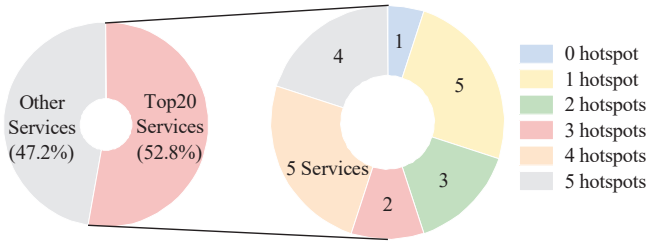


Fig. 5. The top 20 services with the highest storage in WeChat account for 52.8% of the total storage. 19 of the 20 services contain at least one log hotspots.

III. Empirical Study on Log Hotspots in Industry

A. Data Collection and Research Questions

In this section, we aim to investigate the characteristics of log hotspots. We analyzed the logs of the top 20 services with the highest log storage in WeChat on May 7th, 2022, based on our *log parser* in Section IV-C. In total, we found 57 log hotspots from 19 services. For each service, we can identify the service owner who is responsible for that service through the code repository. We sent the log hotspots to the corresponding developers, who determined that all of them could be eliminated.

We then experienced the entire processes of reducing 57 log hotspots and interviewed 19 corresponding service owners to understand why log hotspots were generated and how to deal with them. Since all the interviewees are experienced code maintainers and familiar with hotspot log statements, we take the ideas of the interviewees as the primary guide of root causes and solutions. During each interview, the interviewer presented the root cause and the solution for each log hotspot. We then summarised the presentation as a category. For example, if the root cause presentation was "The log statement is a test log that we forgot to remove", we summarised it as "Forgotten Test Log" type. If interviewees agree with our summaries, they will confirm our results. Otherwise, they can modify our results until matching their views.

Based on the 57 log hotspots from WeChat, our study aims to address the following research questions (RQs):

- **RQ1: How do log hotspots impact storage?**
- **RQ2: How do log hotspots impact runtime?**
- **RQ3: What are the root causes of log hotspots?**
- **RQ4: What are the fixing solutions of log hotspots?**
- **RQ5: How long do developers take to fix log hotspots?**

B. RQ1: Storage Overhead of Log Hotspots

The left part of Figure 5 shows the top 20 services with the highest log storage, accounting for 52.8% of the total storage, while the other 20,000+ services account for the remaining 47.0%. This result shows that optimizing the logs of the top 20 services is the most cost-effective. Therefore, we dug into the logs of these 20 services to mine knowledge related to log hotspots. From the right part of Figure 5, 19 services (19 out of 20) contain at least one log hotspot, revealing that log hotspots are common in industrial systems. Specifically, there are 5, 3, 2, 5, and 4 services containing 1, 2, 3, 4, and 5 log hotspots, respectively.

Figure 6 shows the storage percentage of log hotspots for the 19 services that contain log hotspots. For confidentiality, we use the number ID to indicate the names of the services. The gray color denotes the sum of the storage percentage of all log templates that are not hotspots, while the other color indicates the storage percentage of a log hotspot. From Figure 6, we observe that log hotspots occupied an average of 57.86% of the corresponding service's storage. In particular, the log hotspot of service 1 occupied 95.7% of the storage, while the other 297 log templates only took up 4.3% of the storage of service 1. The results in Figure 6 demonstrate that most of the logging overhead is due to a small number (less than 6) of log hotspots. These results suggest that streamlining a small number of log hotspots can reduce the storage overhead by a significant amount. Compared with traversing and checking all logging statements, developers only need to handle a small number of logging statements if hotspots are identified, which is cost-effective in reducing storage overhead.

> **Finding 1.** Log hotspots are prevalent in different services. A small number of log hotspots occupy an average of 57.86% of the corresponding storage.
>
> **Implication 1.** Reducing log hotspots is cost-effective in reducing storage overhead.

C. RQ2: Runtime Overhead of Log Hotspots

Runtime overhead is considered a major cost of logging [22], [23], as generating log strings involves string concatenations and possible method invocations, writing logs into log files involves expensive IO operations, and sending logs into the log collector backend involves significant network overhead. Some log hotspots overwhelmingly generate similar log messages repeatedly, which incurs an unignorable and unnecessary runtime overhead.

To investigate the runtime overhead of log hotspots on real systems, we worked with developers to experience the complete reduction process from detecting log hotspots on Service 16 to the finishing of the gray release. On May 9, 2022, we detected that Service 16 contains 5 log hotspots and reported them to its developer. The developer fixed the log hotspots on 10 May and released the new version entirely on May 12. Specifically, this new version only modifies the logging statements compared to the old version. After fixing the log hotspots, the log volume of Service 16 has dropped from 162 TB (on May 9) to 3.66 TB (on May 13) under a similar workload, a reduction of 97.7%.

Figure 7 shows the resource usage and performance of Service 16 before and after the logging hotspot fix (i.e., May 9 and May 13). From 7, we observe that Service 16 with log hotspots consumed up to 5.18% more CPU (58 cores in total) and 0.6% more memory (73 GB in total) per minute than Service 16 without log hotspots. Additionally, Service 16

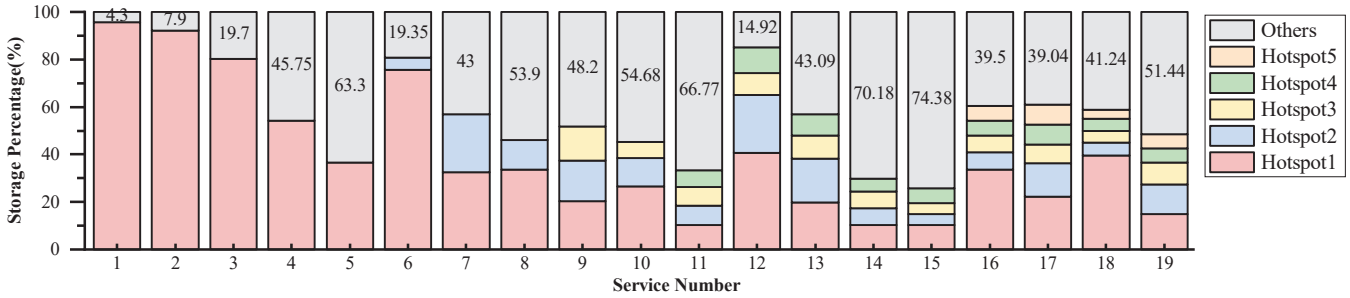


Fig. 6. For services containing at least one log hotspot in Figure 5, log hotspots occupy an average of 57.86% of the corresponding service storage.

