*Log*²*:* A Cost-Aware Logging Mechanism for Performance Diagnosis

Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, and Qingwei Lin, *Microsoft Research;* Qiang Fu, *Microsoft;* Dongmei Zhang, *Microsoft Research;* Tao Xie, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/atc15/technical-session/presentation/ding

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Log^2: A Cost-Aware Logging Mechanism for Performance Diagnosis

Rui Ding[1], Hucheng Zhou[1], Jian-Guang Lou[1], Hongyu Zhang[1], Qingwei Lin[1],
Qiang Fu[2], Dongmei Zhang[1], Tao Xie[3]
[1]*Microsoft Research*
[2]*Microsoft*
[3]*University of Illinois at Urbana-Champaign*

Abstract

Logging has been a common practice for monitoring and diagnosing performance issues. However, logging comes at a cost, especially for large-scale online service systems. First, the overhead incurred by intensive logging is non-negligible. Second, it is costly to diagnose a performance issue if there are a tremendous amount of redundant logs. Therefore, we believe that it is important to limit the overhead incurred by logging, without sacrificing the logging effectiveness. In this paper we propose *Log*^{2}, a cost-aware logging mechanism. Given a "budget" (defined as the maximum volume of logs allowed to be output in a time interval), *Log*^{2} makes the "whether to log" decision through a two-phase filtering mechanism. In the first phase, a large number of irrelevant logs are discarded efficiently. In the second phase, useful logs are cached and output while complying with logging budget. In this way, *Log*^{2} keeps the useful logs and discards the less useful ones. We have implemented *Log*^{2} and evaluated it on an open source system as well as a real-world online service system from Microsoft. The experimental results show that *Log*^{2} can control logging overhead while preserving logging effectiveness.

1 Introduction

Logging has been commonly adopted for monitoring and diagnosing performance issues of online service systems, such as web search engines and online banking systems. Typically, performance logs record the end-to-end execution time of a service request as well as the execution time of a component of the service system. Logging is usually achieved by instrumenting source code with logging statements and the resultant logs are stored on disks. In practice, performance logs constitute a large proportion of total logs. For example, our study of a Microsoft online service system (described in Section 6) shows that around 20%-40% of the total logs are performance logs.

Although logging is effective for performance diagnosis, it comes at a cost. Logging introduces overhead, such as disk I/O bandwidth as well as CPU and memory consumption. Intensive logging could further interfere with the service's normal execution. For example, web search engines are sensitive to performance interference from the logging system, which tends to generate huge volume of logs. Empirical results [20] show that if logging is fully conducted, the average execution time of requests in a search engine could increase by 16.3% and the average throughput could decrease by 1.48%. Therefore, it is critical to reduce the performance interference by reducing the logging overhead. In addition, our survey (See Section 2 for more details) of Microsoft engineers confirms this finding. About 80% of the survey participants confirmed that they had experienced non-negligible performance overhead caused by logging. Furthermore, intensive logging could introduce a large amount of less "useful" logs (i.e., the logs that are not useful for helping diagnose the performance issue under investigation). A study [9] on one large-scale online service system in Microsoft indicates that a high proportion of logs are useless for diagnostic purposes. Our survey of Microsoft engineers also confirms this observation.

Existing techniques for reducing logging overhead include manually removing some logging statements, changing the logging level (e.g., from "Verbose" to "Medium"), and outputting logs in a sampling fashion [20][6]. These techniques aim to reduce the number of logs to be output. However, these techniques are insufficient for several reasons. First, they cannot guarantee to preserve logging effectiveness (i.e., preserving the useful logs for diagnosis purposes). For instance, the sampling technique could miss important events due to randomness of the sampling. Second, there is no control mechanism on "whether to log" (whether or not the executed logging statement should be output) over the existing logging systems. Therefore, once developers decide "where to log", the logging system must strictly output the logs after the execution of the placed logging statements. The resultant logs could still contain many useless ones. Finally, most of these existing techniques do not consider the dynamic properties of a running system. For a running system, the changes of workload and throughput can influence the load of its logging system. Simply using a single logging level or a sampling rate may not be able to control the logging overhead during workload spikes. Therefore, it is desirable to have a new, overhead-constrained logging system for performance diagnosis.

In this paper, we propose a cost-aware logging mechanism called *Log*^{2}. Using *Log*^{2}, developers predefine a resource *budget* allowed for logging. At runtime, the logging system decides "whether to log" such that the logging overhead is constrained under the budget while the logging effectiveness is maximized. The budget for logging overhead is defined as *logging bandwidth*, which is the maximum volume of logs allowed to be output in a time interval (such as 1KB per second). There are two reasons for choosing logging bandwidth as the budget. First, according to our survey, I/O bandwidth is the most concerning overhead in practice. Second, in general, most logging overhead such as disk storage, network I/O and CPU are directly or indirectly affected by I/O bandwidth. The logging effectiveness is measured as the percentage of performance issues that can be captured by the resultant logs.

There are three challenges for realizing such a cost-aware logging mechanism:

- It should be able to control logging overhead while preserving logging effectiveness.
- It should incur low additional overhead such as CPU and memory consumption.
- It should provide flexibility for developers to configure it for different service scenarios, and should be able to adapt to environmental changes dynamically.

To address the above challenges, *Log*^{2} introduces a two-phase filtering mechanism. In the first phase, a large number of irrelevant logs are discarded efficiently. In the second phase, useful logs are cached and output while complying with the logging budget. The two-phase mechanism is updated dynamically to address all the challenges.

We evaluate *Log*^{2} on BlogEngine, which is a popular open source blogging platform. Furthermore, we perform an evaluation of *Log*^{2} using real logs of ServiceX, which is a large-scale online service system from Microsoft. The evaluation results confirm that *Log*^{2} is effective and practical in real-world scenarios.

This paper makes the following main contributions:

- We propose a novel cost-aware logging mechanism *Log*^{2}, which helps achieve a balance between logging overhead and effectiveness. Such a mechanism incurs low additional overhead and is flexible.
- We design and implement *Log*^{2}. We also evaluate *Log*^{2} on both a open source system and a large-scale online service system from Microsoft.

The rest of the paper is organized as follows. Section 2 describes a survey of logging practice in Microsoft, which motivates the design goals of *Log*^{2} described in Section 3. Section 4 describes the design and implementation of *Log*^{2}. Section 5 provides the detailed evaluation of *Log*^{2} on an open source system. Section 6 describes a case study on Microsoft ServiceX system. We discuss the limitations and future work in Section 7. Section 8 introduces the related work, and Section 9 concludes the paper.

