

**lprof:** A Non-intrusive Request Flow Profiler for Distributed Systems

Xu Zhao†, Yongle Zhang†, David Lion, Muhammad FaizanUllah, Yu Luo, Ding Yuan, Michael Stumm
University of Toronto

Abstract

Applications implementing cloud services, such as HDFS, Hadoop YARN, Cassandra, and HBase, are mostly built as distributed systems designed to scale. In order to analyze and debug the performance of these systems effectively and efficiently, it is essential to understand the performance behavior of service requests, both in aggregate and individually.

*lprof* is a profiling tool that automatically reconstructs the execution flow of each request in a distributed application. In contrast to existing approaches that require instrumentation, *lprof* infers the request-flow entirely from runtime logs and thus does not require any modifications to source code. *lprof* first statically analyzes an application’s binary code to infer how logs can be parsed so that the dispersed and intertwined log entries can be stitched together and associated to specific individual requests.

We validate *lprof* using the four widely used distributed services mentioned above. Our evaluation shows *lprof*’s precision in request extraction is 90%, and *lprof* is helpful in diagnosing 65% of the sampled real-world performance anomalies.

1 Introduction

Tools that analyze the performance behaviors of distributed systems are particularly useful; for example, they can be used to make more efficient use of hardware resources or to enhance the user experience. Optimizing performance can notably reduce data center costs for large organizations, and it has been shown that user response times have significant business impact [2].

In this paper, we present the design and implementation of *lprof*, a novel non-intrusive profiling tool aimed at analyzing and debugging the performance of distributed systems. *lprof* is novel in that (i) it does not require instrumentation or modifications to source code, but instead extracts information from the logs output during the course of normal system operation, and (ii) it is capable of automatically identifying, from the logs, each request and profile its performance behavior. Specifically, *lprof* is capable of reconstructing how each service request is processed as it invokes methods, uses helper threads, and invokes remote services on other nodes. We demonstrate that *lprof* is easy and practical to use, and that it is capable of diagnosing performance issues that existing solutions are not able to diagnose without instrumentation.

*lprof* outputs a database table with one line per request. Each entry includes (i) the type of the request, (ii) the starting and ending timestamps of the request, (iii) a list of nodes the request traversed along with the starting and ending timestamps at each node, and (iv) a list of the major methods that were called while processing the request. This table can be used to analyze the system’s performance behavior; for example, it can be SQL-queried to generate *gprof*-like output [16], to graphically display latency trends over time for each type of service request, to graphically display average/high/low latencies per node, or to mine the data for anomalies. Section 2 provides a detailed example of how *lprof* might be used in practice.

Three observations led us to our work on *lprof*. First, existing tools to analyze and debug the performance of distributed systems are limited. For example, IT-level tools, such as Nagios [30], Zabbix [46], and OpsView [33], capture OS and hardware counter statistics, but do not relate them to higher-level operations such as service requests. A number of existing profiling tools rely on instrumentation; examples include *gprof* [16] that profiles applications by sampling function invocation points; *MagPie* [3], Project 5 [1], and X-Trace [14] that instrument the application as well as the network stack to monitor network communication; and commercial solutions such as Dapper [36], Boundary [5], and NewRelic [31]. As these tools require modifications to the software stack, the added performance overhead can be problematic for systems deployed in production. Recently, a number of tools applied machine learning techniques to analyze logs [29, 42], primarily to identify performance anomalies. Although such techniques can be effective in detecting individual anomalies, they often require separate correct and issue-laden runs, they do not relate anomalies to higher-level operations, and they are unable to detect *slowdown creep*.1

Our second observation is that performance analysis and debugging are generally given low priority in most organizations practicing agile development and deployment: each software update might potentially introduce some marginal additional performance overhead (e.g., < 1%) that would not be noticeable in performance testing. However, with many frequent software releases, these individual slowdowns can add up to become significant over time.