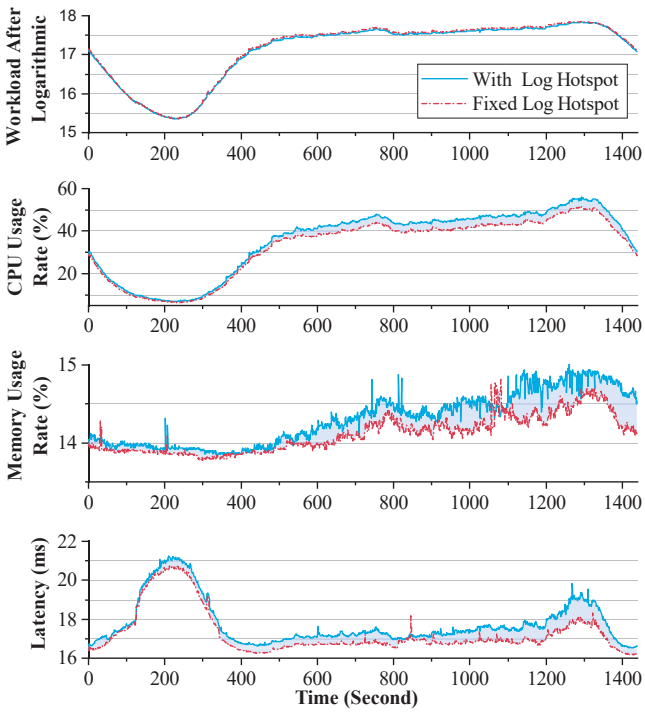


Fig. 7. Under a similar workload, Service 16 with log hotspots consumed up to 5.18% more CPU and suffered up to 3% additional response latency than Service 16 after fixing log hotspots.

with log hotspots suffered up to 3% more response latency (1.8 ms) than Service 16 without log hotspots. Moreover, fixing log hotspots reduced the number of logs sent from Service 16 to the log collector from 162 TB to 3.66 TB per day, a significant saving in network bandwidth.

Finding 2. Log hotspots incur unignorable resource consumption (e.g., CPU, memory, IO, and network bandwidth) and performance degradation.

Implication 2. Reducing log hotspots not only optimizes the resource consumption of applications but also improves their performance.

D. RQ3: Root Causes of Log Hotspots

It is worth noting that in the case of Figure 7 the 5 log hotspots for Service 16 took up 61.5% of the log volume, while the total log volume was reduced by 97.7%. This is because when developers fixed the log hotspots, they found some unnecessary logs printed for the same reasons as the log hotspots. As a result, they reduced these unnecessary logs as well. For example, the root cause of the top 1 log hotspot of Service 16 is a forgotten test log. When the developer checked the logging statement of the log hotspot, he also found other forgotten test logging statements and removed them. This motivates us to understand the nature of log hotspots in order to help developers avoid printing unnecessary logs.

As described in Section III-A, for each log hotspot in Figure 6, we interviewed its service owners to label the root cause and corresponding solution. Based on the labeling process, we identify the following 8 root causes of log hotspots.

① **Incorrect Log Level.** This kind of log hotspots is caused due to incorrect log levels, such as debug logging statements are set to error level. The incorrect log level in the logging framework would produce massive amounts of low-level logs because p level logging statement is enabled in the q level logging framework if $p >= q$.

② **Forgotten Test Log.** This kind of log hotspots is caused by developers forgetting to delete logging statements used in the testing phase when releasing services.

③ **Dependent Module Fault.** This kind of log hotspots occurs because a service's dependent down-streams modules are experiencing intermittent faults. Thus the service would continue to print similar error logs or throw exceptions due to failed invocations.

④ **Dependent Package Log.** This kind of log hotspots occurs because developers focus on the logging statements in their code while ignoring the logs printed by the logging statements in their dependent packages.

⑤ **Incorrect Log Dye.** Log dye is a general method of reducing log volume by adding staining marks to specific requests and only sampling the logs of those requests. This kind of log hotspots is caused by the incorrect configuration of log dye that results in sampling all logs.

⑥ **Reasonable Hotspot.** This root cause refers to that developers determined that the log hotspots are necessary for diagnosing faults, and the massive amount of logs is caused by the massive workload volume.

⑦ **Self-Module Fault.** This kind of log hotspots is caused by the problematic implementation logic of the service.

⑧ **Others.** Each log hotspot in this root cause is unusual and cannot be assigned to any other root cause.

Table I shows the statistics of log hotspots corresponding

TABLE I
STATISTICS OF ROOT CAUSE FOR LOG HOTSPOTS.

Root Cause \ Metric	Distribution		Fixing Time(day)		
	Count	%	Mean	Std	Med
Incorrect Log Level	**23**	**40.35**	10.91	**11.28**	5
Forgotten Test Log	13	22.8	4.76	1.87	4
Dependent Module Fault	6	10.5	2.83	0.408	3
Dependent Package Log	5	8.77	3	0	3
Incorrect Log Dye	4	7.01	**41**	0	**41**
Reasonable Hotspot	3	5.26	5	2.0	5
Self-Module Fault	2	3.5	2	1.41	2
Others	1	1.75	3	0	3
Total	57	100	9.31	11.83	3

TABLE II
STATISTICS OF ROOT CAUSE FOR LOG HOTSPOTS.

Root Cause \ Metric	Distribution		Fixing Time(day)		
	Count	%	Mean	Std	Med
Correct Log Level	**18**	**31.57**	11.77	**12.68**	3
Delete Log Statement	17	29.82	5.76	2.07	7
Fix Module Fault	9	15.78	2.66	0.7	3
Turn on Log Dye	6	10.52	3	0	3
Correct Log Dye	4	7.01	**41**	0	**41**
Merge Log Statement	2	3.5	4	1.41	4
Reduce Log Length	1	1.75	7	0	7
Total	57	100	9.31	11.83	3

to the identified root causes. Among all these root causes, Incorrect Log Level and Forgotten Test Log are the two most common root causes, accounting for 63.15% of log hotspots in total. The reason mainly lies in that developers focus more on program logic than on logging statements, resulting in poor log quality. This indicates that the automated approach that leverages the features of logging statements to make suggestions on choosing log locations and levels may be helpful during the development phase [10]–[12], [24], [25]. Moreover, we observe that faults may accompany log hotspots (e.g., Dependent Module Fault and Self-Module Fault). Therefore, timely detection of log hotspots can improve system availability.

Finding 3. Among all root causes, the two most common root causes are Incorrect Log Level and Forgotten Test Log, accounting for 63.15% in total.

Implication 3. Root causes across services have common points. Understanding these root causes can help to reduce the occurrence of log hotspots. Intelligent log location and level suggestions will be appreciated.

E. RQ4: Fixing Solutions of Log Hotspots

After figuring out the root causes of log hotspots, we experienced the processes of fixing log hotspots with developers and interviewed them to label the solutions. Based on the labeling process, we identify the following 7 fixing solutions.

① **Correct Log Level.** This solution refers to that developers choose a higher log level (e.g., from debug to error) for the log format.

② **Delete Log Statement.** This solution refers to that developers determine the logging statements are unnecessary and delete the statements of log hotspots.

③ **Mitigate Module Fault.** This solution refers to developers identifying faults in dependent modules or their modules when checking for log hotspots and then mitigating module faults to reduce log hotspots.

④ **Turn on Log Dye.** This solution indicates that developers determine the logging statements are necessary, but not all requests need to be recorded. Therefore, they try to turn on the log dye and record only those requests with staining marks.