2 A Survey of Logging Practice in Microsoft

To better understand the current logging practice, we conducted a comprehensive survey among hundreds of engineers from five product teams in Microsoft. We received responses from 84 engineers. According to the survey, 81 out of 84 respondents are "expert" or "knowledgeable" to logging systems. The survey aims to understand the participants' experience in logging systems and logging overhead. The details of survey questions are available online [4].

In general, the logging systems used by Microsoft engineers fall into three categories, including (1) internally developed systems that directly output the executed logging statements via a language-intrinsic component or a wrapped API; (2) ETW logging [2], which writes the buffered logs in a batch fashion, and (3) sampling-based logging tools that are mainly designed for large-scale online services sensitive to logging overhead.

2.1 Logging Overhead

According to our survey, 80% of the participants agreed that logging overhead is a non-negligible issue. The top three most commonly concerned types of overhead are storage (60%), I/O bandwidth (58%), and CPU usage (56%). Among the participants, 59% of them have suffered from the consequences incurred by the logging overhead. Table 1 shows some of the experiences reported by the surveyed engineers.

The top three most widely used approaches to control the logging overhead include adjusting the logging level (93%), manually removing unnecessary logs (64%), and

Table 1: Some of the experiences of the logging overhead

Category	Reported Experiences
Disk I/O bandwidth	Overuse of I/O *caused perception of interference with core functionality.* The bandwidth requirement by *enabling all logs is 8MB/s, which however should be $\leq$ 200KB/s.*
Storage	*OS slows down*, other process that needs disk space *may crash and even logging system could crash.* Storage is a *critical component that may cause system crash, but it is often overlooked.*
CPU	*Service is slowed down significantly once the CPU usage of logging is increased to double digits.* CPU usage of logging is *very sensitive to our super-efficient system.* *3%-5% is the upper bound for CPU usage of logging.*
Memory	Unexpected increases of memory usage of logging system *was the root cause of one service incident.* Memory leak of logging system *caused days of efforts on debugging.*

archiving log files periodically (43%). However, about 65% of the participants replied that they are not satisfied with the existing approaches. For instance, removing logs by changing source code requires extra efforts on re-compiling, testing, and re-deployment. Archiving log files is often expensive because a large volume of data needs to be transformed via network. All these existing approaches are considered to be after-thoughts, and are applied only when logging overhead starts to compromise the system quality.

About 83% of the survey participants also agreed that many log messages are redundant for diagnosing performance issues, implying the feasibility to reduce logging overhead while preserving sufficient logging effectiveness. In addition, about 43% of all participants agreed that logging overhead needs to be controlled, and they considered resource budget for logging in their work.

2.2 Other Limitations of Existing Logging Systems

A number of participants also shared with us additional limitations of the existing logging systems and expressed the needs for a cost-aware logging mechanism. These comments and suggestions strongly motivated the design of *Log*^{2}:

Lack of cost-awareness during log instrumentation. One participant complained about the lack of cost-awareness during log instrumentation. He noticed that some developers often had little idea about the resulting logging overhead when they planned to instrument source code with new logging statements. A typical bad logging practice is to insert logging statements in tight loops (i.e., the loops which iterates intensively), which could cause high overhead, especially in I/O throughput and storage. He suggested a logging system for controlling the logging overhead transparently, so that developers can perform log instrumentation without worrying about the overhead incurred.

Burden in log analysis. One participant commented that too many logs make it challenging to analyze logs via manual inspection. It would be helpful if a logging system can collect all possible logs but do not flush all of them. He also suggested a potential solution: logging system should flush the logs only when some predefined rules are violated.

In summary, the survey results motivate a new overhead-constrained logging mechanism as we propose in this paper.

3 The Design Goals of *Log*^{2}

3.1 Cost-Aware Logging Mechanism

In this paper, we propose *Log*^{2}, a cost-aware logging mechanism that constraints logging overhead. Using this mechanism, developer can perform logging by instrumenting their programs, and predefine a resource *budget* for logging. With the given *budget*, the logging mechanism decides "whether to log" for each logging request at runtime, makes sure that the logging overhead complies with the predefined *budget*, and maximizes the logging effectiveness at the same time. In addition, the logging mechanism can support on-the-fly budget setting. Therefore, the logging mechanism not only provides developers with the flexibility to strike the balance between logging overhead and effectiveness, but also provides the flexibility to configure different logging budgets for different service scenarios, or even the flexibility to dynamically configure the logging budget. Furthermore, such a cost-aware logging mechanism enables better planning of maintenance resources [5], as the logging budget can be determined in advance.

3.2 Design Goals

Log^2 is designed to realize such a cost-aware logging mechanism. The budget for logging overhead in *Log*^{2} is defined as *logging bandwidth*, which is the maximum volume of logs allowed to be output in a time interval. Logging bandwidth is the most concerning logging overhead according to engineers' feedback. It is also the most representative logging overhead, because other types of logging overhead such as disk storage, network I/O and CPU are often directly or indirectly affected by the logging bandwidth.

We have identified four design goals for *Log*^{2}, which are listed below:

Cost-effectiveness: *Log*^{2} should be able to achieve an optimal balance between logging overhead and effectiveness. The logging overhead, defined in terms of log bandwidth, should be constrained under the budget. Although the logging budget is under constraint, logging effectiveness cannot be compromised, i.e., with respect to performance diagnosis, the number of performance issues detected by the reduced number of logs should be similar to the number of issues detected by the total number of logs. In *Log*^{2}, a ranking score named *utility score* is defined to measure how much utility each logging request contributes to performance diagnosis. *Log*^{2} then selects the top-ranked logging requests and outputs them. Other logging requests are filtered away. More details are described in Section 4.3.

Low additional overhead. *Log*^{2} should incur low additional overhead. The additional overhead brought by runtime decision on "whether to log" (i.e., CPU usage and memory consumption) should be negligible. The design choices of *Log*^{2} for minimizing CPU usage and memory consumption are described in detail in Section 4.4.