---

1 Slowdown creep is an issue encountered in organizations practicing agile development and deployment: each software update might potentially introduce some marginal additional performance overhead (e.g., < 1%) that would not be noticeable in performance testing. However, with many frequent software releases, these individual slowdowns can add up to become significant over time.
organizations. This makes having a suitable tool that is
easy and efficient to use more critical, and we find that
none of the existing tools fit the bill. Performance anal-
ysis and debugging are given low priority for a number
of reasons. Most developers prefer generating new func-
tionality or fixing functional bugs. This behavior is also
encouraged by aggressive release deadlines and company
incentive systems. Investigating potential performance
issues is frequently deferred because they can often eas-
ily be hidden by simply adding more hardware due to the
horizontal scalability of these systems. Moreover, un-
derstanding the performance behavior of these systems
is hard because the service is (i) distributed across many
nodes, (ii) composed of multiple sub-systems (e.g., front-
end, application, caching, and database services), and
(iii) implemented with many threads/processes running
with a high degree of concurrency.

Our third observation is that distributed systems imple-
menting internet services tend to output a lot of log state-
ments rich with useful information during their normal
execution, even at the default verbosity.2 Developers add
numerous log output statements to allow for failure di-
agnosis and reproduction, and these statements are rarely
removed [45]. This is evidenced by the fact that 81% of
all statically found threads in HDFS, Hadoop Yarn,
Cassandra, and HBase contains log printing statements
of default verbosity in non-exception-handling code, and
by the fact that Facebook has accumulated petabytes of
log data [13]. In this paper we show that the information
in the logs is sufficiently rich to allow the recovering of
the inherent structure of the dispersed and intermingled
log output messages, thus enabling useful performance
profilers like lprof.

Extracting the per-request performance information
from logs is non-trivial. The challenges include: (i) the
log output messages typically consist of unstructured
free-form text, (ii) the logs are distributed across the
nodes of the system with each node containing the lo-
cally produced output, (iii) the log output messages from
multiple requests and threads are intertwined within each
log file, and (iv) the size of the log files is large.

To interpret and stitch together the dispersed and in-
tertwined log output messages, thus enabling useful performance
profilers like lprof.

2This is in contrast to single-component servers that tend to limit log
output [44]. Distributed systems typically output many log messages,
in part because these systems are difficult to functionally debug, and in
part because distributed systems, being horizontally scalable, are less
sensitive to latency caused by the attendant I/O.
in each specific request. Such identifiers can help asso-
ciate log messages to individual requests. Since in prac-
tice an identifier may not exist in log messages or may
not be unique to each request, static analysis further
captures the temporal relationships between log printing
statements. Finally, static analysis identifies control paths
across different local and remote threads. The information
obtained from static analysis is then used by lprof’s
parallel log processing component, which is implemented
as a MapReduce [12] job.

The design of lprof has the following attributes:
• Non-intrusive: It does not modify any part of the exist-
ing production software stack. This makes it suitable
for profiling production systems.
• In-situ and scalable analysis: The Map function in
lprof’s MapReduce log processing job first stitches to-
gether the printed log messages from the same request
on the same node where the logs are stored, which
requires only one linear scan of each log file. Only sum-
mary information from the log file and only from re-
quests that traverse multiple nodes is sent over the net-
work in the shuffling phase to the reduce function. This
avoids sending the logs over the network to a central-
ized location to perform the analysis, which is unreal-
istic in real-world clusters [27].
• Compact representation allowing historical analysis:
lprof stores the extracted information related to each
request in a compact form so that it can be retained per-
manently. This allows historical analysis where current
performance behavior can be compared to the behavior
at a previous point of time (which is needed to detect
slowdown creep).
• Loss-tolerant: lprof’s analysis is not sensitive to the
loss of data. If the logs of a few nodes are not avail-
able, lprof simply discards their input. At worst, this
leads to some inaccuracies for the requests involving
those nodes, but won’t affect the analysis of requests
not involving those nodes.

This paper makes the following contributions. First,
we show that the standard logs of many systems con-
tain sufficient information to be able to extract the perfor-
ance behavior of any service-level request. Section 2
gives a detailed example of the type of information that is
possible to extract from the logs and how this information
can be used to diagnose and debug performance issues.
Secondly, we describe the design and implementation of
lprof. Section 3 provides a high-level overview, while
Sections 4 and 5 describe details of lprof’s static anal-
ysis and how the logs are processed. Finally, Section 6
evaluates the techniques presented in this paper. We val-
diated lprof using four widely-used distributed systems:
HDFS, Hadoop YARN, Cassandra, and HBase. We show
that lprof performs and scales well, and that it is able to

2
We discuss the limitations of \textit{lprof} in Section 7 and close with related work and concluding remarks.

\section{Motivating Example}

To illustrate how \textit{lprof}’s request flow analysis might be used in practice, we selected a performance issue reported by a (real) user \cite{20} and reproduced the anomaly on a 25-node cluster.