⑤ **Correct Log Dye.** This solution refers to the fact that developers observe that the log dye is not working because it was incorrectly configured. Thus, they correct the configuration of log dye and release the service.

⑥ **Merge Log Statements.** This solution refers to that developers determine the logging statements are necessary, but they find that some log statements can be merged and printed together to reduce log content.

⑦ **Reduce Log Length.** This solution refers to the cases where developers observe that the number of logs is not very large, but the length of logs is very long (more than 1000 chars). Therefore, developers try to reduce the length of the logs to reduce log content.

Table II shows the statistics of fixing solutions corresponding to log hotspots. Among all these solutions, Correct Log Level and Delete Log Statements are the two most common solutions, accounting for 61.69% of log hotspots in total. To reveal why these two are the most frequent fixing solutions, we depict the relation between fixing solutions and root causes in Figure 8. From Figure 8, we observe that the reason lies primarily in the poor log quality discussed in Section III-D. Specifically, 18 of 23 log hotspots caused by Incorrect Log Level and 12 of 13 log hotspots caused by Forgotten Test Log are fixed by Correct Log Level and Delete Log Statement. Moreover, for the reasonable log hotspots, developers attempted to reduce the amount of logs by Turn on Log Dye, Merge Log Statement, and Reduce Log Length. These fixing solutions can also be applied during development to prevent log hotspots.

Finding 4. Among solutions, the two most common fixing solutions are Correct Log Level and Delete Log Statement, accounting for 61.69% in total.

Implication 4. Developers can benefit from historical solutions that not only speed up the fixing but also prevent log hotspots during the development phase.

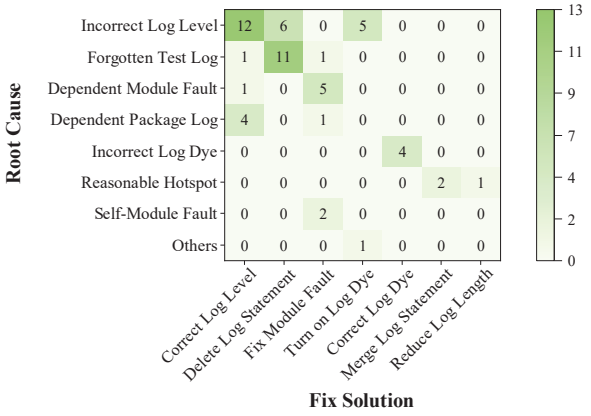


Fig. 8. The relation between fixing solutions and root causes.

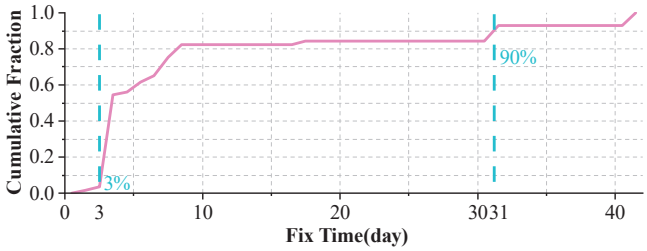


Fig. 9. Distribution of fixing time of log hotspots.

F. RQ5: Fixing time of Log Hotspots

Figure 2 shows that fixing log hotspots needs to experience three phases (i.e., development, test, and release) after developers were notified by log hotspots. We use *fixing time* to denote the time it takes for developers to fix a log hotspot. It includes the time taken to modify programs, test the updated programs, and release new versions. Figure 9 shows the CDF (Cumulative Distribution Function) of the fixing time of log hotspots. We observe that most of the log hotspots are not fixed by developers in time. For 97% of log hotspots, it takes developers at least 3 days and on average 9 days to fix them. About 10% of the log hotspots even took more than 30 days to fix. Two main reasons cause the long fixing time: 1) the complex test and release process of modern software; 2) the cost of storing logs is not a concern for developers but SREs, resulting in no incentive for developers to reduce logs. Such a long fixing time leads to a significant gap between detecting log hotspots by SREs and fixing them by developers, which leads to expensive costs in system performance and log storage.

In addition, after monitoring the services continually in Figure 6, we find that after a service's current log hotspots are fixed, new log hotspots may continue to emerge as successive updates on the service. Figure 10 shows the distribution of the number of times new hotspots that appear in two months after their historic hotspots have been fixed. As shown in Figure 10, 18 of 19 services have been alerted by our *LogReducer* (details in Section IV), because new log hotspots appear. Furthermore, two services have even encountered 6 log hotspot alerts in two months due to frequent releases. Therefore, reducing log hotspots is not a one-time task but a repetitive one. There is

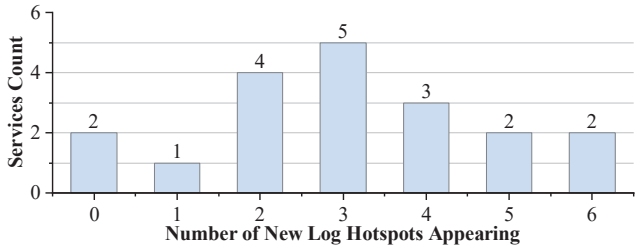


Fig. 10. Distribution of times of new log hotspots appearing in two months after old log hotspots in Figure 6 have been fixed.

a need to reduce hotspots automatically.

Finding 5. For 97% of log hotspots, developers need at least 3 days and on average 9 days to fix them. After the historical hotspots are fixed, 18 of the 19 services encounter new log hotspots.

Implication 5. It is important and necessary to automatically fill the gap between the detection and fixing of log hotspots in production environments on the fly.

IV. *LogReducer* Framework

As shown in the above studies, a small number of log hotspots account for most of the logging overhead. Nevertheless, we notice that most of the log hotspots could not be fixed by developers in time. SREs are desperate for a tool to automatically identify log hotspots and reduce them.

A. Design Challenges

Designing a practical log reduction approach against log hotspots poses several challenges:

1) **Massive log volume.** WeChat produces around 75-100 trillion lines of logs per day (Figure 1). The approach must be fast enough to handle such a large volume of logs without affecting the performance of applications.
2) **Free of developer effort.** The log reduction approach should be transparent to developers and should not place an additional burden on developers.
3) **Multiple program languages.** WeChat consists of 20,000+ services developed by different teams using different programming languages. The log reduction approach should be compatible with multiple languages.
4) **Online production environment.** WeChat serves billions of users in the production environment. To avoid impacting the quality of services, the log reduction approach should be enabled on the fly rather than restarting services.

