Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold

Xu Zhao
University of Toronto

Kirk Rodrigues
University of Toronto

Yu Luo
University of Toronto

Michael Stumm
University of Toronto

Ding Yuan
University of Toronto

Yuanyuan Zhou
University of California, San Diego

ABSTRACT

When systems fail in production environments, log data is often the only information available to programmers for postmortem debugging. Consequently, programmers' decision on *where* to place a log printing statement is of crucial importance, as it directly affects how effective and efficient postmortem debugging can be. This paper presents Log20, a tool that determines a near optimal placement of log printing statements under the constraint of adding less than a specified amount of performance overhead. Log20 does this in an automated way without any human involvement. Guided by information theory, the core of our algorithm measures how effective each log printing statement is in disambiguating code paths. To do so, it uses the frequencies of different execution paths that are collected from a production environment by a low-overhead tracing library. We evaluated Log20 on HDFS, HBase, Cassandra, and ZooKeeper, and observed that Log20 is substantially more efficient in code path disambiguation compared to the developers' manually placed log printing statements. Log20 can also output a curve showing the trade-off between the informativeness of the logs and the performance slowdown, so that a developer can choose the right balance.

CCS CONCEPTS

• **Computer systems organization → Reliability**; • **Software and its engineering → System administration**;

KEYWORDS

Log placement, distributed systems, information theory

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132778

ACM Reference Format:
Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold . In *Proceedings of SOSP '17*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3132747.3132778

1 INTRODUCTION

When systems fail in a production environment, log data is often the only information available for postmortem debugging. Consequently, developers' decision on *where* and what to log is of crucial importance, as the informativeness of logs directly affects the time it takes for postmortem debugging.

Prior work mostly focused on automating *what* to log so as to improve the quality of existing log printing statements (LPS) either by including additional variable values [37] or by adjusting verbosity [36]. However, *where* to place an LPS – a more fundamental and challenging problem – has not been adequately addressed. The only prior work on placing LPSes was Errlog [35] that automatically places *error* LPSes at locations where error conditions may occur (e.g., non-zero system call return values). While error LPS placement is a crucial first step towards automated placement of LPSes, many failures, especially complicated ones with long fault-propagation paths, require additional log outputs that capture non-erroneous but still important execution paths. As evidence, many LPSes of existing systems record non-erroneous execution states, typically at Info verbosity, that are enabled by default as shown in Table 1.

	Enabled by default				Non-default	
System	Fatal	Error	Warn	Info	Debug	Trace
Cassandra	0	184	254	325	240	431
Hadoop	89	1131	1727	2862	2477	288
HBase*	52	817	1137	1043	1052	351
ZooKeeper	0	205	313	239	201	42
Total	141	2337	3431	4469	3970	1112

Table 1: Number of log printing statements under different verbosity levels in the source code, excluding testing code. *HBase used Debug as default log verbosity from 2008-2015.

The placement of LPSes to record non-erroneous events is a much more difficult problem than placing LPSes to record errors. First, it is difficult to know which program locations are more "log-worthy" than others, because it is often hard to predict the usefulness of an LPS prior to the software's release and the occurrence of unexpected failures. Thus, deciding where to insert an Info LPS is a black art. Second, because Info log entries are output during normal execution, developers worry about LPS performance overhead. In contrast, placing Error LPSes is more straightforward. They can be placed where error conditions may occur. Languages with built-in exception support, such as Java or Scala, make the identification of such locations even easier. Overhead is less of a concern as these error conditions do not typically occur during normal execution.

Alternative approaches proposed by other researchers can deterministically replay a prior execution path by recording all events that might result in non-deterministic control-flow changes [1, 5–7, 12, 20, 23, 25, 29, 31, 33]. They typically incur a high performance overhead or can be impractical due to privacy concerns over the recorded information. Ball and Larus [2] proposed an efficient path profiling algorithm. Unlike LPSes, it does not allow developers to balance the trade-off between informativeness and slowdown. Instead, it disambiguates every path while introducing an average slowdown of 31%.

Overall, the state of the art of LPS placement is poor. Prior work has shown that the change rate of log printing code is 1.8 times higher than the rest of the code [36]. We studied the revision history of Hadoop, HBase, and ZooKeeper, and discovered three fundamental problems with current LPS placement practices:

- *Reactive instead of proactive:* the sole purpose of 21,642 revisions was to add LPSes, indicating that they are added only as after-thoughts — we assume after a failure occurred.
- *Difficult to predict the informativeness and overhead:* 2,105 revisions only modify LPS verbosity levels. Analyzing them, we found developers struggled to predict the informativeness of an LPS and the overhead it causes. Moreover, we found that the static nature of LPS placement is not ideal – e.g., a statement that is rarely executed in one workload may be on the hot path of another workload.
- *Scalar design of verbosity is difficult to use:* developers often have difficulties in setting the right verbosity level of each LPS. Whether an LPS should be categorized as Error or Info can be highly subjective, and again, may depend on the particular workload.

This paper proposes Log20, a tool that automates LPS placement without requiring any domain knowledge.[1] Log20 is also able to automatically collect variable values for each LPS. It thus automates all aspects of software logging except for generating the static text included in developer-written logs (it outputs an ID in each LPS together with the variable values).

Users of Log20 simply specify a performance overhead threshold (e.g., 2% of slowdown, or no more than an average of three log messages per request), and Log20 is capable of computing a near optimal placement of LPSes whose overhead is within the specified threshold. We use the term *LPS placement* to refer to both the placement and variable value logging, unless otherwise specified. Log20 is also capable of outputting a curve showing the trade-off between the informativeness of LPS placements and their performance overhead, so that a developer or system administrator can choose the right balance.

We measure the informativeness of a particular LPS placement by measuring its ability to differentiate between different execution paths taken by a program. For a particular placement, two different paths may output the same sequence of log entries, making it impossible to determine which path was executed using logs alone. Thus, we formally reason about the informativeness of a placement first by considering how fine grained it is in disambiguating paths.

To further compare the informativeness of two LPS placements when they can disambiguate different sets of paths, we use Shannon's information theory to measure the entropy [27] of a program, specifically by considering all possible execution paths and the probability of each execution path occurring at runtime. Intuitively, a program with a larger number of paths has a higher entropy, and if its paths are more unpredictable, it also has a higher entropy. Because an LPS placement can disambiguate these execution paths, it reduces entropy. Therefore, given a placement, the remaining entropy of a program indicates the informativeness of the placement. A placement that reduces the entropy to zero is one that can disambiguate every possible execution path.

We designed an algorithm that computes a near optimal LPS placement under a given overhead threshold. An optimal placement is one that results in the smallest entropy among all the placements that respect the threshold. However, computing the optimal placement has $O(2^n)$ complexity. of where n is the total number of basic blocks. Therefore, we designed an efficient dynamic programming algorithm that approximates the optimal placement.

To collect a system's runtime execution paths and their frequencies, we also designed and implemented a JVM-based tracing library that is suitable for continuous profiling of production systems. The instrumentation incurs very low

[1] The name of the tool, Log20, comes from the game of twenty questions, where a player's goal is to identify an object in twenty yes/no questions or less. Similar to the game, placing an LPS is like asking a yes/no question whose answer depends on whether the LPS gets executed or not.

overhead (27ns per trace point). The tracing system can also be used as an alternative to existing logging libraries like Log4j 2 [21] to collect log data.

One fundamental difference between Log20 and other existing logging approaches is that Log20's placement is not static, but instead reacts to the workload. Using our tracing library, Log20 is able to incrementally update the LPS placement, and periodically patch the system's bytecode at runtime to enable new LPSes or disable existing ones. Therefore the same system when deployed in two different environments could have very different placements.

We applied Log20 on four widely used distributed systems: HBase, HDFS, YARN, and ZooKeeper. Compared to the LPSes placed by the corresponding developers, Log20 is substantially more effective. For example, Log20's placement outputs only 5% of the number of the log entries, yet is as informative as developers' manual LPS placement in HDFS. We also demonstrate that the LPSes placed by Log20 can help in the diagnosis of 68% of 41 randomly selected failures.

This paper makes the following three contributions:

- An algorithm for placing LPSes that is near optimal.
- A metric on the informativeness of LPS placement.
- A low-overhead run-time tracing system which can be used for both continuous profiling and the logging library.

Log20 has several limitations. First, one of the key differences between Log20 and other logging approaches is that it does not assign a scalar verbosity level to each LPS. Instead, the frequency of a log message can be used to infer its criticality: an event that occurs only once is likely more critical than one that occurs many times. This could be inconvenient in that it makes postmortem debugging more difficult. Furthermore, if users would like to enable more verbose logs, they can no longer do so by adjusting the logging verbosity; instead, they will need to set a higher overhead threshold. Second, Log20's placement algorithm treats each path as an unordered rather than ordered collection of basic blocks in order to avoid costly sequence comparisons. In practice, order among log entries is often less reliable in the face of concurrency and is more time-consuming for manual examination. Consequently, developers typically rely on the appearance of particular log entries to determine whether a path was executed, instead of relying on log order. Other limitations are discussed in §7.

The rest of the paper is organized as follows. §2 describes our study on the revision history of LPSes. §3 discusses the informativeness measurement of an LPS. §4 presents our dynamic programming algorithm to compute a near-optimal LPS placement under a slowdown threshold. §5 describes our implementation while §6 describes our tracing library. §8 discusses the experimental evaluation of Log20. Finally, we survey related work in §9 before we conclude.

	LPSes		LPS modifying revisions			
	total	mod.	total	add	rm	verb.
Hadoop	9125	42%	12158	9282	1545	1331
HBase	4644	42%	14039	10654	2702	683
ZooKeeper	1094	41%	2624	1706	827	91
Total	14863	42%	28821	21642	5074	2105

Table 2: The number and breakdown of LPS revisions.