In this example, an HDFS user suspects that the system has become slow after a software upgrade. Applying \textit{lprof} to analyze the logs of the cluster produces a request table as shown in Figure 1. The user can perform various queries on this table. For example, she can examine trends in request latencies for various request types over time, or she can count the number of times each request is processed during a time interval. Figures 2 (a) and (b) show how \textit{lprof} visualizes these results.\footnote{We envision that \textit{lprof} is run periodically to process the log messages generated since its previous run, appending the new entries to the table and keeping them forever to enable historical analysis and debug problems like performance creep. If space is a concern, then instead of generating one table entry per request, \textit{lprof} can generate one table entry per time interval and request type, each containing attendant statistical information (e.g., count, average/high/low timestamps, etc.).}

Figure 2 (a) clearly shows an anomaly with writeBlock requests at around 23:42. A sudden increase in writeBlock’s latency is clearly visible while the latencies of the other requests remain unchanged. The user might suspect this latency increase is caused by a few nodes that are “stragglers” due to an unbalanced workload or a network problem. To determine whether this is the case, the user compares the latencies of each writeBlock request after 23:42 across the different nodes. This is shown in Figure 2 (c), which suggests no individual node is abnormal.

The user might then want to compare a few single requests before and after 23:42. This can be done by selecting corresponding rows from the database and comparing the per-node latency between an anomalous request and a healthy one. Figure 2 (d) visualizes the latency incurred on different nodes for two write requests: one before 23:42 (healthy) and the other after (anomalous). The figure shows that for both requests, latency is highest on the first node and lowest on the third node. HDFS has each block replicated on three data nodes (DNs), and each writeBlock request is processed as a pipeline across the three DNAs: DN1 updates the local replica, sends it to DN2, and only returns to the user after DN2’s response is received. Therefore the latency of DN2 includes the latency on DN3 plus the network communication time between DN2 and DN3.

The figure also shows that the latency of one request is clearly higher than the latency of the second request on the first two DNAs. This leads to the hypothesis that code changes are responsible for the latency increase. The HDFS cluster was indeed upgraded between the servicing of the two requests (from version 2.0.0 to 2.0.2). The log sequence identifier is then used to identify the code path taken by both requests, and a diff on the two versions of the source code reveals that an extra socket write between DNAs was introduced in version 2.0.2. The HDFS developers later fixed this performance issue by combining both socket writes into one \cite{20}.

Figure 2 (b) shows another performance anomaly: the number of verifyBlock requests is suspiciously high. Further queries on the request database suggest that before the upgrade, verifyBlock requests appear once every 5 seconds on every datanode, generating a lot of log messages, while after the upgrade, they appear only rarely. Interestingly, we noticed this accidentally in our experiments. Clearly \textit{lprof} is useful in detecting and diagnosing this case as well.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|c|c|}
\hline
Request type & start timestamp & end timestamp & IP & nodes traversed & log sequence ID \\
\hline
\hline
\end{tabular}
\caption{One row of the request table constructed by \textit{lprof} containing information related to one request. The “node traversed” column family \cite{7} contains the IP address, the starting and ending timestamp on each node this request traversed. In this case, the HDFS writeBlock request traverses three nodes. The “log sequence ID” column contains a hash value that can be used to index into another table containing the sequence of log printing statements executed by this request.}
\end{table}
1. Log format-string and variable parsing
2. Request entry and identifier analysis
3. Temporal order analysis
4. Communication pair analysis

Figure 5: Overall architecture of lprof

```java
1 class DataXceiver implements Runnable {
2   public void run() {
3     do { //handle one request per iteration
4         switch (readOpCode()) {
5             case WRITE_BLOCK: // a write request
6                 writeBlock(proto.getBlock(), ..); break;
7             case READ_BLOCK: // a read request
8                 readBlock(proto.getBlock(), ..); break;
9         } //proto.getBlock: deserialize the request
10     } while (!socket.isClosed());
11   }
12
13   void writeBlock(ExtendedBlock block..) {
14     LOG.info("Receiving block "+ block);
15     sender.writeBlock(block,..); //send to next DN
16     responder = new PacketResponder(block,..);
17     responder.start(); // create a thread that
18             // handles the acks
19   }
20
21   void PacketResponder handles the ack responses */
22   class PacketResponder implements Runnable {
23     public void run() {
24         LOG.info("Received block "+ block);
25         replyAck(upstream); //send an ack to upstream
26         LOG.info(myString + " terminating");
27     }
28 }
```

Figure 3: Code snippet from HDFS that handles write request.