Existent possible reduction methods are still criticized for their unsatisfactory granularity and efficiency. 1) Adjusting log configuration typically filters a kind of log rather than a specific template. For example, if adjusting the log level from debug to error, all debug logs would not be printed. Additionally, adjusting software configurations usually requires

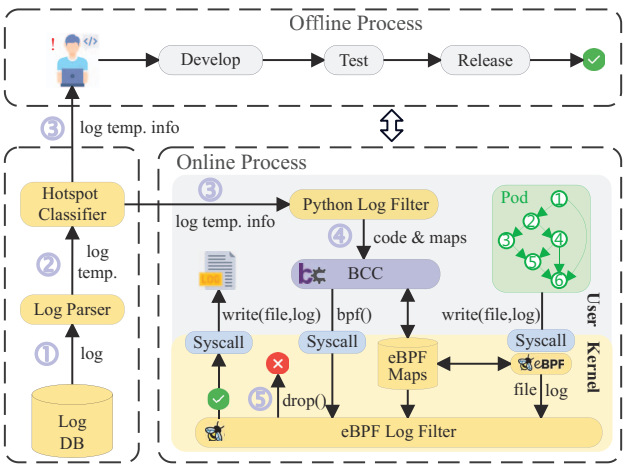


Fig. 11. An overview of the proposed *LogReducer* framework.

restarting systems to activate new configurations, which is unacceptable in production. 2) If filtering log hotspots in log agents, it results in massive disk IO operations to write logs to disks, which is inefficient and resource intensive when facing massive logs. 3) If filtering log hotspots in log collectors, in addition to huge IO operations, significant network bandwidth is consumed to send log hotspots to log collectors, which affects the response latency of services. To address all the above shortcomings of existent methods, we propose *LogReducer*, a non-intrusive, language-independent, and efficient framework designed to filter log hotspots in the Linux kernel space.

B. Framework Overview

Figure 11 shows the framework of *LogReducer* [2]. ① Log Parser module periodically queries logs of a service from the log database and parses the raw logs to log templates (§ IV-C); ② Hotspot Classifier module determines whether the service contains log hotspots based on the storage information of log templates (§ IV-D); ③ Hotspot Classifier module triggers the log reduction process for both offline and online process if log hotspots exist. In the offline process, *LogReducer* notifies the developers of the service to fix the root causes of hotspots; ④ In the online process, our Python Log Filter module loads the eBPF code and log template information into kernel space (§ IV-E1); ⑤ eBPF Log filter module intercepts $sys\_write()$ syscall when a service instance attempts to write logs into a file. If the contents of intercepted logs match the log templates of log hotspots, *LogReducer* drops them in kernel space (§ IV-E2).

C. Log Parser

For each service, the Log Parser module periodically queries all logs in the last time window from the log database as input. Considering log hotspots are non-transient and last for a long time without human intervention, SREs choose a moderate 10 minutes time window to avoid transient noises and respond to hotspots timely. As shown in Figure 3, Log Parser is applied to convert unstructured raw logs into structured log templates that would facilitate further analysis. Parsing a raw log message consists of automatically extracting necessary log headers and distinguishing common parts from the dynamic variables. Writing a regular expression for every logging statement is labour-intensive and time-consuming in practice. Thus, some automatic log parsers have been proposed in recent years [26]–[30].

However, existing log parsers are still complained for the unsatisfactory parsing accuracy and performance. Specifically, AEL [27] took about 450 seconds when parsing one million log messages. Owing to code reuse, a service may have some of the same logging statements in different code locations. Existing log parsers that extract common parts as templates tend to mistakenly treat the same logging statements at different code locations as the same template.

For convenience, we name the strings that can uniquely identify a logging statement as the *log signature*. To deal with the problems mentioned above, we propose a log parser that combines log signature matching and frequency analysis to parse large log files efficiently. In this study, we use the location of the logging statement (e.g., controller.go:107 in Kubernetes Controller [31]) as the log signature because it is precisely bound to the logging statement. In fact, adding code location into logs is common in open-source systems such as Kubernetes [32] and Promethues [33], as well as in industrial systems such as WeChat and Tencent Cloud [34].

Figure 12 depicts an example of Log Parser. ① Given a raw log message, Log Parser conducts word splitting with spaces, tabs, or other special characters. Log Parser then uses regular expressions to extract the log level, service name, and code locations from log headers and filter out numeric characters. After that, the log messages are clustered into coarse-grained groups based on their code location. Log Parser also records the total storage usage of the logs in each cluster in this phase; ② For each cluster, Log Parser builds a frequency table that has the number of times a particular word occurs in the first log in that cluster. In the table of Figure 12, we show a frequency table after we parse through the 3 logs in the [s1.cpp:6] cluster; ③ For each cluster, Log Parser looks for words with as many occurrences as or more than the number of logs in that cluster. We extract log templates based on a key property, i.e., if the word appears in every log in a cluster, then it is a constant word. Finally, Log Parser consolidates the extracted information of log headers and all constant words to the log template.

D. Hotspot Classifier

Hotspot Classifier module takes the log templates and corresponding storage information as input. It identifies log hotspots from all log templates based on Equ. 1. In this study, WeChat SREs empirically set $\xi = 0.05$. For the identified hotspot, Hotspot Classifier triggers a reduction process in the offline and online processes, respectively.

[2]https://github.com/IntelligentDDS/LogReducer

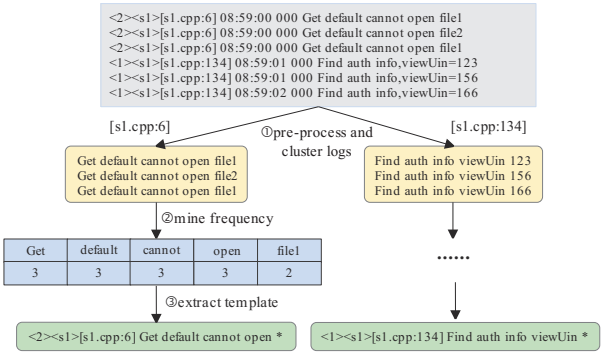


Fig. 12. An example process of Log Parser. For confidentiality, we replace the service, files, and uins in logs as dummy ones.

In the offline process, if log hotspots exist in a service, Hotspot Classifier alerts the developers of the service informing them of the locations and templates of the log hotspots. Lessons learned from the empirical study (§ III) can help developers speed up locating the root causes of log hotspots and fixing them. After developers have fixed the log hotspots, the updated service will be redeployed to the online environment. These hotspots will not appear in the new release.

In the online process, we need to fix log hotspots on the fly, as systems in production cannot be stopped. When Hotspot Classifier identifies a log hotspot for a service, it will start a Log Filter on each node that holds an instance of that service. Filtering all log hotspots without developers' intervention is decided by SREs because it is costly to diagnose an issue if a tremendous amount of redundant logs is present [22]. These filters can also be stopped if developers do not wish to filter that log hotspot. After developers have fixed the log hotspot, *LogReducer* will remove these Log Filters.

E. Log Filter

As discussed in Section IV-A, our Log Filter module is expected to be efficient, non-intrusive and language-independent. Thus, in this study, we use eBPF (Extended Berkeley Packet Filter) [17] to intercept and filter the write operations of log hotspots in the kernel space.

eBPF is an in-kernel virtual machine that allows running user space-provided code in the kernel in a sanitized way [35], [36]. An eBPF program can be loaded from the user space to the kernel space and triggered by a specific kernel event, e.g., file writing. Compared with BPF, eBPF introduces a new bytecode and just-in-time compilation, which allows eBPF programs to achieve native code performance [37]. eBPF programs can maintain and access persistent memory thanks to kernel data structures called BPF maps [38]. BPF maps are used to communicate between different eBPF programs or between eBPF programs and user applications. Our Log Filter is implemented with BPF Compiler Collection (BCC) [39], a development toolchain for eBPF programs that combines Python user space code with eBPF code written in C.