Scalable. *Log*^{2} should be scalable to the number of logging requests. It is very common that thousands of requests are processed per second, and considering that many logging statements are executed when serving one single request, the scale of the *logging requests* per second is large. A traditional logging system, which makes centralized decision, suffers since such centralized decision can delay the logging time as well as increasing the corresponding memory buffer usage. In contrast, *Log*^{2} includes a two-phase filtering design to avoid the potential bottleneck. The details are described in Section 4.

Flexible. *Log*^{2} should provide developers with the flexibility to configure the system. First, *Log*^{2} provides several types of predefined *utility scores*, which are designed for the most common diagnostic scenarios (to be described in Section 4.3.1). It also allows developers to configure a user-defined function for computing utility scores. Such flexibility enables *Log*^{2} to tackle various types of performance issues. Second, the budget can be configured on-the-fly. Such on-the-fly configuration enables developers to select a proper *logging bandwidth* according to the different resource plans in different scenarios. Since there is no one-fit-for-all configuration for all kinds of services, such flexibility is crucially important for wide adoption in different scenarios. More details are described in Section 4.3.2 and Section 4.4.2.

```
Log2.Begin(string McrName, ...); //begin
DoSomething();
Log2.End(string McrName, ...);   //end
```

Figure 1: Logging API in *Log*^{2}.

4 Design and Implementation of *Log*^{2}

This section illustrates the detailed design and implementation of *Log*^{2}. We first discuss the high level workflow of *Log*^{2}, and then illustrate its two core components, namely local filter and global filter. These core components are essential for achieving the goals of *Log*^{2}.

4.1 Logging Requests

For performance diagnosis, developers can specify an area of code that should be monitored and logged. We call such an area of code Monitored Code Region (MCR). Examples of typical MCR include:

- Expensive system-level APIs, such as operations on I/O, database, networking, etc.
- Loop blocks. Previous work [13] found that a significant portion of real-world performance issues are caused by inefficient loops.
- Function calls cross application-level component boundaries, such as RPC or the connection between GUI and backend services.

Performance logs should record two timestamps at the beginning and end of a MCR, which are sufficient to compute the execution time of the MCR. *Log*^{2} provides two logging APIs, *Begin* and *End*, to denote the beginning and end of an MCR, respectively. The APIs compute the execution time of an MCR and also record the unique ID of the MCR. Figure 1 depicts the logging API usage in *Log*^{2}, where the execution time of *DoSomething* is recorded. A pair of logs *Begin* and *End* form a *logging request*, which will be further processed by *Log*^{2} to decide whether they should be filtered or output.

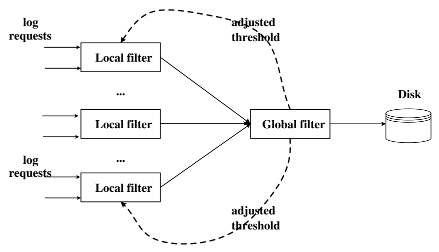

Figure 2: The workflow of *Log*^{2}.

4.2 Overall Workflow

The workflow of Log^2 is depicted in Figure 2. Two filtering phases, local filter and global filter, are adopted to decide whether or not the incoming logging requests should be logged (whether to log). Such a two-phase filtering mechanism is used to avoid the potential bottleneck of a single centralized filter, when a huge number of log requests come in simultaneously. The local filters are responsible for discarding the trivial logging requests, which are logging requests that have low utility scores. The global filter is responsible for flushing the top ranked logging requests to disk and in the meantime complying with the logging budget.

Each thread of logging requests has a local filter. Only the logging requests with *utility scores* (which are calculated dynamically) higher than a global threshold can pass through the local filter to a memory buffer in the global filter. Other logging requests are discarded.

The global threshold is adjusted dynamically, to adapt to environment dynamics, while optimizing the effectiveness and efficiency of Log^2. Usually, a significant high portion of logging requests are discarded in the first phase. In the global filter, the final decision on log outputting is made periodically to make sure that the budget constraint is compliant. The logging requests from all local filters during the last time window are cached in memory. When a periodic event is triggered, the cached logging requests are sorted according to their utility scores. Only the top-ranked requests with total volume equal to the logging budget are flushed to disk. Meanwhile, the global threshold for utility scores is updated by the global filter by considering the volume of logging requests in recent time intervals. Lastly, the global filter feeds the new threshold back to each local filter.

Details about each component are described in the following subsections.

4.3 Local Filter

The major task of the local filter component is to compute the utility score for each logging request. The utility score measures the usefulness of a logging request for performance diagnosis. Note that a local filter is executed in the same service thread being monitored. The overhead for computing utility score should be kept low to reduce the impact on the service.

4.3.1 Formula of utility score

To compute the utility score for each logging request, we analyze the histogram of the execution time of the corresponding MCR. The intuition is that the utility score should be higher if the execution time of a MCR deviates further away from its past behavior. For each MCR, we can measure the degree of performance deviation based on the histogram of the execution time of the MCR. However, it is inefficient to maintain the complete history of execution time for each MCR and compute the histogram. In our work, we adopt the concept of *method of moments* [15], which can be efficiently computed. According to statistical theory, *moments* can well approximate histogram [10]. The 1-order of *moment* is mean, and the 2-order of *moment* (σ^2) is the square of *standard deviation* (σ).

Based on the *mean* (μ) and the *standard deviation* (σ) of execution time of an MCR, we propose three forms of utility scores, given the current execution time t of the MCR:

$$utility = \frac{t - \mu - \tau}{\sigma} \tag{1}$$

$$utility = t \tag{2}$$

$$utility = t - \mu - \tau \tag{3}$$

In Equation (1), a constant value τ is a tolerance factor, which is used to further reduce false-positives for MCRs. For example, execution time of $5ms$ is significantly abnormal compared to $1ms$ as the average execution time, but is ignorable for performance diagnosis. The default value of τ is $25ms$.

Equation (2) simply uses the execution time as the utility score, which is suitable when the users would like to identify performance hotspots (e.g., those components with the longest execution time). Equation (3) computes utility score based on the mean execution time. Compared with Equations (1) and (2), it considers the abnormality (t-μ) while ignores the fluctuation.

Besides the predefined utility formulas, we also allow users to specify their own utility functions to cater for their own scenarios.

4.3.2 Updating the utility scores dynamically

During performance monitoring, the execution time t of each MCR varies at runtime. Therefore, the *mean* and *standard deviation* of t should be updated dynamically over time. *moments* can be updated incrementally, with the time complexity of $O(1)$:

$$\mu_n = (1 - \frac{1}{n})\mu_{n-1} + \frac{1}{n}t_n \tag{4}$$

$$\sigma_n^2 = (1 - \frac{1}{n})[\sigma_{n-1}^2 + \frac{1}{n}(t_n - \mu_{n-1})^2] \tag{5}$$

where n denotes the n^{th} update; t_n is the n^{th} execution time.