2 REVISION HISTORY OF LPSES

To understand the challenges of manual LPS placement, we analyzed the complete revision history of LPSes in three systems: Hadoop, HBase, and ZooKeeper. Hadoop includes HDFS, YARN, and MapReduce. Each addition, removal, or verbosity change of an LPS suggests a corrective action that was taken to improve this LPS' informativeness or decrease its incurred overhead.

Yuan *et al.* conducted a similar study [36] in 2012, but they analyzed systems written in C and did not focus on LPS placement (e.g., they did not study the revisions that only added LPSes). We observe newer distributed systems are increasingly built on JVM languages which are generally considered to have better logging quality. For example, studies [34, 35] have shown that 57% of failures in C/C++-based servers do not even output error messages, whereas only 24% of failures in HDFS, Hadoop, Cassandra, and HBase do not output error messages. Thus, we repeat the study with the new focus and new systems while using much of the same methodology from Yuan *et al.*'s work.

The study is designed to find and categorize changes to LPSes that have to do with the LPSes themselves, rather than being associated with other functional changes. We first analyze each patch made to the master branch, collecting those that contain LPSes. To filter functionality-associated logging changes, we discard any LPS modification where the branch condition that dominates the LPS was changed [36]. To do so, we obtain the abstract syntax tree (AST) of the old and new versions using JavaParser [15]. For each tree, we then extract the paths leading from the AST root (i.e., the entry of the function) to the basic block that contains the modified LPS. This path contains all of the branch conditions dominating the LPS. If the path is the same in both trees, we assume this patch's only purpose was to modify the LPS. We use the term *revision* to refer to these singularly LPS-modifying patches.

For each revision, we further categorize the change as an addition, removal, verbosity change, variable value change, or static text modification. An add-remove pair (in the revision's diff) is considered a static text modification if the Levenshtein [17] distance ratio between the two is less than 0.5. Otherwise it is categorized as an addition and a removal.

Table 2 summarizes the results of our analysis. We find that there are 28,821 revisions to LPSes alone, meaning on average, each LPS is modified 1.93 times. Note that the total number of LPS modifying revisions includes those that change variable values or static text, but their specific numbers are not shown as they are irrelevant to LPS placement. The third column of the table further shows that 42% of the LPSes were modified at least once since they were first introduced.

We observe a total of 21,642 revisions that add LPSes. This suggests that developers often add LPSes as after-thoughts, i.e., after a failure has occurred, or they are adding debugging support after developing functionality code. For example, the sole purpose of YARN-2627 is to add Info LPSes to aid the debugging of a feature (YARN-611) that was added weeks prior. As another example, the only purpose of HBase-6004 is to add 14 Info LPSes to "help debugging MR jobs".

We observe a large number (2,105) of revisions only modify the verbosity level. In many of them, developers are reconsidering the trade-off between the LPS' overhead and informativeness. For example, HDFS-6836 reduces the verbosity of datanode Info logs to Debug because they are "present within the inner loops". However, HDFS-8315 strives to revert this change since the reporter does not observe the same bottleneck under HBase workloads, and they would like to have the information back because *"... There is a good reason why it was at INFO for so many years. This is very useful in debugging load issues."* Such debate is common, and it highlights the difficulty in balancing the trade-off between informativeness and performance overhead under different workloads.

The rest of the verbosity changes are made as developers reconsider the criticality of an event. For example, the reporter of HDFS-4048 wanted to increase the verbosity of a failed directory access log from Info to Error since he could not detect a permission error by only grep'ping for Error messages. However, his code reviewer suggests that the message only be increased to Warn, as the logged situation may arise due to *expected* disk failures. In another example, HDFS-1054, we also observe the verbosity being increased from Debug to Info since the developer feels the information is worth having under normal operating conditions. However, the change does not stop there – 6 years later, HDFS-10381 again increases the verbosity to Warn for the same LPSes. Both examples highlight the difficulty in setting verbosity levels, especially as different scenarios suggest different semantics.

3 INFORMATIVENESS OF LOGGING

In this section, we discuss how we measure an LPS placement's power in disambiguating execution paths. We first discuss how log output is used to disambiguate paths, and then how different placements can be compared. This eventually leads us to the design of a single metric, entropy, that measures the informativeness of an LPS placement.

```
1  int max3 (int a[]) {
2    if (a.length != 3)
3      return ERROR;
4    int max = 0, i = 0;
5    while (i < 3) {
6      if (a[i] > max)
7        max = a[i];
8      i++;
9    }
10   return max;
11 }
```

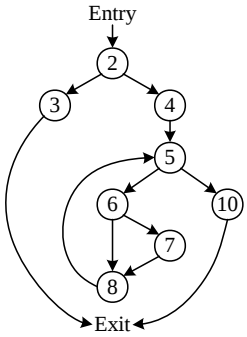

Figure 1: An example program, and its control flow graph, that selects the maximum value of a three-element array.

3.1 Disambiguating Paths with Log Output

The informativeness of an LPS placement directly affects a developer's ability to disambiguate execution paths using log output from the placement. The most informative LPS placement is one where each unique path outputs a unique log sequence. Accordingly, a less informative placement is one where multiple unique paths output the same unique log sequence. For example, consider the program in Figure 1. Table 3 shows every possible path that the program could take. An obvious placement that is the most informative is one where an LPS is placed in every basic block. However, if we place an LPS, "l_7", in block 7, then P_2 will output the log sequence $[l_7, l_7, l_7]$; P_3, P_5, and P_6 will output $[l_7, l_7]$; P_4, P_7, and P_8 will output $[l_7]$; and finally, P_1 and P_9 will output []. Therefore, each unique log sequence could be printed by multiple unique paths. This means that if, for example, developers see the log sequence $[l_7]$, they will know that one of P_4, P_7, and P_8 was taken, but not which one exactly.

Before discussing how to compare the informativeness of placements, we first define the previous concepts.

- A program consists of a set of basic blocks, $BB = \{bb_1, bb_2, .., bb_n\}$.
- $EP = \{P_1, P_2, .., P_m\}$ is the set of all possible execution paths of a program, where each path P_i is a sequence of basic blocks.
- An LPS placement, S, is a subset of BB where a unique LPS is placed in each block. For simplicity, we postpone the consideration of logging variables until §4.3.
- Under placement S, L_i is the log sequence output by execution path, P_i (e.g., in the previous example $L_2 = [l_7, l_7, l_7]$).
- $O(S) = \{L_1, .., L_k\}$ is the set of all possible log sequences under placement S. In the previous example, $O(S) = \{[], [l_7], [l_7, l_7], [l_7, l_7, l_7]\}$.

ID	Basic block sequence	Input
P_1	<2, 3>	[]
P_2	<2, 4, 5, 6, 7, 8, 5, 6, 7, 8, 5, 6, 7, 8, 5, 10>	[1,2,3]
P_3	<2, 4, 5, 6, 7, 8, 5, 6, 7, 8, 5, 6, 8, 5, 10>	[1,3,2]
P_4	<2, 4, 5, 6, 7, 8, 5, 6, 8, 5, 6, 8, 5, 10>	[3,2,1]
P_5	<2, 4, 5, 6, 7, 8, 5, 6, 8, 5, 6, 7, 8, 5, 10>	[2,1,3]
P_6	<2, 4, 5, 6, 8, 5, 6, 7, 8, 5, 6, 7, 8, 5, 10>	[0,1,2]
P_7	<2, 4, 5, 6, 8, 5, 6, 7, 8, 5, 6, 8, 5, 10>	[0,2,1]
P_8	<2, 4, 5, 6, 8, 5, 6, 8, 5, 6, 7, 8, 5, 10>	[0,0,1]
P_9	<2, 4, 5, 6, 8, 5, 6, 8, 5, 6, 8, 5, 10>	[0,0,0]

Table 3: Possible execution paths of the code snippet in Figure 1. Each row shows a path with its ID, the sequence of basic blocks it traversed, and an example input that results in this path. A blank space indicates that a block is not traversed in this path but is in others.

- The *Possible Paths* set of a log sequence L_i, $PP(L_i) = \{P_i, .., P_j\}$, is the set of paths that would output L_i when executed (e.g., $PP([l_7, l_7]) = \{P_3, P_5, P_6\}$).
- *Disambiguated Paths*, $DP(S) = \{PP(L) \mid L \in O(S)\}$, is the set of all possible PP sets. In the previous example, $DP(S) = \{\{P_2\}, \{P_3, P_5, P_6\}, \{P_4, P_7, P_8\}, \{P_1, P_9\}\}$.

It should be clear that $DP(S)$ is a partition of EP into disjoint subsets of possible path sets (PP), i.e., for any $PP_1, PP_2 \in DP(S)$, $PP_1 \cap PP_2 = \emptyset$, and $\cup_{PP \in DP(S)} PP = EP$.

3.2 Disambiguated Paths of LPS Placement

Now we can consider the path disambiguating capability of an LPS placement. We say placement S_1 *subsumes* placement S_2, represented as:

$S_1 \succeq S_2$, if and only if $DP(S_1)$ is a refinement of $DP(S_2)$.

That is, every $PP \in DP(S_1)$ is a subset of some $PP' \in DP(S_2)$. Intuitively, this means that S_1 is more disambiguating than S_2, since for any log output $L \in O(S_1)$ and $L' \in O(S_2)$ where L and L' are the outputs of the same execution path under the two placements, $PP(L)$ is a subset of $PP(L')$; this in turn means that developers have less possible paths to disambiguate. If $S_1 \succeq S_2$ and $DP(S_1) \neq DP(S_2)$, we say $S_1 \succ S_2$.