Given a target log hotspot of a service, *Log Reducer* launches a Log Filter on each host that holds the service based on the configuration management database (CMDB). In this study, we name the Python user space code and eBPF code in Log Filter as Python Log Filter and eBPF Log Filter, respectively.

1) Python Log Filter: Our Python Log Filter takes information about the target service and the log hotspot as input. The Python Log Filter on each host then locates all the process IDs (PIDs) associated with the target service on that host. Each service runs as a process with a command line (i.e., service name). We associate processes with services by searching the processes of the command line with the Linux tool Process Status (ps). For example, *"ps -ef | grep service_name"*. For each PID, Python Log Filter obtains the file descriptor (FD) of its log file and the location of the log signature (e.g., the location of log signature $[s1.cpp : 6]$ in Figure 12 is 5) by monitoring its open file process. Finally, Python Log Filter updates the PID, FD, and log signature into eBPF maps and loads eBPF Log Filter into kernel space via $bpf()$ syscall.

2) eBPF Log Filter: Our eBPF Log Filter is executed in kernel space and triggered by a user space program calls to $write()$ syscall, which writes information to disk. When a log is written into a log file, the eBPF Log Filter first determines whether the PID of the program writing to the file is in the PID map and whether the FD corresponding to the file is in the FD map. If both are found, the eBPF Log Filter reads what is written from kernel space starting from the location of the log signature. After that, eBPF Log Filter performs a prefix match between the log signature and the written content. If the match is successful, the eBPF code will skip the remaining execution of $write()$, override and return the execution result directly to drop the log and avoid writing to disk. Otherwise, logs will be written to disk as before.

In this study, we perform a prefix match based on log signature location rather than a fuzzy match on the total content, as eBPF programs are limited to the maximum of 4096 assembly instructions before Linux kernel version 5.2. Moreover, the BPF assembly instructions generated after the compilation may be significantly higher than the number of lines of source code [40]. Thanks to the native code performance and no context switches involved, the eBPF Log Filter can handle the massive amount of logs in the kernel in real time without affecting the application code running (challenge 1 solved). Furthermore, eBPF is a kernel technology that allows our log filter to run without changing the source code of applications or adding additional modules (challenge 2-4 solved).

For compatibility with log hotspots that do not have log signatures, we also provide a user space Log Filter. The user space Log Filter exploits eBPF to intercept log messages in the kernel and send them to user space. Then Log Filter performs a fuzzy match between the log messages and the log templates in the user space. We discuss the performance differences between kernel space and user space Log Filter in Section V-C.

V. Experimental Evaluations

In this section, we evaluate *LogReducer* to answer two questions:

- How effective and efficient is *LogReducer* in parsing logs?
- How effective and efficient is *LogReducer* in filtering log hotspots?

A. Experimental Settings

Log Datasets. To evaluate the effectiveness and efficiency of the Log Parser, we conduct extensive experiments on 3 log datasets collected from service $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ in WeChat. $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are randomly selected from the 19 investigated services. We query their logs in the last 10 minutes. The $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ dataset contain 3,615,678, 9,017,654 and 6,712,058 lines of logs, respectively. Each log message is labeled with a log template based on its logging statement as ground truth. We compare our Log Parser with 4 state-of-the-art methods (including LogCluster [26], AEL [27], Drain [28], and LFA [29]) on all 3 log datasets. We use Message-Level Accuracy (MLA) [41], where a log message is considered correctly parsed if and only if every token of the message is correctly identified as the template or parameter, to measure the effectiveness of methods.

Log Benchmarks. To quantitatively characterize the overhead of the Log Filter, we implement 4 log benchmarks based on 4 widely-used logging frameworks with Golang, Python, Java, and C++ in WeChat. We evaluate the overhead of log filters at different log counts by controlling the number of logs printed per second (from 500/second to 10,0000/second, i.e., from 720 thousand to 8.6 billion lines per instance per day). When evaluating the overhead of the log filter with different lengths of logs, we adjusted the length (from 50 chars to 1,000 chars) of 10,000 logs printed per second in the benchmark. This is because most of the logs in WeChat are less than 1000 chars long.

Implementation and Settings. We conduct experiments on a server with the 16-core AMD EPYC 7K62 Processor (2.6 GHz) and 32GB memory, running with Tencent Linux 3.2 with Linux kernel v5.4. The code of *Log Reducer* is implemented based on Python 3.6 and BCC 0.24.

B. Log Parser Evaluation

The results of Message-Level Accuracy for log parsers are shown in Table III. From the table, we observe that the log parser of *Log Reducer* outperforms the other methods on all datasets. Compared with the powerful log parser Drain, our log parser outperforms it by 57.03% on MLA on average. The reason why our log parser performs better because we group logs based on log signatures, which can identify logging statements precisely.

Besides parsing accuracy, performance is another critical metric for log parsers. Thus, we compare the running time of our log parser with other log parsers under different volumes of log data. From the results in Figure 13, it can be seen that the running time of log parsers increases slowly with the log scale expansion. Even at the scale of one million log messages, our log parser took about 96 seconds, only about half of the time used by Drain (236 s) and around a quarter of the time used by AEL (453 s).

TABLE III
COMPARISON WITH THE STATE-OF-THE-ART LOG PARSERS ON MLA.

Datasets	LogCluster [26]	AEL [27]	Drain [28]	LFA [29]	LogReducer
Service $\mathcal{A}$	0.369	0.212	0.508	0.164	**1.000**
Service $\mathcal{B}$	0.266	0.431	0.476	0.187	**0.986**
Service $\mathcal{C}$	0.507	0.343	0.283	0.408	**0.992**

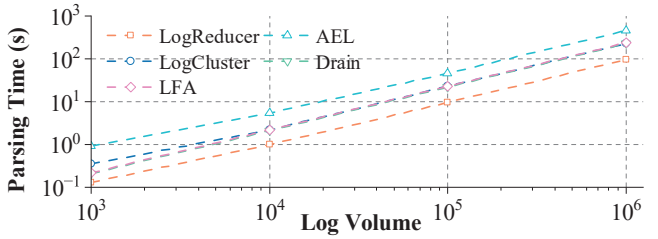


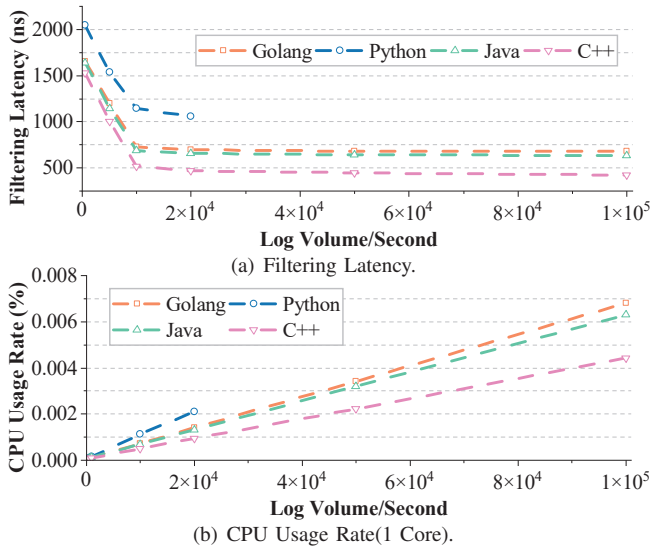
Fig. 13. Log parsing time of different log parsers under different log volumes.