We also modify the Equations (4) and (5) in a manner similar to *Exponential Smoothing*[11]. Exponential Smoothing can better capture the slow-varying system dynamics. The corresponding formulas are as follows:

$$\mu_n = (1 - \alpha)\mu_{n-1} + \alpha t_n \tag{6}$$

$$\sigma_n^2 = (1-\alpha)[\sigma_{n-1}^2 + \alpha(t_n - \mu_{n-1})^2] \quad (7)$$

where α is a weighting factor, which is empirically set to 0.01.

4.4 Global Filter

In *Log*^{2}, the global filter component performs two major tasks: log flushing and utility-threshold adjusting.

4.4.1 Log flushing

Log flushing is triggered periodically, and such period is called *flush interval*. When the timer is triggered, *Log*^{2} first sorts the buffered logs according to the utility score, and then flushes the top ranked logs so that the total flushed log volume does not exceed the logging budget. All selected logs are packed together and are flushed once in a batched fashion.

Buffer design. Proper buffer design is important for reducing logging overhead, especially for reducing CPU usage. Note that the buffer will be accessed by multiple local filters with fast inserting operation, as well as the global filter thread with slow sorting and flushing operations. To make sure that the latter one does not affect the inserting performance and thus does not block working threads, *Log*^{2} includes a data structure called *swap buffer*, which has two buffers: one serves for inserting operation, and the other serves for sorting and flushing operations. These two buffers are swapped periodically after a flush interval. A 0/1 flag is used to indicate which buffer is currently used for insertion, and which one is for flushing. Such mechanism guarantees that the two threads work on different buffers without lock contention except swapping the global flag.

Flush-interval selection. Long flush interval would result in larger swap buffer, and thus more memory consumption; while shorter interval benefits less from batched flushing, and incurs frequent overhead in swapping buffers. *Log*^{2} currently sets the default flush interval to 30 seconds, which works well in our experiments and practice. Users are also allowed to configure the flush interval on-the-fly.

4.4.2 Utility threshold adjustment

The utility threshold is used to control the volume of logs to be inserted into the *swap buffer*. Because only the logging requests with utility scores larger than the threshold is cached, setting a proper threshold is very important for *Log*^{2}. Specifically, if the threshold is set too low, massive logs could be inserted into the *swap buffer*, the consequence is larger overhead. On the other hand, if the threshold is set too high, only a small amount of logs could be cached in the buffer, thus the important logs could be missed, leading to unacceptable logging effectiveness.

The optimal objective is to cache just budget-volume logs by selecting a proper threshold. Choosing such an optimal threshold value in one-shot in unrealistic, because either the environment dynamics or the frequency of different utility scores is unknown. To address this challenge, we design an iterative way for adjusting the threshold by ‘learning from history’. The duration of each iteration is called *adjust interval*. Intuitively, when the volume of logs in the previous *adjust interval* is higher than the budget, then the threshold should be increased. The threshold should be decreased when the volume of logs in the previous *adjust interval* is lower than the budget. From both effectiveness and efficiency perspectives, it is desirable that the adjusting algorithm should converge quickly, and the volume of logs in the buffer should not be too large (low overshoot [19]) in any interval. We next illustrate the details of *Log*^{2}’s threshold-adjustment algorithm, which is agile and has low overshoot.

Adjustment mechanism. Let us denote the threshold and log volume as T_n and V_n, respectively. Here n is the index of the *adjust interval*. Let us denote B as the logging budget. The threshold adjusting mechanism used in *Log*^{2} is as follows (in the form of Secant Method [18]):

$$T_n = T_{n-1} + (V_{n-1} - B) \times \frac{T_{n-1} - T_{n-2}}{V_{n-1} - V_{n-2}} \quad (8)$$

Mathematically, the convergence of our algorithm is super-linear, with an order of 1.618 [18]. More details about the mathematical deduction of our method are available at our project website [4]. The interpretation is that the ‘gain’ $T_n - T_{n-1}$ on the threshold is proportional to ‘error’ $V_{n-1} - B$, and coefficient $\frac{T_{n-1}-T_{n-2}}{V_{n-1}-V_{n-2}}$ approximates the reciprocal of the derivative, if we treat V as a function of T.

In our implementation, to avoid a divide-by-zero error, we add 1 if $V_{n-1} - V_{n-2}$ is close to 0. When $T_{n-1} - T_{n-2}$ is equal to zero, threshold updating can trap to a certain number and never changes. To avoid such issue, we add a very small value (0.01) under such situation.

Adjustment interval. To make the threshold adjustment mechanism more effective, a properly chosen *adjustment interval* is needed. The adjustment interval should mitigate the fluctuation of environment change, i.e., the workload varies slowly under the granularity of the chosen adjustment interval. Therefore, the adjust interval cannot be too short; otherwise, the transient random variation of workload will be significant, On the other hand, a too long interval indicates longer time for convergence, making *Log*^{2} less agile. In our implementation, *Log*^{2} sets the adjust interval to 30 seconds, which is the same as the *flush interval*.

4.5 Implementation Details

We have implemented *Log*^{2} using the C# language. Some details about the implementation are as follows.

Bounded memory usage. The maximum memory usage of *Log*^{2} is set to 50MB in configuration, so that *Log*^{2} has negligible memory contention with normal service operations. In our implementation, when the maximum memory usage is reached, new *logging requests* will be dropped in the same flushing interval. In fact, 50MB is rarely reached in most cases. Specifically, two components in *Log*^{2} consume most memory usage. One is the cache for maintaining μ and σ for all the MCRs. For a large-scale online service, the number of MCRs is in a magnitude of 100,000, so the corresponding memory usage is 100,000 $\times$ 2 $\times$ 8B = 1.6MB. The other component that consumes most memory usage is the swap buffer. Its size depends on both the budget size and flush interval. The I/O bandwidth of logging is 200KB/s (which is 20GB per day!) per machine for a typical large-scale online service. Because the budget size does not exceed the overall throughput, a much loose upper bound of memory usage on the swap buffer is 200KB/s $\times$ 60s $\times$ 2 = 24MB. In addition, the 50MB threshold has not been reached in all of our experiments.