Consider two placements: $S = \{6, 7\}$ and $S' = \{3, 7\}$. We have $S \succ S'$, because $DP(S) = \{\{P_1\}, \{P_2\}, \{P_3\}, \{P_4\}, \{P_5\}, \{P_6\}, \{P_7\}, \{P_8\}, \{P_9\}\}$, and $DP(S') = \{\{P1\}, \{P2\}, \{P3, P5, P6\}, \{P4, P7, P8\}, \{P9\}\}$. In fact, $\{6, 7\}$ is one of the most disambiguating placement because different paths will output different log output.

Note that $S \cup S' \succeq S$, i.e., adding more LPSes to a placement, S, will result in a more disambiguating placement. However, the reverse it not true, i.e., a placement, S, that is more disambiguating than S' does not necessarily imply that S is a superset of S'. More formally, $S \succeq S' \not\Longrightarrow S \supseteq S'$. For example, consider $S = \{6, 7\}$ and $S' = \{3, 7\}$, we have $S \succ S'$, but $S \not\supseteq S'$.

3.3 Entropy of LPS Placement

The $\succ$ relation does not allow us to compare the informativeness of two placements, S_1, S_2 when one does not subsume the other. We use Shannon's *entropy* from information theory to further compare different placements. Shannon's entropy measures the uncertainty, or unpredictability, of a system. It is defined as:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x) \tag{1}$$

where, in the context of a program, we use $p(x)$ to represent the probability of observing the execution path, x. We use X to represent all of the possible paths of a program. For any real system, there may be an infinite number of paths, but since Log20 relies on sampling the production system to collect path profiles, they are a finite number for our purposes. Accordingly, we calculate $p(x)$ as the number of occurrences of path x divided by the total number of paths sampled in the production system.

Intuitively, a program with an entropy value H means programmers have 2^H possible paths to disambiguate during postmortem analysis. A program with a lower number of possible paths has a lower entropy, and thus a lower degree of uncertainty. Similarly, for programs with the same number of paths, the more predictable which path will be taken, the lower the entropy. For example, if two programs both have 2 paths, but in the first program each path has a 50% of probability of being taken whereas in the second program one particular path has a 99% chance of being taken, then the second program is more predictable as it has a lower entropy value. A program with entropy 0 means that there is no uncertainty, i.e., there is only one path.

We can now use entropy to measure the informativeness of an LPS placement. Given a placement, S, we measure its entropy in two steps. First, we consider a particular log output L that is produced by S, and measure H_L:

$$H_L(X) = -\sum_{x \in PP(L)} \frac{p(x)}{p(L)} log_2 \frac{p(x)}{p(L)} \tag{2}$$

where $p(L)$ is the probability of the program taking a path that outputs L. $\frac{p(x)}{p(L)}$ is the probability of the software taking path x among all possible paths in $PP(L)$. For example, consider the LPS placement $S = \{3, 7\}$ and log output $[l_3]$. $H_{[l_3]} = 0$ because only one path, P_1, produces this output.

Next, we can measure the entropy of placement S, H_S, by considering all possible log outputs produced by S:

$$\begin{aligned} H_S(X) &= \sum_{L \in O(S)} p(L) H_L \\ &= -\sum_{x \in X} p(x) log_2 \frac{p(x)}{p(L_x)} \end{aligned} \tag{3}$$

bb	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	w
2	1	1	1	1	1	1	1	1	1	1.00
3	1	0	0	0	0	0	0	0	0	0.11
4	0	1	1	1	1	1	1	1	1	0.89
5	0	4	4	4	4	4	4	4	4	3.56
6	0	3	3	3	3	3	3	3	3	2.67
7	0	3	2	1	2	2	1	1	0	1.33
8	0	3	3	3	3	3	3	3	3	2.67
10	0	1	1	1	1	1	1	1	1	0.89

Table 4: Basic block count matrix of the example program. Each row represents a basic block. Weight w is the number of times the basic block is expected to be traversed by a path if we assume each path has equal probability.

.

where L_x is the log output produced by path x under S.

Consider the two placements $S_1 = \{3\}$ and $S_2 = \{7\}$ from the above example. $DP(\{3\}) = \{\{P_1\}, \{P_2 - P_9\}\}$ and $DP(\{7\}) = \{\{P_1, P_9\}, \{P_2\}, \{P_3, P_5, P_6\}, \{P_4, P_7, P_8\}\}$. Therefore $S_1 \not\geq S_2$ and $S_2 \not\geq S_1$. To measure their entropy, we need to know the probability of each path. If we assume each path shown in Table 3 has equal probability, then we will have $H(\{3\}) = 2.67$ whereas $H(\{7\}) = 1.28$, indicating $\{7\}$ is a more informative placement compared to $\{3\}$.

We have the following property:

Theorem 1. *If $S \geq S'$, then $H_S \leq H_{S'}$ regardless of the paths' probability distribution.*

Proof. For each execution path $x \in X$, its contributions to H_S and $H_{S'}$ are $-p(x)log_2\frac{p(x)}{p(L_x)} = -p(x)log_2p(x)+p(x)log_2p(L_x)$ and $-p(x)log_2\frac{p(x)}{p(L'_x)} = -p(x)log_2p(x) + p(x)log_2p(L'_x)$, respectively, where L_x and L'_x are the log outputs of x under placement S and S', respectively. Since $S \geq S'$, $PP(L_x) \subseteq PP(L'_x)$, so $p(L_x) \leq p(L'_x)$, and thus $log_2p(L_x) \leq log_2p(L'_x)$. Hence we have $-p(x)log_2\frac{p(x)}{p(L_x)} \leq -p(x)log_2\frac{p(x)}{p(L'_x)}$, and $H_S \leq H_{S'}$. □

4 THE LPS PLACEMENT ALGORITHM

In this section, we discuss the design and implementation of a placement algorithm. The inputs are: (1) runtime execution paths and their frequencies, (2) the system's Java bytecode (used to determine the effect of logging variable values in each basic block and to identify the start and end of a request), and (3) a slowdown threshold. The algorithm will compute a near optimal placement – i.e., where to place a log printing statement and what variable values to include – that incurs less overhead than the specified threshold.

4.1 Estimating Slowdown

One approach to consider the slowdown of a placement is to run the system and measure the actual slowdown. While this is accurate, we cannot afford to benchmark the system with each LPS placement being considered by our algorithm. Instead, we opt to estimate the slowdown by considering the number of times the placed LPSes will get executed in a path. If this number is w, then we assume the slowdown will be $w \times t$, where t is the latency of executing a single LPS. The performance threshold is thus being considered as a threshold on the number of expected log output entries per run, e.g., no more than 3 log entries per request.

We compute the number of expected log output entries by considering the frequency of each path. For each basic block, bb, that appears in the path profile, we compute its weight, w, which is the expected number of times it will be traversed in a path as follows:

$$w = \sum_{x \in X} p(x)count(x, bb)$$

Recall that x is an execution path, and $p(x)$ is the probability of observing path x. $count(x, bb)$ is the number of times x traverses basic block bb. Table 4 shows the number of times each basic block appears in each path for the example in Figure 1. At line 3, the weight of the basic block is 0.11, indicating that if we place an LPS there, it will be printed 0.11 times on average in an execution path, or 11 times per 100 execution paths. For an LPS placement, S, the number of expected log output entries per execution path under this placement is simply the sum of the weights of all basic blocks in S. For example, if $S = \{6, 7\}$ in our example, there will be an average of 4 log entries printed by an execution path.

4.2 The Placement Problem and Algorithm

Now we can define the problem of LPS placement. Given a set of basic blocks, BB, where each block has a weight, w, the problem of placement is to find a subset of BB, $S \subseteq BB$, such that the sum of the weights of all basic blocks in S is under a threshold, W_T, and entropy H_S is minimized. We call this minimum entropy the *optimal placement*. Note that this does not yet take into account the use of logging variable values. We consider this in §4.3.

Computing the optimal placement using a brute force search requires enumerating every combination of basic blocks, and then selecting the one that offers the lowest entropy with a total weight that is under the threshold. The complexity of this brute force algorithm is 2^N, where N is the number of basic blocks. In practice, this algorithm is computationally infeasible; e.g., 1,402 unique basic blocks are traversed in a single HDFS write request on average.

Log20 solves this combinatorial optimization problem using a dynamic programming algorithm that approximates the optimal solution. Given all basic blocks, $[bb_1, bb_2, .., bb_N]$, the algorithm first sorts them by their weights such that $w_i \leq w_{i+1}$, where w_i is the weight of bb_i. Let $S(i-1, w)$ be a near optimal placement of LPSes in $[bb_1, bb_2, .., bb_{i-1}]$ within a total weight of w. For each basic block in $[bb_i, bb_{i+1}, .., bb_N]$,

row	bb	w=0.00		w=0.11 - 1.11		w=1.33		w=1.44 - 3.78		w=4.00	
		S	H_S	S	H_S	S	H_S	S	H_S	S	H_S
1	3	{}	3.17	{3}	2.67	{3}	2.67	{3}	2.67	{3}	2.67
2	10	{}	3.17	{3}	2.67	{3}	2.67	{3}	2.67	{3}	2.67
3	4	{}	3.17	{3}	2.67	{3}	2.67	{3}	2.67	{3}	2.67
4	2	{}	3.17	{3}	2.67	{3}	2.67	{3}	2.67	{3}	2.67
5	7	{}	3.17	{3}	2.67	{7}	1.28	{3,7}	1.06	{3,7}	1.06
6	6	{}	3.17	{3}	2.67	{7}	1.28	{3,7}	1.06	{6,7}	0.00
7	8	{}	3.17	{3}	2.67	{7}	1.28	{3,7}	1.06	{6,7}	0.00
8	5	{}	3.17	{3}	2.67	{7}	1.28	{3,7}	1.06	{6,7}	0.00

Table 5: The entropy-weight matrix for our example. Each entry shows the placement $S(i, w)$ and its entropy H_S. We merge the columns representing w_1 and w_2 when $S(i, w_1) = S(i, w_2)$ for any i.