C. Log Filter Evaluation

Filtering log hotspots in kernel space. We now characterize the overhead of filtering log hotspots in the kernel. Figure 14 shows the filtering latency of per log message and CPU usage of our Log Filter when facing the different amount of logs generated by one service instance per second. The overhead in the kernel is measured by bpftool-prog [42], which is a tool for the inspection and simple manipulation of eBPF programs. In our experiments, the benchmark produce at most 100,000 logs per second. But for Python, we only measure its overhead up to 20,000 logs because the logging framework of Python cannot write more than 20,000 logs into log files per second due to its poor performance. Python has poor performance in writing logs because each individual Python's write() calls waits for the full write to complete, whereas other languages returns before the full write to complete.

From Figure 14, we observe that our Log Filter has lightweight overheads on different benchmarks. When filtering log hotspots in kernel space, it only increases latency by up to 2,000 nanoseconds (ns) (i.e., $2 * 10^{-6}$ second) for each log and consumes 0.008% additional CPU utilization of 1 CPU core even when handling 100,000 logs per second. It is noted that Service 16 in Figure 7 suffered up to 3% more latency in the presence of a logging hotspot, while Log Filter increases only 0.001% more latency. In addition, we find that the latency overhead of per log introduced by Log Filter decreases when handling more log messages from 500 to 20,000. The reason is that the internal overhead of executing eBPF instructions is shared amongst log messages. Therefore, when the log volume is not heavy, the average filtering latency decreases. However, when it reaches the limit, the average filtering latency becomes stable, just as shown in Figure 14. (a).

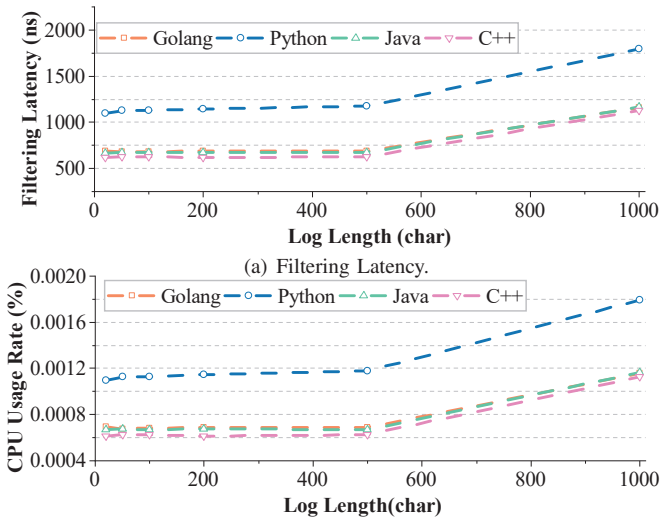
Figure 15 shows the overhead of Log Filter when filtering log hotspots in kernel space under 10,000 logs with 20 chars to 1000 chars lengths. As shown in Figure 15, When the log length is less than 500, the time and CPU usage for filtering logs in kernel space is almost unaffected by the log



(a) Filtering Latency.

(b) CPU Usage Rate(1 Core).

Fig. 14. The overhead for Log Filter in *LogReducer* when filtering log hotspots in kernel under different log volumes of 20 chars per log length.



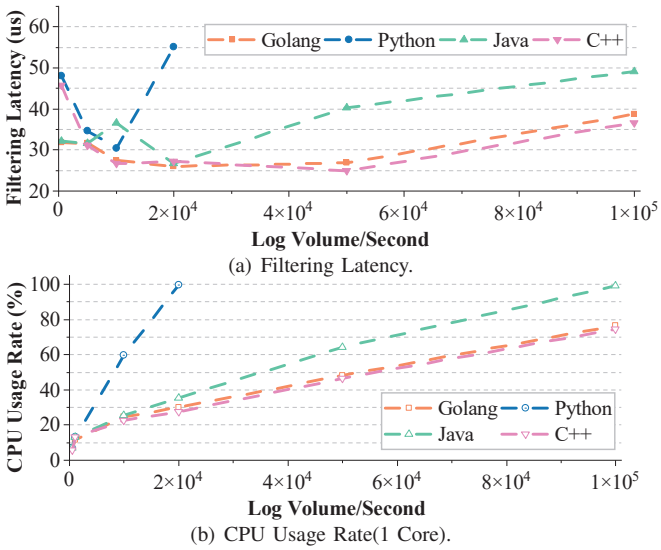
(a) Filtering Latency.

(b) CPU Usage Rate(1 Core).

Fig. 15. The overhead for Log Filter in *LogReducer* when filtering log hotspots in kernel under 10,000 logs with different char lengths.

length. For each log, our Log Filter only increases up to 1,760 nanoseconds latency in python and 1,200 nanosecond latency in other languages, even when the log has 1,000 chars.

Filtering log hotspots in user space. To handle the complex and constrained conditions in the industry (e.g., the missing log signature or the disable of eBPF override configuration), we also provide a Log Filter that can use eBPF to intercept log writing in kernel space and preform complex matching to filter target log hotspots in the user space. Figure 16 shows the filtering latency and CPU usage when filtering log hotspots in user space. As shown in Figure 16, filtering logs in user space takes over $1,000\times$ longer than filtering in kernel space. This is because when filtering hotspots in user space, all raw logs must be copied from kernel space to user space, which is extremely taxing on the time and CPU.



(a) Filtering Latency.

(b) CPU Usage Rate(1 Core).

Fig. 16. The overhead of Log Filter when filtering hotspots in user space.

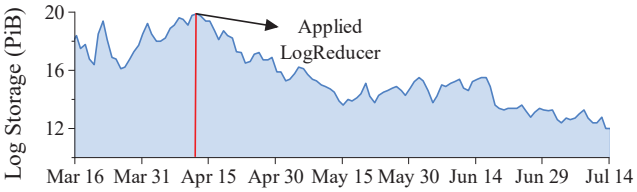


Fig. 17. Changes in the log storage of WeChat from Mar 16 to Jul 14 in 2022.

VI. Usage in Practice

So far, our proposed offline process of *Log Reducer* has already been successfully applied to WeChat and is used by SREs daily both in the production and test environment. Constrained by the lower version of the Linux kernel in WeChat, the online process of *LogReducer* is only used in the test environment of WeChat. As shown in Figure 17, SREs applied our offline process of *LogReducer* to the production environment of WeChat on April 14, 2022. After two months, the overhead of log storage in WeChat dropped from 19.7 PB per day to 12.0 PB (i.e., about a 39.08% decrease). Furthermore, after applying the online process of *LogReducer* in the test environment, the time affected by log hotspots can be reduced from an average of 9 days in production to 10 minutes (i.e., the period run time of *Log Reducer*) in test. In the future, we will continue to promote the adoption of online process *LogReducer* in the production environment of WeChat.