Handling system idle time. System idling is a special circumstance that needs to be handled. Specifically, when *logging requests* are rare, the budget will not be reached no matter how the utility threshold is adjusted. The consequence is that the utility threshold could become extremely low, and thus the system will overshoot dramatically (i.e., there will be a burst of flushing) when the intensity of *logging requests* turns back to normal. In order to avoid such circumstances, a lower bound on the *adjust interval* is set. In our implementation, we set the lower bound to 0. Such mechanism is commonly used in the area of control engineering [8].

Nested instrumentation. To support nested instrumentation, it is noteworthy that each local filter actually maintains a timestamp stack to match the logging begin-end pair, When a *Begin* is invoked, the corresponding timestamp is pushed into the stack; and when an *End* is invoked, the top element in the stack is popped, and is matched as the *Begin* corresponding to the current *End* invocation. As illustrated in Section 4.3.2, the historical information of each MCR is maintained separately, therefore, dropping the outer log request will not directly lead to the dropping of the inner log request.

5 Evaluation

In our evaluation, we intend to evaluate *Log*^{2} from the following three aspects:

Logging throughput: How much I/O throughput (the volume of logs flushed to disk within a time interval) can be reduced by *Log*^{2}, compared with the existing logging system?

Logging effectiveness: How effective is *Log*^{2} in diagnosing performance issues? The effectiveness is measured as the percentage of performance issues that can be captured by the flushed logs.

Additional overhead: How much additional CPU and memory overhead is incurred by *Log*^{2}?

5.1 Experimental Subject and Setup

To evaluate *Log*^{2}, we design experiments on BlogEngine [1], which is a popular open-source, ASP.NET based blogging platform. BlogEngine has received more than 1,000,000 downloads as of January 30, 2015. It supports various blogging activities, such as writing blogs, adding comments, sharing, and following. We choose the version 2.8, as it is a recent stable version.

To evaluate *Log*^{2} on BlogEngine, we run the BlogEngine as a service, and we simulate concurrent access to the service via multiple synthetic users. We then analyze the logs generated by *Log*^{2} as well as the runtime performance. We set up the experiment on BlogEngine with four steps: instrumentation, deployment, performance issue injection, and overhead monitoring. Below are the detailed setup procedures.

Instrumentation. We perform program instrumentation guided by previous work [14] [13]. Specifically, three types of code regions in BlogEngine are marked as MCRs and logged, since they have relatively high potential to cause performance issues. These three types of MCRs include expensive system-level APIs, loop blocks, and function calls. In summary, about 1000 MCRs are identified and instrumented.

Deployment. We use one physical machine to deploy the BlogEngine service, and two other physical machines are configured as client nodes. Each machine runs Windows Server 2012 R2, with CPU Intel(R) Xeon(R) E5-2650 v2 @ 2.60GHz (2 processors) and 192GB Memory.

We adopt a tool named WebTest [3] to simulate high workload from multiple synthetic users to access the BlogEngine service. WebTest is a new testing tool released with Visual Studio 2012. It can be configured to generate mixed types of requests with user-specified loads. In our experiment, we generate five typical types of requests in WebTest - *read blogs, write comments, search, download files* and *upload files*. These requests cover the most common usage scenarios of BlogEngine.

Performance Issue Injection. In order to evaluate the logging effectiveness of *Log*^{2}, we inject three types of performance issues, namely *upload an extremely large file*, *search a strange term*, and *exhaust CPU by other process*. Specifically, when uploading a file with size larger than 100MB, the GUI on the client side starts to hang (a possible fix is to put the uploading job in a back-end thread). The response to the search operation becomes significantly slow when entering a strange query term that is long and contains special characters (a possible fix is to pre-process the query term). Both of these two performance issues can be directly pinpointed by the corresponding logs.

We write a program named *ResourceEater* to consume high CPU usage in a certain period to mimic the third type of performance issues. When ResourceEater is launched, it occupies CPU intensively. The runtime performance of BlogEngine degrades significantly. Such performance issues can be reflected in the corresponding logs (e.g., the logs that mark the loop blocks).

Overhead monitoring. To measure the I/O throughput, we record the number of logs flushed to disk per time interval. To measure the additional CPU/Memory overhead of Log^2, we write a program named $PerfMonitor$ to periodically monitor the CPU and memory usage of BlogEngine at every second. The CPU overhead is measured as the percentage of total CPU cycles Log^2 occupies, and the memory overhead is measured as the bytes of memory space Log^2 consumes.

5.2 Experimental Design

We design an experiment to evaluate Log^2. We use the WebTest tool [3] to simulate 101 synthetic users concurrently accessing BlogEngine. The experiment runs for two hours. Among the 101 users, 100 users mimic the normal user behaviors, which fall into the five aforementioned groups (*read blogs, write comments, search, download files* and *upload files*). One user mimics the abnormal usage to inject two types of performance issues (*upload an extremely large file* and *search a strange term*), which are generated 78 times during the 2-hour experiment.

To inject the issues caused by *exhausting CPU by other process*, the ResourceEater is triggered on the service machine one hour after start, and lasts for 10 minutes.

We also evaluate the logging effectiveness of Log^2 using three utility scores: t, $(t-\mu-\tau)$, $(t-\mu-\tau)/\sigma$, respectively.

In the experiment, we compare Log^2 with the baseline approach, which directly outputs all executed logs without considering cost-effectiveness. As we instrument all the interested MCRs, the baseline approach is able to detect all injected performance issues. We are interested in knowing how Log^2 can detect similar number of issues using fewer amount of logs.

In addition, we compare Log^2 with two sampling-based logging approaches, named Sampling-counter and Sampling-time, respectively. Sampling-counter is counter-based, which uses a global counter to record how many logging requests are processed. Only the logs whose corresponding counter is divisible by the reciprocal of the sampling rate are flushed to disk. Sampling-time is time-interval based, which uses a timer to control when the logs are flushed to disk. Only the logs executed when the timer is triggered are flushed.