S	P_1	P_2	P_3	P_4	P_5	P_6	W	H
1	0	0	1	2	2	3	1.33	0.67
2	0	0	2	2	3	3	1.67	1.00
3	3	2	2	1	1	3	2.00	1.00
1,2	0,0	0,0	1,2	2,2	2,3	3,3	3.00	0.33
1,3	0,3	0,2	1,2	2,1	2,1	3,3	3.33	0.33
2,3	0,3	0,2	2,2	2,1	3,1	3,3	3.67	0.00

Table 6: An example that shows our dynamic programming algorithm is not optimal. The program has three basic blocks (1, 2, and 3), and a total of 6 execution paths P_1-P_6. Each row shows a placement. Columns P_1-P_6 show the log output, represented as the number of appearances of the LPS that is placed at each basic block, produced by each path. We assume that the order of log entries cannot further distinguish different paths; i.e., if two paths produce the same type and number of log entries, then they also have the same sequence. W shows the total weight of each placement assuming each path has the same probability. The last column shows the entropy of each placement.

the algorithm adds it to the current placement, $S(i-1, w)$, if and only if it reduces the entropy of $S(i-1, w)$ while respecting the weight threshold. Formally,

$$S(i,w) = \begin{cases} S(i-1,w) & \text{if } H(S(i-1,w)) \leq \\ & H(S(i-1,w-w_i) \cup \{bb_i\}) \\ S(i-1,w-w_i) \cup \{bb_i\} & \text{otherwise} \end{cases}$$

Log20 implements this algorithm by maintaining an entropy-weight (EW) matrix, where $EW(i, w)$ shows the entropy of $S(i, w)$. Table 5 shows the EW matrix for our example. Given a weight threshold W_T, $EW(8, W_T)$ shows the computed near-optimal placement and entropy. For example, when the weight threshold is 1.00, indicating that the user expects that on average there are no more than 1.00 log entries being printed by an execution path, she should place an LPS at line 3 that leads to an entropy of 2.67. When the threshold is set to 2 log entries per execution she should log at line 3 and line 7. To achieve entropy 0.00, she should log at line 6 and 7 that results in 4 log entries per execution.

Note that this algorithm may not compute the optimal placement because such a placement of $[bb_1, bb_2, .., bb_i]$ may be a combination of a non-optimal placement of $[bb_1, bb_2, .., bb_{i-1}]$ and bb_i. In other words, the optimal placement $S_{opt}(i, w)$ could be $S'(i-1, w-w_i) \cup \{bb_i\}$, even though $H(S'(i-1, w-w_i)) > H(S(i-1, w-w_i))$. Consider the example shown in Table 6. If we assume the weight threshold is 3.67, i.e., on average we allow no more than 3.67 log entries to be printed on an execution path, then the optimal placement will be {2, 3}. However, using our dynamic algorithm we will compute $S(N, W_T)$ to be $\{1, 2\}$, because when we are computing $S(3, 3.67)$, we only consider $S(2, 3.67) = \{1, 2\}$ and $S(2, 1.67) \cup \{3\} = \{1, 3\}$.

The complexity of this algorithm is $N \times [W]$, where $[W]$ is the total number of possible weight values that could occur from a selection of basic blocks. In the worst case, $[W]$ is still 2^N since the basic block weights are not integers and there are 2^N possible selections. We solve this by scaling-up and rounding the block weights and weight threshold to make them integers. The choice of scale factor, C, allows us to trade the precision of the algorithm with its efficiency. This implementation has complexity of $N \times W_{max}$, where W_{max} is given by,

$$W_{max} = \lfloor C \times min(W_T, \sum_{i=1}^{N} w_i) \rfloor$$

The $min(W_T, \sum_{i=1}^{N} w_i)$ term is used to check if weight threshold W_T is greater than the sum of all basic blocks' weights.

An optimization we made in the implementation is to differentiate execution paths by the count of basic blocks instead of their sequences. This means that as long as two paths have the same number of basic blocks, and each block appears the same number of times, we consider them as the same path regardless of their ordering. Operations on sequences are expensive, and to compute H_S for each placement, S, we would need to compare the sequence of basic blocks in each path. Comparing paths by basic block count allows us to simply compare their columns in the basic block

count matrix, as shown in Table 4. For example, P_3, P_5, and P_6 are now considered as the same path because their corresponding columns in Table 4 are the same. When considering the different log outputs of a placement, S, we simply select the rows corresponding to the basic blocks in S.

In real systems, the utility of a log would not be significantly reduced by ignoring the order of log entries. Developers or administrators typically use three properties of log entries: the existence of particular log entries, their counts, and their sequence, ordered in decreasing usefulness for post-mortem debugging. The sequence of log entries, in particular, is often not reliable as real systems have high degrees of concurrency. Furthermore, as we show in §8, even basic block count is not as useful when it comes to path differentiation, since in most debugging tasks, one merely searches for the existence of log entries.

4.3 Placement with Variable Values

By additionally logging variable values, a single LPS can record the direction of multiple branches in a function, thus having the same power to disambiguate paths as placing multiple LPSes in multiple basic blocks. We refer to these variable-containing LPSes as dynamic LPSes and the previously discussed variable-lacking statements as static LPSes. In each LPS, we simply try to include all the variable values used in the branch conditions from the same method.

An LPS placed in a basic block can only record those branch variable values that are live, i.e., they still hold the value they had when used in the branch condition. We perform variable liveness analysis by transforming the code into a static single assignment (SSA) form and then perform a dataflow analysis. SSA form gives every variable value a single definition point and a unique variable name; thus we know exactly which value is live at each instruction. For variables that are out-of-scope, such as a function return value, we rewrite the bytecode to introduce a new temporary variable that holds the value so that it can be recorded at the basic block where an LPS is placed. Note that this can only be done if the LPS is after the function is called on a path; if the LPS is before the function call, we cannot know the function's return value and thus the value is treated as dead (unrecordable). The logged value is guaranteed to be a scalar since Java bytecode does not allow comparisons between non-scalars.

We do not perform dataflow analysis across threads, effectively making the assumption that a class member variable is not modified by other threads concurrently. We also cannot analyze the liveness of a variable from a `try` block into a `catch` block because we often do not know the exact program location that throws the exception. Therefore, we conservatively assume a variable value from `try` is not live in the `catch` block.

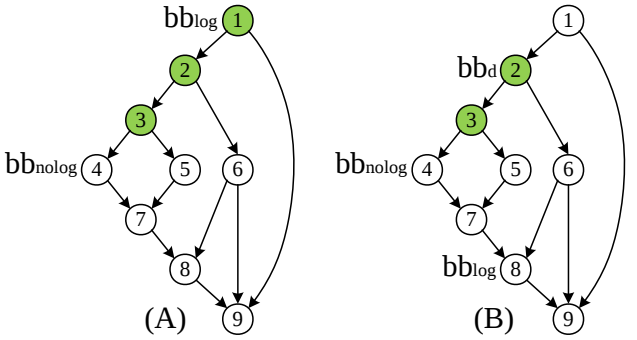

Figure 2: Two control-flow-graphs showing bb_{log} is either a dominator (A) or post-dominator (B) of bb_{nolog}. The green nodes show the branch conditions that must be disambiguated via logged variable values in bb_{log} for it to replace an LPS in bb_{nolog}.

A dynamic LPS placed in basic block bb_{log} can replace a static LPS in basic block bb_{nolog} when two constraints are satisfied: (1) bb_{log} must dominate or post-dominate bb_{nolog}; and (2) the variable values logged must unambiguously indicate whether a path from the entry block to bb_{nolog} is taken. Recall a node, N_a, dominates a node, N_b, if every path from the entry node to N_b goes through N_a. Similarly, N_a post-dominates N_b if every path from N_b to the exit node goes through N_a. Constraint (1) ensures that every path that exercises bb_{nolog} also exercises bb_{log}, so that the placed log will be printed. And if bb_{log} is not traversed, one can infer that bb_{nolog} is not traversed either. A limitation of this approach (compared to purely static LPSes) is that if the path between bb_{log} and bb_{nolog} is interrupted by an exception, the user can no longer determine if bb_{nolog} was executed. In these cases, we assume that the exception handler has logging sufficient enough to identify the source of the exception, implying whether bb_{nolog} was executed or not.

To check the second constraint, we first check whether bb_{log} dominates or post-dominates bb_{nolog}. If it dominates bb_{nolog}, bb_{log} must be able to disambiguate all branch conditions between it and bb_{nolog}. Figure 2 (A) shows this case. If any of the branch conditions cannot be disambiguated from variable values, then one cannot unambiguously determine whether bb_{nolog} is executed. If bb_{log} post-dominates bb_{nolog} (Figure 2 (B)), we first need to locate bb_d that is a dominator of both bb_{log} and bb_{nolog}. Then to satisfy the second constraint, the LPS in bb_{log} must record all branches between bb_d and bb_{nolog}. This guarantees that the dynamic LPS replaces the static LPS in bb_{nolog} because when the dynamic log is printed, we know bb_d must have occurred (as it dominates bb_{log}), and we can further infer whether the path from bb_d to bb_{nolog} is taken via the logged variable values.