VII. Discussion

Log Hotspots. We conduct a comprehensive study about log hotspots based on the raw logs collected from 20 different services in WeChat. Different excellent groups develop the services and are responsible for different businesses (e.g., login, chats, and short videos). Thus, a big confidence can be obtained regarding the high quality of our study data. As WeChat is deployed globally and serves billions of users, the generalizability of our study in log hotspots can be demonstrated to some extent.

Log Signature. *LogReducer* relies on the log signatures, which uniquely identify a logging statement. In this study, we use the location of the logging statement as the log signature because it is precisely bound to the logging statement and is common in modern systems [32]–[34]. In addition, we find it easy to add the location of the logging statement into the log message based on existing logging frameworks. For example, we only need to turn on a configuration about the location when initializing the logging framework in Golang Zap [43]. We can also extract log signatures based on LogSig [44] if the locations of the logging statements are missing.

eBPF Support. Our eBPF Log Filter is developed based on the newer Linux kernels (at least v4.18). In addition, the Linux kernel should be compiled with $CONFIG\_BPF\_KPROBE\_OVERRIDE = y$ configuration option [45], which is enabled by default at ArchLinux 5.0.7 [46], to allow eBPF to override the execution of a probed function. If the configuration is not enabled, we can use eBPF to intercept log writing in the Linux kernel space and filter log hotspots in the user space.

VIII. Related Work

Where to Log. Prior studies propose approaches to suggest where developers should add logging statements during the development phase [10]–[13], [22], [47], [48]. Errlog [10], Log20 [13], and LogEnhancer [47] insert additional logging statements into the source code to maximize the debugging capability of logging. In contrast, Log2 [22] and Log4Perf [48] proactively insert logging statements into the source code for performance monitoring and diagnosis.

Fu et al. [11] study the logging practices in two industrial software projects. They investigate what categories of code snippets (e.g., exception catch blocks) are logged. LogAdvisor [12] and SmartLog [49] extract contextual features of a code snippet and learn statistical models to suggest whether a logging statement should be added to such a code snippet. However, a significant issue of the above approaches is that they mainly focus on the location of logs during the development phase and cannot reduce log hotspots at program runtime.

Log Compression. After collecting logs during runtime, archiving massive volumes of logs over long periods can introduce expensive overhead. A series of studies have focused on log compression to reduce storage overhead. MLC [50] and Hassan et al. [51] split raw log messages into distinct blocks and compress each block in parallel. Nanolog [14], CLP [16] and Cowic [52] construct a dictionary for the fields in logs and replace the strings by referring to the dictionary. LogZip [15] and RoughLogs [53] achieve log compression by building complex statistical models to identify possible redundancy in logs. Nevertheless, an important drawback of log compression approaches is that they cannot prevent the overhead of writing to the disk and sending logs to the log database.

Log Parsing. To implement log parsing, a straightforward approach is to manually design regular expressions based on raw logs, but it suffers from the low scalability [54]. To overcome the above shortcoming, some data-driven log parsers [18], [26], [27], [29], [30], [55], [56] have been proposed. LogCluster [26], LFA [29] and Logram [55] build frequent itemsets based on tokens and grouped log messages into several clusters to extract log templates. Swisslog [18], LenMa [30], and LogMine [56] cluster similar logs and identify the common tokens shared within each cluster as its template. Drain [28] represents log messages as fixed-depth trees and extracts common log templates based on the trees.

eBPF. Although eBPF is a relatively new Linux kernel feature, eBPF has been widely adopted in many domains. Kmon [36] and Liu et al. [57] introduce a non-intrusive application observability analysis system based on eBPF. BASTION [38] is a new high-performance security enforcement network stack that extends the container hosting platform with an intelligent container-aware communication sandbox. Cilium [58] uses eBPF as a foundation to offer eBPF-backed networking, observability, and security platform on Kubernetes. Syrup [37] and Katran [59] build high-performance load balancing planes based on eBPF. BMC [35] exploits eBPF to create an in-kernel cache for Memcached that serves requests before the execution of the standard network stack.

IX. Conclusion

We conduct a comprehensive study about log hotspots in WeChat, which motivates us to localize log hotspots automatically and reduce them on the fly. We propose *LogReducer*, a non-intrusive and language-independent log reduction framework based on eBPF. After two months of serving the offline process of *LogReducer* in WeChat, the log storage overhead has dropped from 19.7 PB per day to 12.0 PB. Practical implementation and experimental evaluations in the test environment demonstrate that the online process of *LogReducer* can control the logging overhead of hotspots while preserving logging effectiveness. Moreover, the log hotspot handling time can be reduced from an average of 9 days in production to 10 minutes in test with the help of *LogReducer*.

X. Data availability

The log benchmarks in the experiments and eBPF-based Log Filter are available at [60].

Acknowledgment

We greatly appreciate the insightful feedback from the anonymous reviewers. We thank all participants in the interviews for their analysis and responses to log hotspots. The research is supported by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No.62272495), the Basic and Applied Basic Research of Guangzhou (No. 202002030328), the Guangdong Basic and Applied Basic Research Foundation (No. 2018B030312002), and sponsored by Tencent Rhino-Bird Research Elite Program and CCF-Lenovo Blue Ocean Research Fund and the Fundamental Research Funds for the Central Universities,Sun Yat-sen University(No. 22qntd1004). The corresponding author is Pengfei Chen.

REFERENCES

[1] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *ICWS 2019*. IEEE, 2019, pp. 68–75.

[2] Z. Huang, P. Chen, G. Yu, H. Chen, and Z. Zheng, "Sieve: Attention-based sampling of end-to-end trace data in distributed microservice systems," in *ICWS 2021*. IEEE, 2021, pp. 436–446.

[3] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach," *IEEE TCC*, 2020.

[4] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *WWW 2021*. ACM, 2021, p. 3087–3098.

[5] G. Yu, Z. Huang, and P. Chen, "Tracerank: Abnormal service localization with dis-aggregated end-to-end tracing data in cloud native systems," *Journal of Software: Evolution and Process*, p. e2413, 2021.

[6] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust anomaly detection and localization for interleaved unstructured logs," *IEEE TPDS*, pp. 1–1, 2022.

[7] S. He, Q. Lin, J. Lou, H. Zhang, M. R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *ESEC/FSE 2018*. ACM, 2018, pp. 60–70.

[8] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *ICSE 2019*. IEEE / ACM, 2019, pp. 140–151.

[9] C. M. Rosenberg and L. Moonen, "Spectrum-based log diagnosis," in *ESEM 2020*. ACM, 2020, pp. 18:1–18:12.

[10] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging," in *OSDI 2012*. USENIX Association, 2012, pp. 293–306.

[11] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *ICSE 2014*. ACM, 2014, pp. 24–33.

[12] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *ICSE 2015*. IEEE, 2015, pp. 415–425.

[13] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *SOSP, 2017*. ACM, 2017, pp. 565–581.