Figure 4: Part of an HDFS log. Request identifiers are shown in bold. Note that the timestamp of each message is not shown.

3. Overview of lprof

In this Section, before describing lprof's design, we first discuss the challenges involved in stitching log messages together that were output when processing a single request. For example, consider how HDFS processes a write request as shown in Figure 3. On each datanode, a DataXceiver thread uses a while loop to process each incoming request. If the op-code is WRITE_BLOCK, then writeBlock() is invoked at line 7. At line 15, writeBlock() sends a replication request to the next downstream datanode. At line 16 - 17, a new thread associated with PacketResponder is created to receive the response from the downstream datanode so that it can send its response upstream. Hence, this code might output log messages as shown in Figure 4. These six log messages alone illustrate two challenges encountered:

1. The log messages produced when processing a single writeBlock request may come from multiple threads, and multiple requests may be processed concurrently. As a result, the log output messages from different requests will be intertwined.

2. The log messages do not contain an identifying sub-string that is unique to a request. For example, block ID “BP-9..9:blk_5..7” can be used to separate messages from different requests that do not operate on the same block, but cannot be used to separate the messages of the read and the first write request because they operate on the same block. Unfortunately, identifiers unique to a request rarely exist in real-world logs. In Section 7, we further discuss how lprof could be simplified if there were a unique request identifier in every log message.

To address these challenges lprof first uses static analysis to gather information from the code that will help map each log message to the processing of a specific request, and help establish an order on the log messages mapped to the request. In a second phase, lprof processes the logs using the information obtained from the static analysis phase; it does this as a MapReduce job.

We now briefly give a brief overview of lprof’s static analysis and log processing, depicted in Figure 5.

3.1 Static Analysis

lprof’s static analysis gathers information in four steps:

1. Parsing the log string format and variables obtains the signature of each log printing statement found in the code. An output string is composed of string constants
and variable values. It is represented by a regular expression (e.g., “Receiving block BP-(.*):blk-(.*)”), which is used during the log analysis phase to map a log message to a set of log points in the code that could have output the log message. We use the term log point in this paper to refer to a log printing statement in the code. This step also identifies the variables whose values are contained in the log message.

(2) Request identifier and request entry analysis are used to analyze the dataflow of the variables to determine which ones are modified. Those that are not modified are recognized as request identifiers. Request identifiers are used to separate messages from different requests; that is, two log messages with different request identifiers are guaranteed to belong to different requests. However, the converse is not true: two messages with the same identifier value may still belong to different requests (e.g., both of the “read” and the “write 1” requests in Figure 4 have the same the block ID).

Identifying request identifiers without domain expertise can be challenging. Consider “BP-9..9:blk_5..7_1032” in Figure 4 that might be considered as a potential identifier. This string contains the values of three variables as shown in Figure 6: poolID, blockID, and generationStamp. Only the substring containing poolID and blockID is suitable as a request identifier for writeBlock, because generationStamp can have different values while processing the same request (as exemplified by the “write 2” request in Figure 4).

To infer which log points belong to the processing of the same request, top-level methods are also identified by analyzing when identifiers are modified. We use the term top-level method to refer to the first method of any thread dedicated to the processing of a single type of request. For example, in Figure 3 writeBlock() and PacketResponder.run() are top-level methods, but DataXceiver.run() is not because it processes multiple types of requests. We say that method M is log point ρ’s top-level method if M is a top-level method and ρ is reachable from M.

If lprof can identify readBlock() and writeBlock() as being two top-level methods for different types of requests, it can separate messages printed by readBlock() from the ones printed by writeBlock() even if they have the same identifier value. We identify the top-level methods by processing each method in the call-graph in bottom-up order: if a method M modifies many variables that have been recognized as request identifiers in its callee M’, then M’ is recognized as a top-level method. The intuition behind this design is that programmers naturally log request identifiers to help debugging, and the modification of a frequently logged but rarely modified variable is likely not part of the processing of a specific request.