We extend the placement algorithm to use these properties as follows: When considering a placement $S = \{bb_1, bb_2, .., bb_n\}$, for each bb_i, we determine whether the block and the logged variable values satisfy the constraints necessary to replace a static LPS in block, bb'_j. If so, the placement with variables has the same effect as if additional LPSes were placed in basic blocks $S' = \{bb'_1, bb'_2, .., bb'_m\}$. This allows us to use the same dynamic programming algorithm to compute the entropy of this placement using $S \cup S'$ while the weight is that of S, since we observe that the additional overhead of outputting variable values is negligible compared to the overhead of outputting even a simple log entry.

When the two constraints are not satisfied, a dynamic LPS in bb_{log} may still replace a static LPS in bb_{nolog} when differentiating a subset of paths. For example, in Figure 2 (A), assume the dynamic LPS in bb_{log} cannot record the variable values in the branch condition at basic block 2, but it can record the branch conditions at blocks 1 and 3. If we are differentiating between paths that went through basic block 2 (e.g., $\{1, 2, 3, 4\}$ and $\{1, 2, 3, 5\}$), the LPS still has the same effect as if bb_{nolog} is logged. Therefore, we further consider the changes to the disambiguated paths set, $DP(S)$, caused by additional variable values, and recalculate the entropy.

5 IMPLEMENTATION

The Log20 system consists of three major components: the instrumentation library, the tracing library that is used for both request tracing and logging, and the LPS placement generator. The systems we evaluate in §8 are all distributed systems used for processing user requests. Thus, we limit our consideration of execution paths to request processing code. In a typical use case, first the instrumentation library statically instruments the application's bytecode in every basic block of the request processing code so it invokes the tracing library. Then, at runtime, the tracing library (§6) collects traces for each request, feeding them to the LPS placement generator. The LPS placement generator applies the algorithm from §4 to generate the LPS placement strategies for the user to choose from. Finally, the instrumentation library instruments the application's bytecode again to place an LPS in each basic block from the chosen placement that invokes our tracing library to perform the actual event logging. Alternatively, developers can manually insert the LPS into the code. This process is repeated periodically to ensure the placement remains effective should the workload change.

The instrumentation library identifies entry and exit points of request processing code in a semi-automatic manner. It uses the Soot [32] static analysis framework to first statically identify event processing loops by searching for infinite loops or loops controlled by member variables within every thread's `run()` method; every method called from such a loop is identified as a request entry-point. In case this heuristic fails, e.g., requests that are processed by the Java main thread which does not have a `run()` method, we rely on the user to determine the request entry and exit points – a mostly one-time effort for mature applications. We also use Soot to instrument the bytecode.

The LPS placement generator aggregates the request traces before applying the algorithm from §4 with a few practical optimizations. In order to aggregate each sampled path into a set of unique paths and their probabilities, we must compare each sampled path with every other path. If the average path length is large, this process can be prohibitively expensive. Instead, we compute a SHA1 hash for each sampled path and perform the comparison on the hashes. We found any potential hash collisions negligibly affected the accuracy of the algorithm. The probability of each path is computed as the number of occurrences of this path in the trace, divided by the number total number of paths. When computing a set of disambiguated paths, we use a simple partition refinement algorithm instead of naively computing the intersection of every basic block's disambiguating paths. Finally, we again use Soot to statically analyze what variables are log-worthy and the basic blocks they disambiguate. Note that currently, we only analyze paths from a single node rather than stitching them together with paths from other nodes where a request may be processed by multiple nodes.

6 TRACING LIBRARY

We designed and implemented a JVM-based tracing library that collects runtime profiling information. It is used for both sampling-based request profiling and as a logging library rivaling existing ones such as Log4j 2 [21]. The tracing library can switch between tracing and logging mode at runtime. In tracing mode, it records the observed frequency of each basic block in each sampled request. In logging mode, it prints logs only in basic blocks that are included in the LPS placement.

6.1 Usage

Upon application initialization, the library will inject the following instrumentation code to the end of each basic block when used for request profiling:

```
log(MethodID#BBID);
```

Instrumentation of each request is controlled via similar instrumentation code at the start and end of the request processing code path:

```
logRequestBegin(RequestID, MethodID#BBID);
logRequestEnd(RequestID, MethodID#BBID);
```

`MethodID`, `BBID`, and `RequestID` are integer constants that are statically computed, uniquely representing a method, a basic block within that method, and a type of request (e.g., a

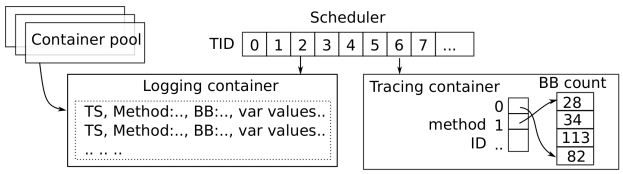

Figure 3: Architecture of the tracing library.

read block request). Inside `logRequestBegin()`, the library decides whether to turn on a flag to enable tracing of the current request based on a specified sampling rate. This sampling rate controls how often Log20 collects and updates the run-time profile, which in turn affects how quickly Log20 can react to workload changes. The `log()` method checks the flag to determine whether to log or not. When used for logging, the library statically instruments the bytecode at the target basic blocks as described in §5, with the differences that (1) it does not check for the flag but always logs the data, and (2) it uses the `MethodID` and `BBID` to index into a table that indicates which variable values to record.

Users can configure the instrumentation library dynamically via HTTP request, including setting the sampling rate of each type of request, enabling or disabling the tracing of a certain type of request, etc. Users can also patch the bytecode at runtime without restarting the application, to enable, disable, add or remove instrumentation points. Dynamic bytecode rewriting is done by reloading a modified class using a modified version of the Spring-loaded library [28].

6.2 Design

Figure 3 shows the overall architecture of our tracing library. It consists of a scheduler and multiple logging containers. Each container is a memory buffer (set to 4MB) that is used to log the data from one thread. When the application is started, the tracing library allocates a pool of containers. At the beginning of each request that is being traced, the scheduler is invoked to select a container from the pool and attaches it to the thread. The scheduler attempts to select a container that was last used by a thread executing on the same core as the current thread. If multiple containers are available, it tries to select the approximated last-recently-used one because data might still be in the cache. The scheduler maintains a container allocation table that maps a thread ID to the container. Each log point only writes data to the buffer within the container attached to this thread. At the end of the processing of a request, the scheduler returns the container to the pool.

Since more than one thread can be created to process a request, we also instrument the thread creation points, such as `Thread.start()` in Java. Consequently, any thread that is created during the processing of a traced request will also be traced. Note that even if concurrent threads are processing the same request each one writes to its own container.

This design allows our tracing library to be almost free from synchronization. Each thread writes to a unique container buffer, therefore synchronization is not necessary at each log point, and a write operation simply increments the container's end-of-buffer pointer. There is only one critical region that is used to protect the scheduler's container allocation and reclamation.

Each container operates in one of the two modes: logging or tracing. When used as logging library, it simply appends a record to the memory buffer each time that `log()` is invoked. A log entry consists of the following fields: a timestamp, `MethodID#BBID`, `threadID` and any variable values.

Tracing mode is only used for request profiling. Since Log20's LPS placement algorithm only considers the count of each basic block rather than the sequence, the container simply updates a counter for each basic block in a basic block table. The container on the right in Figure 3 shows such an example. We maintain a method table that is indexed by the method ID, and each entry stores the index of the *first* basic block of this method in the basic block table. The first time a method with a particular method ID is invoked, we allocate N_m entries in the basic block table where N_m is the number of basic blocks in this method. Whenever a log point is executed, the index of the counter for this basic block in the basic block table is computed as `method_table[MethodID] + BBID`, and the corresponding counter is incremented.

Containers are flushed to disk when they are nearly full at the end of a request. Instead of outputting data to a single log file, each container writes to a different log file. Thus, there is also no need to synchronize writes to the file. We developed a post-processing tool capable of stitching log entries from different log files into a single sequence.

7 LIMITATIONS

Other than those discussed in §1, Log20 has a few other limitations. First, the coverage of the trace is not exhaustive. Some functions or basic blocks will not be exercised by the workload, so they are excluded from LPS placement consideration. Currently, Log20 places an LPS in every unexecuted basic block, with an optimization that if a function is unexecuted, we place an LPS at the beginning of the function and include all path-disambiguating variables; additional LPSes are placed in any uncovered paths remaining.

A change of workload can result in undesirable logging behavior because Log20's placement is optimized towards the old workload. For example, when the system suddenly executes a new path, an excessive amount of log entries could be printed, incurring large overhead. To mitigate this problem, LPSes can use an adaptive sampling strategy, where

System	Sampled paths	Unique paths, measured by BB count	BB appearance	Static BBs
HDFS	2,350	1668	250	1402
HBase	7,429	26	11	50
YARN	1,281	55	34	309
ZooKeeper	38,978	16	15	63

Table 7: Systems and sampled paths used in experiments.

the sampling rate backs off exponentially (e.g., each LPS only records its 2^nth dynamic occurrence) [10].

Finally, Log20 is designed to work independently from developers' manual LPS placement. If developers have already placed LPSes in the system, Log20 can be used to suggest additional LPSes while keeping the existing ones. However, Log20 cannot 'learn' from the logging patterns in existing LPSes (e.g., which conditions are log-worthy) and apply them to place LPSes on unlogged paths.

8 EXPERIMENTAL EVALUATION

We answer the following questions in our evaluation: (1) How effective is Log20's automatic LPS placement? (2) How does it compare with developers' manual placement? (3) Do the LPSes placed by Log20 help with debugging? (4) How does it compare with the static-analysis based path profiling algorithm by Ball and Larus [2]? (5) How well does Log20 approximate an optimal placement? (6) How much overhead does our tracing library incur?