[14] S. Yang, S. J. Park, and J. K. Ousterhout, "Nanolog: A nanosecond scale logging system," in *USENIX ATC 2018*. USENIX Association, 2018, pp. 335–350.

[15] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: Extracting hidden structures via iterative clustering for log compression," in *ASE 2019*. IEEE, 2019, pp. 863–873.

[16] K. Rodrigues, Y. Luo, and D. Yuan, "CLP: efficient and scalable search on compressed text logs," in *OSDI 2021*. USENIX Association, 2021, pp. 183–198.

[17] eBPF, "ebpf," https://ebpf.io/, 2022, accessed June 6, 2022.

[18] X. Li, P. Chen, L. Jing, Z. He, and G. Yu, "Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults," in *ISSRE 2020*, 2020, pp. 92–103.

[19] Grafana, "Promtail," https://grafana.com/docs/loki/latest/clients/promtail/, 2022, accessed June 6, 2022.

[20] ——, "Loki," https://github.com/grafana/loki, 2022, accessed June 6, 2022.

[21] ClickHouse, "Clickhouse," https://github.com/ClickHouse/ClickHouse, 2022, accessed June 6, 2022.

[22] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *USENIX ATC 2015*. USENIX Association, 2015, pp. 139–150.

[23] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *TSE*, vol. 47, no. 12, pp. 2858–2873, 2021.

[24] 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.

[25] Z. Li, H. Li, T. P. Chen, and W. Shang, "Deeplv: Suggesting log levels using ordinal based neural networks," in *ICSE 2021*. IEEE, 2021, pp. 1461–1472.

[26] R. Vaarandi and M. Pihelgas, "Logcluster - A data clustering and pattern mining algorithm for event logs," in *CNSM 2015*. IEEE, 2015, pp. 1–7.

[27] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *J. Softw. Maintenance Res. Pract.*, vol. 20, no. 4, pp. 249–267, 2008.

[28] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *ICWS 2017*. IEEE, 2017, pp. 33–40.

[29] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *MSR 2010*. IEEE, 2010, pp. 114–117.

[30] K. Shima, "Length matters: Clustering system log messages using length of words," *CoRR*, vol. abs/1611.03213, 2016. [Online]. Available: http://arxiv.org/abs/1611.03213

[31] Kubernetes, "Example of code location of log in kubernetes controller," https://github.com/kubernetes/kubernetes/blob/ea0764452222146c47ec826977f49d7001b0ea8c/staging/src/k8s.io/kube-aggregator/pkg/controllers/openapi/controller.go#L107, 2021, accessed June 6, 2022.

[32] ——, "Kubernetes," https://kubernetes.io/, 2022, accessed June 6, 2022.

[33] Prometheus, "Prometheus," https://prometheus.io/, 2022, accessed June 6, 2022.

[34] T. Cloud, "Tencent cloud," https://intl.cloud.tencent.com/, 2022, accessed June 6, 2022.

[35] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: accelerating memcached using safe in-kernel caching and pre-stack processing," in *NSDI 2021*. USENIX Association, 2021, pp. 487–501.

[36] T. Weng, W. Yang, G. Yu, P. Chen, J. Cui, and C. Zhang, "Kmon: An in-kernel transparent monitoring system for microservice systems with ebpf," in *CloudIntelligence 2021*. IEEE, 2021, pp. 25–30.

[37] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-defined scheduling across the stack," in *SOSP 2021*. ACM, 2021, pp. 605–620.

[38] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *USENIX ATC 2020*. USENIX Association, 2020, pp. 81–95.

[39] iovisor, "Bcc - tools for bpf-based linux io analysis, networking, monitoring, and more," https://github.com/iovisor/bcc, 2022, accessed June 6, 2022.

[40] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *HPSR 2018*. IEEE, 2018, pp. 1–8.

[41] Y. Liu, X. Zhang, S. He, H. Zhang, L. Li, Y. Kang, Y. Xu, M. Ma, Q. Lin, Y. Dang, S. Rajmohan, and D. Zhang, "Uniparser: A unified log parser for heterogeneous log data," in *WWW 2022*. ACM, 2022, pp. 1893–1901.

[42] Ubuntu, "Bpftool-prog: a tool for inspection and simple manipulation of ebpf progs," https://manpages.ubuntu.com/manpages/focal/en/man8/bpftool-prog.8.html, 2022, accessed June 6, 2022.

[43] Uber, "Zap - blazing fast, structured, leveled logging in go," https://github.com/uber-go/zap, 2022, accessed June 6, 2022.

[44] L. Tang, T. Li, and C.-S. Perng, "Logsig: generating system events from raw textual logs," in *CIKM 2011*. ACM, 2011, pp. 785–794.
[45] Linux, "Config_bpf_kprobe_override: Enable bpf programs to override a kprobed function," https://www.kernelconfig.io/config_bpf_kprobe_override, 2022, accessed June 6, 2022.
[46] A. Linux, "Arch linux enable config_bpf_kprobe_override by default," https://bugs.archlinux.org/task/62384, 2022, accessed June 6, 2022.
[47] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS 2011*. ACM, 2011, pp. 3–14.
[48] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems' performance monitoring," in *ICPE 2018*. ACM, 2018, pp. 127–138.
[49] Z. Jia, S. Li, X. Liu, X. Liao, and Y. Liu, "SMARTLOG: place error log statement by deep understanding of log intention," in *SANER 2018*. IEEE, 2018, pp. 61–71.
[50] B. Feng, C. Wu, and J. Li, "MLC: an efficient multi-level log compression method for cloud backup systems," in *IEEE Trustcom/BigDataSE/ISPA 2016*. IEEE, 2016, pp. 1358–1365.
[51] A. Hassan, D. Martin, P. Flora, P. Mansfield, and D. Dietz, "An industrial case study of customizing operational profiles using log compression," in *ICSE 2008*, 2008, pp. 713–723.
[52] H. Lin, J. Zhou, B. Yao, M. Guo, and J. Li, "Cowic: A column-wise independent compression for log stream analysis," in *CCGrid 2015*. IEEE, 2015, pp. 21–30.
[53] M. Meinig, P. Tröger, and C. Meinel, "Rough logs: A data reduction approach for log files," in *ICEIS 2019*. SciTePress, 2019, pp. 295–302.
[54] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *ICSE (SEIP)*. IEEE / ACM, 2019, pp. 121–130.
[55] H. Dai, H. Li, C. Chen, W. Shang, and T. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *TSE*, vol. 48, no. 3, pp. 879–892, 2022.
[56] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *CIKM 2016*. ACM, 2016, pp. 1573–1582.
[57] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on ebpf," in *ICPADS 2020*. IEEE, 2020, pp. 697–702.
[58] Cilium, "ebpf-based networking, observability, security," https://cilium.io/, 2022, accessed June 6, 2022.
[59] FaceBook, "Katran:a high performance layer 4 load balance," https://github.com/facebookincubator/katran, 2022, accessed June 6, 2022.
[60] L. Reducer, "The log filter and benchmark of log reducer," https://github.com/IntelligentDDS/LogReducer, 2022, accessed June 6, 2022.