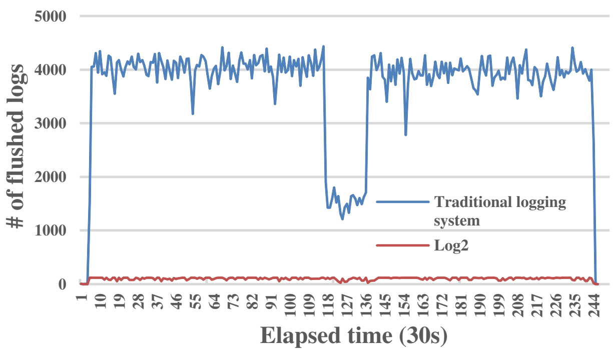

Figure 3: Comparison of logging throughput. (budget = 120 logs/interval)

5.3 Experimental Results

Logging throughput. Figure 3 shows the number of logs flushed per time interval (30s) using Log^2 and the baseline logging approach, respectively. The budget is set to 120 logs/interval. The big drop on the number of logging requests (around interval 118-136) is due to the launching of *ResourceEater*.

Figure 3 shows that the logging throughput is significantly reduced using Log^2. The average number of logs flushed per interval is 104 for Log^2, while it is 3,800 for the baseline logging approach. The reduction on logging throughput is over 97%. In addition, the logging throughput of Log^2 strictly complies with the budget constraint ($<$ 120 logs/interval).

Logging effectiveness. The logging effectiveness is inherently associated with the budget size, i.e., the logging bandwidth. Higher logging bandwidth would induce higher logging effectiveness. We evaluate the logging effectiveness by varying the budget size. In addition, we also evaluate three alternative formulas of utility scores (t, $t-\mu-\tau$, and $(t-\mu-\tau)/\sigma$).

Figure 4 illustrates how the logging effectiveness increases as the budget size increases. All the three proposed utility scores help achieve high effectiveness, i.e., the coverage of marked logs increases quickly to almost 100% when the budget size starts to increase. The results indicate that Log^2 has strong ability to preserve high logging effectiveness while reducing a significant amount of logs.

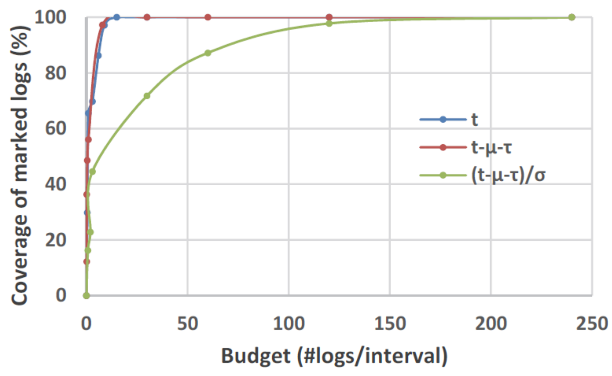

Figure 4: Logging effectiveness vs. budget size

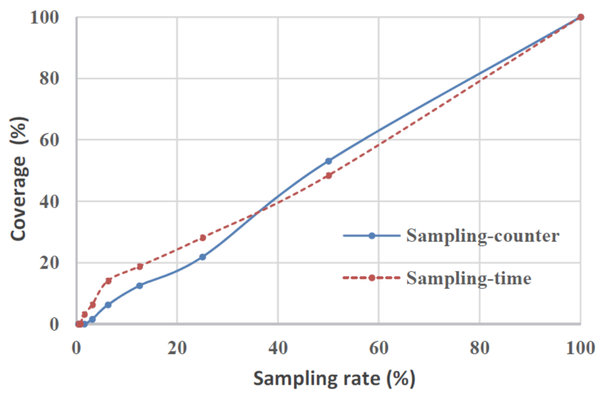

Figure 5: Logging effectiveness of two sampling-based approaches

The results of two sampling-based logging systems, Sampling-counter and Sampling-time, are illustrated in Figure 5. The effectiveness of either Sampling-counter or Sampling-time is approximately proportional to the sampling rate, which is much lower than what *Log*^{2} achieves. It is worth noting that the budget size of 120 logs/interval is equivalent to the sampling rate of 3%. While *Log*^{2} achieves almost 100% coverage with such budget size, Sampling-counter and Sampling-time achieve only 2% and 6% coverage, respectively.

For the issues injected by *exhausting CPU by other processes*, there are in total 690,000 individual calls on 6 instrumented loop blocks during the experiment (note that each loop block is one MCR). By using *Log*^{2}, only 22,000 (97% reduction) calls on loop blocks are recorded (budget size = 120 logs/interval), with the average execution time of 160ms. By inspecting the loop-related logs, we found that the average execution time is 423ms when ResourceEater is launched, which is significantly larger than the average value (160ms) without the impact of ResourceEater. Our inspection shows that the logs reflecting loops with long execution time are recorded, which demonstrates the capability of *Log*^{2} to detect the performance issue due to exhausted CPU usage.

In summary, the experimental results show that *Log*^{2} is effective in detecting performance issues, while keeping the volume of logs low.

Additional overhead. *Log*^{2} works in the same process of the BlogEngine service, hence its own CPU/Memory usage cannot be measured directly. In order to evaluate the overhead of *Log*^{2}, we measure the overall CPU/Memory usage of the BlogEngine system integrated with *Log*^{2}, and compare it with the overall usage of the BlogEngine system integrated with the baseline logging approach (outputting all logs). We run the experiment with each setting 7 times to overcome random variations.

Table 2: Comparison on overall resource usage

Logging system	Memory(GB)	CPU(%)
Log^2	4.74±0.21	63.4±3.0
Baseline	4.70±0.25	70.6±4.1

According to Table 2, the additional memory usage of *Log*^{2} over the baseline approach is not noticeable. When integrated with *Log*^{2}, the average CPU usage of BlogEngine is slightly lower than that with the baseline logging system. This is because using *Log*^{2}, a large number of logging requests are discarded at early stage, therefore a significant amount of processing (such as logging state extraction or string conversions) as well as lock contention are avoided, leading to reduced CPU usage.