Our evaluation was setup as follows: we evaluated Log20 on four systems – HDFS, HBase, Hadoop YARN, and ZooKeeper. HiBench [13] was used to generate workloads for HDFS and YARN while we used HBase's built-in Performance Evaluation tool to generate workloads for HBase. ZooKeeper's workload was generated and evaluated using the benchmark tool from the original ZooKeeper paper [14]. For HDFS, HBase, and YARN, we ran an 11-node cluster comprising one master and 10 slaves. We used a 10 node cluster for ZooKeeper because leader-node promotion is dynamic.

To obtain the path profiles, we monitored the systems using our tracing library. The number of paths collected is shown in Table 7. Each path is the execution of a request on one node. For HDFS, HBase, and YARN, we used the trace from the slave (datanode, region server, and node manager respectively). We chose slave nodes instead of the master because slaves are on the critical path of most user requests, requiring a critical balance between informativeness and the overhead of logging. ZooKeeper is a decentralized system, so we randomly selected a node when analyzing its trace.

Table 7 shows the counted number of different execution paths in two ways: (1) by the count of basic blocks, and (2) by only considering basic block appearances. The large variance in the number of paths under these two approaches is a result of data-processing loops. For example, HDFS contains a loop that processes a batch of data per iteration. This results in a large number of unique paths when they are distinguished by the basic block count. This further leads to a high entropy value that can only be eliminated if we record the number of iterations of that loop. Log20 is able to disambiguate different paths based on both basic block appearance and basic block execution counts. Our evaluation result is mainly based on appearance because we believe developers usually look for the appearance of log entries rather than their counts during postmortem debugging. Nevertheless, we also show the count-based result for HDFS.

8.1 Entropy versus Overhead

The curves in Figure 4 illustrate the relationship between entropy and overhead. Overhead is measured as the expected number of log entries per request. On average, the execution of each LPS takes 1.5 μs in our system using Log4j, the default logging library used by the evaluated systems. With these curves, developers can choose the right balance between the informativeness and overhead, measured either by number of log entries per request or by slowdown.

The reduction of entropy has a non-linear relationship with the increase in overhead. At the beginning, with relatively small overhead, Log20 can compute a near-optimal placement that quickly reduces the entropy as the overhead is increased. Later additions of LPSes have diminishing returns in reducing the entropy. Fundamentally, this is because the production workload is highly skewed. There are just a small number of hot paths while the vast majority of paths are cold. For example, in our HDFS workload, 20% of the unique paths (by basic block appearance) account for 78% of all paths are sampled. Similarly, at the basic block level, the weight distribution is also skewed – a small number of basic blocks appear the majority the time across the trace. Consequently, under a small overhead threshold, Log20 infers a placement that covers the cold paths and basic blocks, and can quickly reduce the entropy, since there is only a small number of unique hot paths. However, to further reduce entropy, we need to place LPSes in hot paths to further differentiate them.

This non-linear relationship further suggests the importance of a good placement strategy. A sophisticated placement strategy could eliminate a majority of the uncertainty with relatively little overhead, whereas a less optimal strategy could incur large overheads without offering meaningful information. For example, in HDFS, Log20's placement can reduce the entropy from 6.41 to 0.91 with fewer than two log entries per request. Intuitively, this means that with two log entries per request, Log20 can reduce the number of possible paths from $2^{6.41} \approx 85$ to $2^{0.91} \approx 2$.

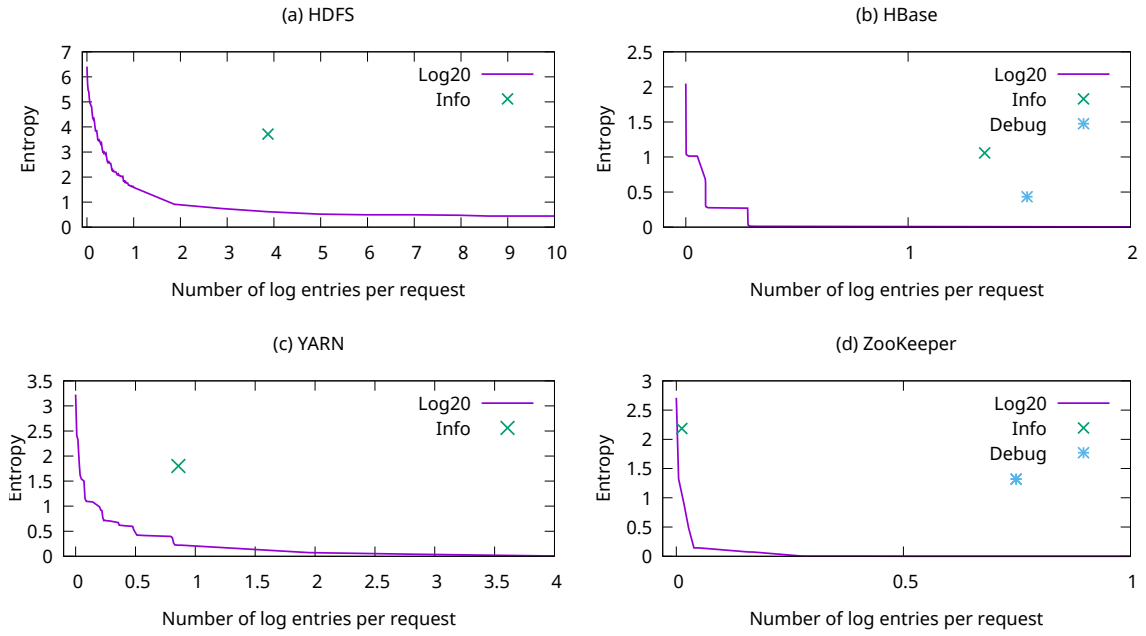

Figure 4: Entropy versus overhead. The x-axis shows the overhead, measured in average number of log entries per request. The y-axis shows the entropy. We also show the entropy and overhead of existing logs. For HDFS and YARN, we only show the Info verbosity because Debug verbosity outputs thousands of entries that are outside of the range.

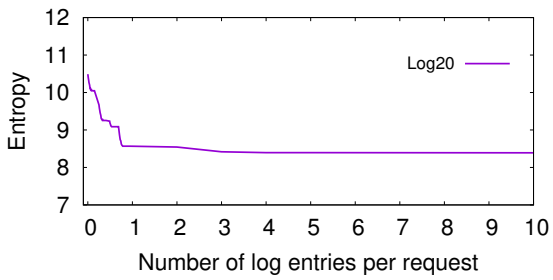

Figure 5: The entropy-overhead trade-off for HDFS when paths are differentiated by basic block counts. Most of the remaining entropy is caused by data-processing loops.

Figure 5 illustrates the entropy-overhead relationship for HDFS when execution paths are differentiated by basic block counts. Due to the high cost of recording the number of iterations in each data-processing loop, Log20 can only reduce a small amount of entropy within the threshold of 10 log entries per request.

Comparison with existing LPS placements. Figure 4 shows the entropy and overhead of the existing LPS placements in each system. Log20 is substantially more efficient in disambiguating code paths compared to the existing placements. Table 8 further illustrates this point. To be as informative as existing Info logs, Log20's placement only outputs 0.08 log entries per request, compared with the 1.58 log entries

Num. of log entries	Info			Debug		
	Log20	existing	%	Log20	existing	%
HDFS	0.22	3.87	5%	0.32	2434.92	0%
YARN	0.03	0.86	3%	0.14	13.69	1%
ZooKeeper	0.005	0.012	41%	0.02	1.31	1%
Average	0.08	1.58	17%	0.16	816.98	1%

Table 8: Comparing the overhead, as measured by the average number of log entries per request, between the existing LPS placement and the placement generated by Log20 that has the same entropy as the existing LPS. % is computed as Log20/existing.

Entropy	Info		Debug	
	Log20	existing	Log20	existing
HDFS	0.61	3.71	0.16	3.21
YARN	0.23	1.80	0.00	1.08
ZooKeeper	1.32	2.18	0.00	0.74
Average	0.72	2.57	0.05	1.68

Table 9: Comparing the informativeness of LPS placements between Log20 and developers' manual effort, under the same overhead threshold.

required by the existing Info log placement. The difference is even more substantial when compared with Debug-level log output. For example, HDFS outputs an average of 2434.92 log entries per request under Debug verbosity; In contrast,

Existing (H = 3.71)			Log20 (H = 0.63)		
Location	w	H	Location	w	H
writeBlock:677	0.58	5.42	BlockSender:342	0.10	5.54
writeBlock:715	0.83	5.44	convert:376	0.27	5.58
writeBlock:593	0.85	5.47	sendBlock:706	0.12	5.58
receivePacket:616	0.54	5.65	receivePacket:616	0.54	5.65
receivePacket:520	1.04	5.74	add:158	0.06	5.68
writeBlock:754	0.01	6.16	close:298	0.11	5.72
flushOrSync:386	0.00	6.18	receivePacket:520	1.04	5.74

Table 10: Comparison between the LPSes placed by developers (existing) and by Log20 on HDFS (v2.6.0). HDFS datanode only outputs 7 types of log entries under the default verbosity in our sampled traces. All of them are shown. Log20's placement is computed under the same overhead threshold as existing Info log. The LPSes are ranked by H, which is the entropy of the placement that only contains that one LPS. 23 LPSes under Log20's placement are exercised; we only show the top 7. w is the average number of times the LPS gets executed per request. The format of the location is "method name:line number".

Log20's placement only outputs 0.32 log entries per request to achieve the same entropy.

Table 9 further compares the entropies of the existing logs and Log20's placement computed with the same overhead threshold as the existing logs. Log20 significantly reduces the amount of uncertainty compared with developers' manual placement effort. In particular, Log20's placement can achieve 0 entropy, i.e., disambiguate all paths, when given the same overhead threshold as existing Debug-level logs in YARN and ZooKeeper.

In Table 10, we further zoom into HDFS to compare each LPS placed by Log20 and developers' manual effort under the same overhead threshold. Log20's placement achieves much lower entropy (0.63) compared to the existing Info logs. This is because Log20 carefully places LPSes at basic blocks that have a low weight yet reduce the most amount of entropy when an LPS is placed within them. Moreover, Log20 avoids placing LPSes on hot paths. For example, consider the code for "add:158":

```
157 public void add(double value) {
158   if (value > max) max = value;
159   if (value < min) min = value;
160 }
```

Logging at the true branch at line 158 can reduce the entropy from 6.41 to 5.68, yet the log entry only appears 0.06 times per request. This method is used to compute the maximum and minimum latency of readBlock and writeBlock requests. In 2350 requests, the value of max was updated only 143 times. In fact, Log20 also places an LPS at the true branch at line 159 that updates the value of min.[2] Consequently, Log20's placement is able output 23 different types of log entries, compared to only 7 in the existing logs.

	Instrumentation points executed		
System	Ball-Larus	Log20	Ball-Larus/Log20
HDFS	310056	104089	3.0X
HBase	14.98	4.98	3.0X
YARN	527.73	11.63	45.4X
ZooKeeper	12.74	0.77	16.5X

Table 11: The average number of instrumentation points executed by Ball-Larus algorithm and Log20 (zero-entropy).

Interestingly, despite the differences, 3 out of the 7 existing LPSes that are executed in our workload overlap with Log20's placement (see the highlighted entries in Table 10), suggesting that Log20 matches developers' intuition. All three are warning messages that record (1) when disk write takes longer than a threshold, (2) when a network request is longer than a threshold, and (3) when flushing the buffered data to disk takes longer than a threshold.

Comparison with Ball-Larus path profiling. Ball and Larus proposed an algorithm to instrument a target program to collect a trace that can determine how many times *each* execution path runs [2]. This trace has the same effect as a zero-entropy LPS placement. The algorithm works by carefully assigning an integer value to each edge on the control-flow-graph such that the sum of the edge values traversed in different paths is different. The basic blocks are instrumented to update the sum value, and each path outputs this sum at its end. It does not need to instrument every block, because those with only 0-valued incoming edges do not need to be instrumented as they do not need to update the sum. It uses runtime path profile to avoid instrumenting the most frequently executed basic blocks.

We implemented the Ball-Larus algorithm, and compared it with Log20's zero-entropy LPS placement. We focus on the number of instrumentation points executed. For Log20, this is the number of log entries. Table 11 shows the result. Log20's instrumentation is at least 3X more efficient than that of the Ball-Larus algorithm. This is because Ball-Larus algorithm uses a static approach; it aims to differentiate every *possible* execution path. In comparison, Log20 only needs to place LPSes to differentiate the execution paths in the trace sampled from the current workload, which is orders of magnitude smaller than the total number of possible paths. In addition, variable values are used by Log20's LPSes to further disambiguate paths.

[2]We filed a bug report with the HDFS developers, suggesting that the two LPSes be added [11]. Developers confirmed their usefulness and agreed to add them. We are currently working on a patch as developers demand throttling control and recording the calling context in the log message.

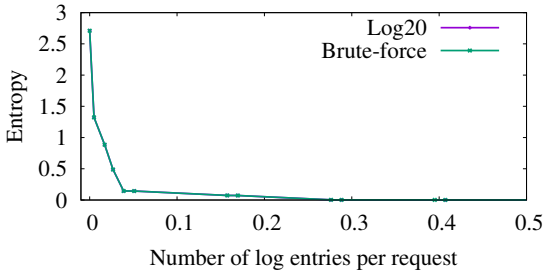

Figure 6: Entropy-overhead curves generated by Log20 and a brute-force (optimal) algorithm for ZooKeeper.

	Avg. num. of variables	Avg. num. of BBs replaced
HDFS	0.551	0.774
HBase	0.563	0.802
YARN	0.536	0.719
ZooKeeper	0.587	0.807
Average	0.559	0.776

Table 12: The number of variables included per LPS and the number of basic blocks that can be disambiguated by them.

Note that in this experiment we only compared Ball-Larus' result with Log20's LPS placement (under logging mode) instead of its instrumentation under tracing mode. Tracing mode and Ball-Larus' algorithm will have similar results as we instrument every basic block. However, Log20's tracing need not be enabled all the time since the traces are only used to compute LPS placements for the current workload. Thus, Log20 can be configured to trace every *n*th request, balancing the accuracy of the trace with its overhead. For I/O-bound workloads, we found sampling unnecessary since the tracing overhead was less than 1% (see §8.3) due to the efficiency of our library.

Comparison to an optimal placement. Figure 6 shows the entropy-overhead curves generated by Log20 and an optimal, but brute-force algorithm which enumerates every possible placement. We used ZooKeeper because it has the smallest number of basic blocks (63) in its trace. Even then, the optimal algorithm was still not able to finish; we had to restrict the algorithm to the first 25 blocks in the block count matrix. For 9 out of 13 data points, we see that Log20 indeed matches the optimal placement. For the remaining 4 points, Log20 differs by a negligible amount (< 0.0001 in entropy).

Effectiveness of variable logging. Table 12 shows the effectiveness of Log20's variable logging. On average, an LPS at each basic block can record 0.559 variable values, which in turn can be used to replace 0.776 static LPSes in other basic blocks. This suggests every branch variable logged in a basic block can disambiguate at least one other block.

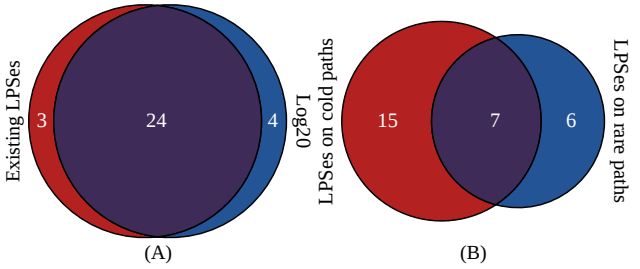

Figure 7: The usefulness of logs in debugging 41 HDFS failures. (A) shows the number of cases where the existing LPSes or Log20 is helpful. (B) shows the number of cases where helpful LPSes placed by Log20 are on cold or rare paths.

8.2 Debugging Real-world Failures

We evaluated the usefulness of Log20 for debugging using 41 randomly selected user-reported HDFS failures, sourced from a prior work that studied distributed systems failures [34]. We carefully analyzed each failure to understand its propagation path from the fault (i.e., root cause) to the failure symptom. We then examined whether the LPSes placed by Log20, subject to the same performance threshold as existing Info logs, helped in narrowing down the failure propagation paths. We manually reproduced 33 of the 41 failures.

Figure 7 shows the result. Overall, we found Log20 is helpful in debugging 68% (28/41) of the real-world failures. In comparison, existing default verbosity LPSes are helpful in 27 cases. Although Log20 only marginally outperforms manual logging, manual logging is loosely based on trial-and-error where many LPSes are added as after-thoughts (see §2) while Log20 is fully automated.

In 24 of the failures, both Log20 and existing LPSes help with debugging. In fact, in 17 of them, Log20 places LPSes in the same basic blocks as existing Error LPSes because these blocks were not exercised by the traces from our production workload (i.e., cold paths). Typically, these are (buggy) exception handling paths that are only exercised in failure scenarios. As described in §7, Log20 generates a zero-entropy placement for all paths that were not covered by the trace, allowing developers to reconstruct the exact execution flow in these cases.

There are four failures where Log20 helps but existing LPSes do not. In these cases, existing LPSes are not placed on the failure propagation path; nor are they placed on paths that are not executed but are relevant to the failure (so that the absence of their output can be used to narrow down the bug), leaving developers searching for root causes "in the dark". There are three bugs where existing LPSes help but Log20 does not. In one of them (HDFS-4660), an existing LPS outputs the size of a file block on the error path, which is critical for diagnosis. Log20 also places an LPS on this error path, but fails to include the block size because it is not used

in any branch conditions. In the other two bugs, the useful logs are output by the HDFS client which is not traced nor analyzed by Log20.

Figure 7 (B) further breaks down the 28 cases where Log20 helps with debugging. In 22 of them, the helpful LPSes are placed on cold paths. More interestingly, in 13 bugs, the helpful LPSes are in basic blocks that are covered by our trace. These are normal but rarely executed blocks, and these LPSes significantly reduce entropy without incurring large overhead.

Next we discuss two failure case studies.

Case 1: HDFS-4328. In this failure, the entire HDFS cluster becomes unresponsive during system shutdown. There are no existing LPSes printed on this failure path, and it is hard to infer anything from the absence of log entries. In contrast, when we reproduced this failure with Log20's placed LPSes, two log entries were printed repeatedly. The first is in the following function:

```
private boolean isLongRead() {
  return (endOffset - initialOffset) > LONG_READ;
}
```

This function returns true if the size of a single file block is larger than a threshold, so that HDFS will advise the OS to access the block sequentially using `posix_fadvise (POSIX_FADV_SEQUENTIAL)`. Log20 places an LPS only when this function returns true, which is a rare condition but did occur in the workload we used. (Recall that Log20 works on bytecode where the true and false returns are in different basic blocks.)

The second repeated log entry is from this snippet:

```
while (..) {
    try { wait ( curPeriodEnd - now ); }
    catch (InterruptedException ignored) {}
}
```

Log20 places an LPS in the block that catches the InterruptedException. In this code, the HDFS block scanner thread is waiting for the thread that actually performs the block read to finish. In case of a shutdown, it will receive an InterruptedException; however, it mistakenly ignores the exception and goes back to `wait()`. This causes the entire HDFS cluster to hang when the user shuts down HDFS while the block scanner thread is scanning a large block. The fix is to handle this InterruptedException appropriately instead of ignoring it. The LPSes placed by Log20 clearly point to this failure path – the first log entry indicates that the block scanner thread is reading a large block, and the second log entry leads to the incorrect handling of the InterruptedException.