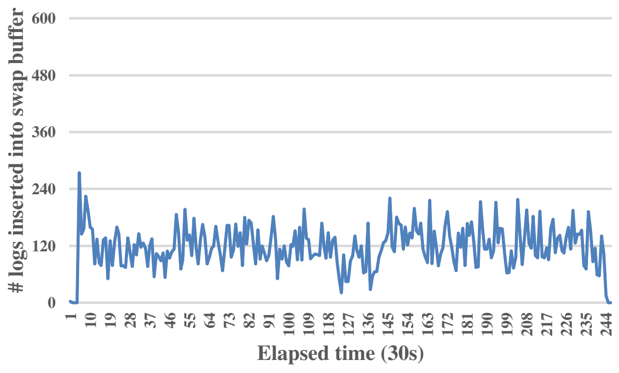

Figure 6: Dynamics of swap buffer size

In order to evaluate the memory usage of *Log*^{2}, we monitor the size of the swap buffer over time. Figure 6 shows the number of logs inserted into the swap buffer per flush interval. There is one peak at the beginning, when the threshold for the utility score is not converged. The peak is about 1.3 times higher than average, which is far from the default maximum memory limit set in *Log*^{2}. In addition, it takes only five iterations to converge, which shows that the small memory peak disappears quickly. The variation of the curve is mostly caused by the randomness in the workload.

6 An Application to Microsoft ServiceX

To further evaluate *Log*^{2}, we have applied it to analyze the performance logs of Microsoft ServiceX (the service name is anonymized due to confidentiality). ServiceX is a large-scale online service system, serving millions of users globally.

Designed with a 3-tier architecture, ServiceX is run on a large number of machines, each of which continuously generates huge amount of logs. A typical front-end machine usually generates logs with a speed of 30MB per minute. Log aggregation from all the machines is a heavy task, since each machine generates about 40GB logs every day. ServiceX provides a logging API called MoS for performance diagnosis. The corresponding logs are called MoS logs (i.e., performance logs), which take up 20%-40% of the total logs. Engineers of ServiceX would like to reduce the large volume of MoS logs, since most of them are not useful for performance diagnosis and they simply incur overhead.

We apply *Log*^{2} to evaluate its ability to reduce the volume of MoS logs.

Setup. Each MoS log entry contains the following information: log time, execution time of the MCR, code region ID, and thread ID. Such information is sufficient to re-construct the execution flows of all the MoS logs. We randomly select 12 different datasets. Each dataset contains logs generated during one continuous hour.

We focus on evaluating logging bandwidth and effectiveness in our study. To do so, we identify performance hotspots, which are the code regions that take most time to execute. We choose the MoS logs having the top 0.3% (i.e., 1 - 99.7%, which is a 3-sigma rule of thumb [22]) longest execution time as the performance hotspots. We then apply *Log*^{2} to see how many of these performance hotspots can be successfully identified. We choose t as the utility formula. We evaluate logging effectiveness as the coverage of the performance hotspots by varying the budget size. Additionally, we also evaluate how the flush interval affects the effectiveness.

Results. Figure 7 shows the logging effectiveness of *Log*^{2} by varying the budget size. Since we conduct experiments on 12 datasets, the effectiveness on each budget is represented by a range. As shown in Figure 7, the coverage of performance hotspots quickly comes up to 100% when the budget size increases. Particularly, when the budget is set to 100 logs/interval, which is equivalent to the sampling rate of 0.77%, the coverage is already 98%. On the other hand, only 4.5 MB logs are recorded, while the size of original MoS logs is 500MB for each dataset.

Figure 8 shows the effectiveness of *Log*^{2} under different flush interval values. Here the budget is set to 120 logs/interval, which is equivalent to the sampling rate of

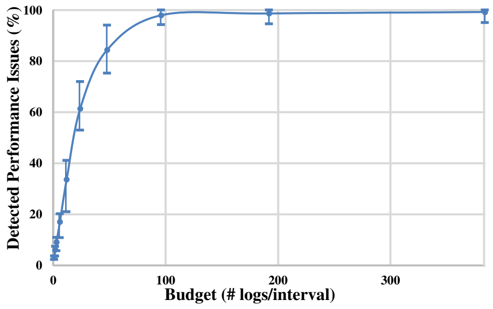

Figure 7: Logging effectiveness vs. budget

1.0% . When the flush interval is very small, the coverage rate is relatively low, mainly due to the significance of randomness on the workload. Setting the flush interval to 30 seconds is satisfactory, since the coverage rate here is almost 100%.

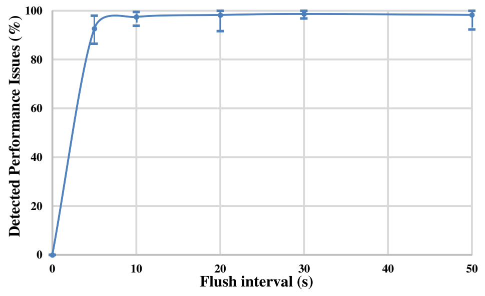

Figure 8: Logging effectiveness vs. flush interval

In summary, our case study on ServiceX has confirmed the applicability of *Log*^{2} to real-world systems.

7 Discussion

Budget control for multiple services. In our current design, *Log*^{2} is implemented as a runtime logging library and can be dynamically linked to a service system under monitoring. It controls the budget for only one single service. As budget can be changed dynamically, it is possible to make *Log*^{2} a standalone process, which manages a set of budgets for multiple services. Such a centralized budget control system can further enable dynamic budget re-allocation to different services.

Supporting more types of performance analysis. *Log*^{2} is very effective for capturing performance hotspots on-the-fly. In practice, there are other commonly required types of performance analysis. For example, to understand the overall latency status of the system under monitoring, the total number of times the latency hits the 3-sigma threshold, the average latency of a component, and so on. *Log*^{2} has the ability to provide such information. For example, *Log*^{2} maintains the *mean* and *standard deviation* values for each MCR. In addition, *Log*^{2} can record how many times each MCR is updated. Hence, many other measures of performance status (such as the 3-sigma measures) can be easily derived from these basic statistics. Additionally, all the data of *Log*^{2} can be dumped periodically and used by other performance analysis tools. Analytical reports based on the off-line processing of the data can produce comprehensive information for postmortem analysis.

Multiple objectives. Currently we only define budget in terms of I/O bandwidth, as it is mostly concerned by our surveyed participants. It is possible to consider more objectives, such as CPU and memory usage, and to control logging overhead by performing multi-objective optimization. We will address it in our future work.

Where to log. As described in Section 4.1, we identify MCRs for performance diagnosis. In this paper, we focus on the problem of "whether to log". Another important topic is "where to log", i.e., the automatic identification of code regions that should be logged and monitored. The two problems, "whether to log" and "where to log" are closely related to each other. For example, the logging mechanism we proposed enables "conservative logging", i.e., developers can instrument a large amount of logging statements without concerning about the logging cost. This is an important topic of our future work.

Leveraging non-performance logs. Although performance logs are common in practice, there are also other types of logs such as those for failure diagnostics. Two adjacent log entries indicate the time spent on executing code between the two log entries. It would be interesting to leverage those logs for performance diagnosis.

Extension to failure diagnosis. Our current work focuses on analyzing performance logs for effective and efficient monitoring and diagnosis of performance issues. Apart from performance logs, there are other types of logs such as logs recording error and failure information. These logs are mainly for diagnosing software failures in production environment [24, 26, 27]. How to extend our work to support failure diagnosis is important future work.

8 Related Work

Performance monitoring and diagnosis has becoming increasingly important, especially in the era of Internet-based services and cloud computing. A large amount of research has been conducted to characterize [13, 28, 16] and improve system performance [14, 21, 23, 12, 7].

In production environment, logging is still the most commonly used technique for performance monitoring and diagnosis. Dapper [20] is a large-scale distributed tracing infrastructure widely adopted by Google for ubiquitous and continuous monitoring. Dapper is designed to have low overhead, application-level transparency and scalability. *Log*^{2} shares the same design goals with Dapper, and goes one-step forward with finer-grained and more accurate control on logging overhead to comply with the resource budget. Dapper flushes only a fraction of all traces using a sampling (with a manually configured sampling rate) approach such that interesting traces could be missed. *Log*^{2} preserves useful logs with significantly higher effectiveness. At the same time, *Log*^{2} guarantees the resource budget constraints, which can be violated in Dapper.

ETW (Event Tracing for Windows) [2] is a framework that can log Windows kernel or application-specific events to a log file. It has a buffering mechanism that reduces the number of disk accesses for logging. However, ETW is not cost-aware: it cannot selectively record a number of logs based on a given budget.

Paradyn [17] also controls its instrumentation overhead dynamically. However, it depends on users to explicitly configure *where to log*, and predict *whether to log*. *Log*^{2} instead is user-transparent in that *whether to log* decisions are dynamically made by the logging mechanism. Excessive instrumentation is commonly adopted in the profiling domain. Matthew [6] presents sampling based low-cost instrumentation to enable feedback-guided just-in-time optimization. Like Dapper, logging based on random sampling would miss interesting traces.

Yuan et al. [25, 26, 27] have pioneered the work on log-based failure diagnosis. LogEnhancer [27] aims to enhance the recorded contents in existing logging statements by automatically identifying and inserting critical variable values into them. ErrLog [26] utilizes a number of exception patterns that potentially cause system failures, and then adds proactive logging code to automatically log all of them. These work mainly address the problems of "what to log" and "where to log". Our work, instead, focuses on "whether to log".

9 Conclusion

In this paper, we have presented *Log*^{2}, a cost-aware logging system for making the optimal "whether to log" decisions. *Log*^{2} adopts a two-phase filtering mechanism to selectively record useful logs based on a given logging bandwidth. The experimental results on both BlogEngine and ServiceX demonstrate the capability of *Log*^{2} to control logging overhead while preserving effectiveness.

Currently, *Log*^{2} analyzes performance logs for performance monitoring and diagnosis. As we discussed in Section 7, in the future we will extend *Log*^{2} to support more type of analysis, such as supporting other kinds of logs for failure diagnosis.

References

[1] Blogengine, 2007. http://www.dotnetblogengine.net/.

[2] Etw tracing, 2007. https://msdn.microsoft.com/en-us/library/ms751538(v=vs.110).aspx.

[3] Record and run a web performance test, 2013. http://msdn.microsoft.com/en-us/library/ms182539.aspx.

[4] Log2, an overhead-constrained logging system, 2014. http://research.microsoft.com/en-us/projects/log2/default.aspx.

[5] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[6] ARNOLD, M., AND RYDER, B. G. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation* (2001), ACM Press, pp. 168–179.

[7] ARULRAJ, J., CHANG, P., JIN, G., AND LU, S. Production-run software failure diagnosis via hardware performance counters. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, USA* (2013), pp. 101–112.

[8] CHONG, K. H., Y., G. C., AND Y, L. Pid control system analysis, design, and technology. In *IEEE Trans Control Systems Tech* (2005).

[9] DING, R., FU, Q., LOU, J., LIN, Q., ZHANG, D., AND XIE, T. Mining historical issue repositories to heal large-scale online service systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA* (2014), pp. 311–322.

[10] ERLING, A. Sufficiency and exponential families for discrete sample spaces. In *Journal of the American Statistical Association* (1970).

[11] GOODELL, B. R. *Smoothing Forecasting and Prediction of Discrete Time Series*. Englewood Cliffs, NJ: Prentice-Hall, 1963.

[12] HAN, S., DANG, Y., GE, S., ZHANG, D., AND XIE, T. Performance debugging in the large via mining millions of stack traces. In *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland* (2012), pp. 145–155.

[13] JIN, G., SONG, L., SHI, X., SCHERPELZ, J., AND LU, S. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12* (2012), pp. 77–88.

[14] JOVIC, M., ADAMOLI, A., AND HAUSWIRTH, M. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 155–170.

[15] L, W. *All of statistics: A concise course in statistical inference*. New York: Springer, 2004.

[16] LIU, Y., XU, C., AND CHEUNG, S. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India* (2014), pp. 1013–1024.

[17] MILLER, B., CALLAGHAN, M., CARGILLE, J., HOLLINGSWORTH, J., IRVIN, R., KARAVANIC, K., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. In *IEEE Computer* (1995).

[18] MYRON, A., AND ELI, I. *Numerical analysis for applied science*. John Wiley, Sons, 1998.

[19] OGATA, K. *Discrete-time control systems*. Prentice-Hall, 1987.

[20] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure. In *Google technical report* (2010).

[21] SONG, L., AND LU, S. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA,* (2014), pp. 561–578.

[22] WHEELER, D. J., AND CHAMBERS, D. S. *Understanding Statistical Process Control*. SPC Press, 1992.

[23] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 117–132.

[24] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems* (March 2010).

[25] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[26] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., ZHOU, Y., AND SAVAGE, S. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association.

[27] YUAN, D., ZHENG, J., PARK, S., ZHOU, Y., AND SAVAGE, S. Improving software diagnosability via log enhancement. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, March 2011).

[28] ZAMAN, S., ADAMS, B., AND HASSAN, A. E. A qualitative study on performance bugs. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, Zurich, Switzerland* (2012), pp. 199–208.