Case 2: HDFS-4182. This case demonstrates how the absence of a log entry helps with debugging. In this failure, the Secondary Namenode has a resource leak of NameCache entries. There is one NameCache for each directory, and it is supposed to be populated only during HDFS' initialization phase. There is a function, `initialized()`, that should be invoked at the end of the initialization to clear the NameCaches loaded from the last HDFS checkpoint image. Although the Namenode correctly calls `initialized()` at the end of the initialization stage, the Secondary Namenode never invokes `initialized()`, which leads to the leak of NameCache entries. Log20 places an LPS in `initialized()` as it is a rare path on the Namenode. We can infer from the absence of this log entry on the Secondary Namenode that the leak occurred, since `initialized()` was not invoked.

	Logging mode	Tracing mode		
		no-flush	flush-long	flush-short
Latency	43 ns	24 ns	25 ns	35 ns

Table 13: Performance of the tracing library.

8.3 Performance of the Tracing Library

Microbenchmark. We first measure the execution time of each invocation to our tracing library in logging mode. We created a benchmark following the same method that was used by the Log4j 2 developers to benchmark Log4j 2 [26]. This benchmark first warms up the JVM [19], and then uses a loop to invoke the tracing library 10 million times. Each logging invocation outputs an average of 18.23 bytes of data, which is the same as the average log output on the four real systems. We set the size of the in-memory buffer to 4MB, and it is flushed whenever it is filled. The latency of each invocation is measured by the total execution time divided by 10 million. The experiment is done on a server with Intel Xeon E5-2630V3 2.4GHz CPUs and 128GB DDR4 RAM.

As shown in Table 13, each logging invocation using our tracing library takes 43ns on average. This is significantly faster than Log4j, which takes 1.5 μs per LPS.

We further measure the performance of the tracing library when it operates in tracing mode. Recall that under this mode, it only flushes the buffer at the end of the request. Therefore, the average latency per log depends on the latency of a request – a long request leads to less frequent flushes. Therefore, we created two types of requests: a short request that consists of 1,750 basic blocks being executed, and long request that consists of 35,000 basic blocks. In both requests, we invoke our library in every basic block. We repetitively execute each request until we collect 10 million invocations to the tracing library.

Table 13 shows this result. When we turn off flushing, the latency of each invocation to our tracing library is 24 ns. If we use it to trace a long request where we flush the buffer at the end, this latency becomes 25 ns. Finally, if we perform the same experiment with the short request, this latency grows to 35 ns.

	Base	Tracing
HDFS read (1GB)	16.26s	16.24s (-0.2%)
HDFS write (1GB)	21.02s	21.20s (0.9%)

Table 14: Tracing overhead on HDFS.

Real system evaluation. Given the efficiency of our tracing library, it imposes only negligible overhead when used to trace requests in real distributed systems. For example, as shown in Table 14, it imposes less than 1% slowdown on HDFS even when we include the flush time. Interestingly, we observe that the traced read consistently outperforms the uninstrumented read. The reason is that when we use Soot to instrument bytecode, it further rewrites the bytecode using a variety of optimizations [32].

9 RELATED WORK

LPS placement. Errlog [35] automatically places error LPSes in a small set of generic error patterns (e.g., system call error returns). Fu *et al.* [9] presented an empirical study of logging practices at Microsoft and identified a set of program patterns that are considered log-worthy. Li *et al.* [18] enhanced the work of Fu *et al.* and offered suggestions to place LPSes based on existing placements, using a machine learning algorithm without considering performance overheads. In contrast, Log20 does not rely on existing LPSes. It automatically computes a placement with near-optimal entropy under a specified overhead threshold. In a position paper [38], we proposed the idea of using information theory to guide LPS placement. This paper further formalizes the path disambiguation problem based on set theories, proposes a concrete algorithm, and provides a full implementation.

Log enhancement. Other works enhance existing LPSes. LogEnhancer [37] includes additional variable values in each LPS to enhance its informativeness. Our variable value logging is similar, but simpler. Unlike LogEnhancer, we do not perform inter-procedural analysis and thus we are not aware of variable values that can be used to replace static LPSes in other methods. Yuan *et al.* proposed a tool to adjust the verbosity of each LPS [36]. These works are complementary to Log20 because they address different aspects of log automation.

Path profiling and customized program coverage. As discussed in §8.1, Ball and Larus [2] proposed a path profiling algorithm that disambiguates every path. Larus later extended this method by adding latency information and the ability to trace inter-procedurally [16]. Ohmann *et al.* [24] formalized the problem of customized program coverage, that is to instrument the program while respecting a certain set of constraints in order to generate coverage information for a given set of program points. Their practical methods used static analysis and approximation algorithms to compute placements that minimized the runtime of instrumentation.

Log20 is complementary to these approaches. These approaches are based on static analysis so they are capable of differentiating *all* paths independent of the runtime workload. Therefore, they can be used by Log20 in placing LPSes in basic blocks that are not exercised by the trace. However, Log20's use of runtime tracing means that its placement can be more efficient as it adapts to the workload (as shown in §8.1). In addition, in practice developers must consider the trade-off between overhead and informativeness. Log20's application of entropy allows it to generate a near-optimal placement under *any* overhead threshold, a unique feature that does not exist in any prior work.

Tracing frameworks, including DTrace [4], SystemTap [30], X-Trace [8], MagPie [3], Pivot Tracing [22], just to name a few, allow developers to write scripts or queries to collect a system's runtime statistics. Instead of relying on manual work, Log20 automatically infers the instrumentation locations that are the most informative. The inferred placement can be further leveraged by these tracing frameworks to collect more informative traces with less overhead.

10 CONCLUDING REMARKS

This paper introduced Log20 that can automate the placement of log printing statements in software programs. Guided by information theory, it measures how effective each logging statement is in disambiguating code paths. We have shown that the placement strategy inferred by Log20 is significantly more efficient in path disambiguation than the placement of log printing statements in existing programs.

ACKNOWLEDGEMENTS

We thank our shepherd, Junfeng Yang, and the anonymous reviewers for their insightful feedback. In particular, our shepherd has provided invaluable comments through multiple revision iterations; these comments significantly improved the paper. This research is supported by an NSERC Discovery grant, a NetApp Faculty Fellowship, a MITACS grant, and a Huawei contract.

REFERENCES

[1] G. Altekar and I. Stoica. ODR: Output-deterministic Replay for Multi-core Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206. ACM, 2009.

[2] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO '96, pages 46–57. IEEE Computer Society, 1996.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI '04, pages 259–272. USENIX Association, 2004.

[4] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the 10th USENIX*

Annual Technical Conference, USENIX ATC '04, pages 15–28. USENIX Association, 2004.

[5] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405. ACM, 2013.

[6] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 525–540. USENIX Association, 2014.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 211–224. ACM, 2002.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI '07, pages 271–284. USENIX Association, 2007.

[9] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion '14, pages 24–33. ACM, 2014.

[10] M. Hauswirth and T. M. Chilimbi. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, pages 156–164. ACM, 2004.

[11] HDFS-12332: Logging Improvement for SampleStat Function MinMax.add. https://issues.apache.org/jira/browse/HDFS-12332.

[12] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 207–216. ACM, 2010.

[13] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *26th International Conference on Data Engineering Workshops*, ICDEW '10, pages 41–51. IEEE Computer Society, 2010.

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 16th USENIX Annual Technical Conference*, USENIX ATC '10, pages 145–158. USENIX Association, 2010.

[15] JavaParser: Process Java Code Programmatically. http://javaparser.org/.

[16] J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269. ACM, 1999.

[17] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics Doklady*, 10(8), pages 707–710, 1966.

[18] H. Li, W. Shang, Y. Zou, and A. E. Hassan. Towards Just-in-time Suggestions for Log Changes. *Empirical Software Engineering*, pages 1–35, 2016.

[19] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems . In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 383–400. USENIX Association, 2016.

[20] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336. ACM, 2011.

[21] Log4j - Log4j 2 Guide - Apache Log4j 2. http://logging.apache.org/log4j/2.x/.

[22] J. Mace, R. Roelke, and R. Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393. ACM, 2015.

[23] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295. IEEE Computer Society, 2005.

[24] P. Ohmann, D. B. Brown, N. Neelakandan, J. Linderoth, and B. Liblit. Optimizing Customized Program Coverage. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 27–38. ACM, 2016.

[25] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192. ACM, 2009.

[26] Performance of Log4j 2. https://logging.apache.org/log4j/log4j-2.2/performance.html.

[27] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(4):623–656, 1948.

[28] Spring Loaded. https://github.com/spring-projects/spring-loaded.

[29] D. Subhraveti and J. Nieh. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 109–120. ACM, 2011.

[30] SystemTap. https://sourceware.org/systemtap/.

[31] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-time Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, ECRTS 2000, pages 265–272, 2000.

[32] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java Bytecode Optimization Framework. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 214–224. IBM Corp., 2010.

[33] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 15–26. ACM, 2011.

[34] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 249–265. USENIX Association, 2014.

[35] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, OSDI '12, pages 293–306. USENIX Association, 2012.

[36] D. Yuan, S. Park, and Y. Zhou. Characterising Logging Practices in Open-Source Software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112. IEEE Press, 2012.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 3–14. ACM, 2011.

[38] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. The Game of Twenty Questions: Do You Know Where to Log? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 125–131. ACM, 2017.