(3) Temporal order analysis is needed because there may not exist an ID unique to each request. For example, by inferring that line 26 is executed after line 24 in Figure 3, lprof can conclude that when two messages appear in the following order: “… terminating” and “Received block…”, they cannot be from the same request even if they have the same block ID.

(4) Communication pair analysis is used to identify threads that communicate with each other. Log messages output by two threads that communicate could potentially be from processing of the same request. Such communication could occur through cooperative threads in the same process, or via sockets or RPCs across the network.

3.2 Distributed Log Analysis

The log analysis phase attributes each log message to a request, which is implemented using a MapReduce job. The map function groups together all log messages that were output by the same thread while processing the same request. A log message is added to a group if (i) it has the same top-level method, (ii) the request identifiers do not conflict, and (iii) the corresponding log point matches the temporal sequence in the control flow.

The reduce function merges groups if they represent log messages that were output by different threads when processing the same request. Two groups are merged if (i) the two associated threads could communicate, and (ii) the request identifiers do not conflict.

4 Static Analysis

lprof’s static analysis works on Java bytecode. Each of the four steps in lprof’s static analysis is implemented as one analysis pass on the bytecode of the target system. We use the Chord static analysis framework [9]. For convenience, we explain lprof using examples in source code. All the information shown in the examples can be inferred from Java bytecode.

4.1 Parsing Log Printing Statements

This first step identifies every log point in the program. For each log point, lprof (i) generates a regular expres-
sion that matches the output log message, and (ii) identifies the variables whose values appear in the log output.

lprof identifies log points by searching for call instructions whose target method has the name fatal, error, warn, info, debug, or trace. This identifies all the logging calls if the system uses log4j [25] or SLF4J [37], two commonly used logging libraries that are used by the systems we evaluated.

To parse the format string of a log point into a regular expression, we use techniques similar to those used by two previous tools [42, 43]. We summarize the challenges we faced in implementing a log parser on real-world systems.

On the surface, parsing line 14 in Figure 3 into the regular expression “Receiving block (.*)”, where the wildcard matches to the value of block, is straightforward. However, identifying the variables whose values are output at the log point is more challenging. In Java, the object’s value is printed by calling its toString method. Figure 6 shows how the value of block is eventually printed. In this case, lprof has to parse out the individual fields because only poolID and blockID are request identifiers, whereas generationStamp is modified during request processing. To do this, lprof recursively traces the object’s toString method and the methods that manipulate StringBuilder objects until it reaches an object of a primitive type.

For the HDFS log point above, the regular expression identified by lprof will be:

“Receiving block (.*) blk_(\d+)_(.d+)”.

The three wildcard components will be mapped to block.poolID, block.block.blockID, and block.block.generationStamp, respectively.

lprof also needs to analyze the data-flow of any string object used at a log point. For example, mystring at line 26 in Figure 3 is a String object initialized earlier in the code. lprof analyzes its data-flow to identify the precise value of mystring.

Class inheritance and late binding in Java creates another challenge. For example, when a class and its super class both provide a toString method, which one gets invoked is resolved only at runtime depending on the actual type of the object. To address this, lprof analyzes both classes’ toString methods, and generates two regular expressions for the one log point. During log analysis, if both regular expressions match a log message, lprof will use the one with the more precise match, i.e., the regular expression with a longer constant pattern.

4.2 Identifying Request Identifiers

This step identifies (i) request identifiers and (ii) top-level methods. We implement the inter-procedural analysis as a summary-based analysis [35]. It analyzes one method at a time and stores the result as the summary of that method. The methods are analyzed in bottom-up order along the call-graph and when a call instruction is encountered, the summary of the target method is used. Not being summary-based would require lprof to store the intermediate representation of the entire program in memory, which would cause it to run out of memory.

Data-flow analysis for request identifiers: lprof infers request identifiers by analyzing the inter-procedural data-flow of the logged variables. For each method M, lprof assembles two sets of variables as its summary: (i) the request identifier candidate set (RIC), which contains the variables whose values are output to a log and not modified by M or its callees, and (ii) the modified variable set (MV) which contains the variables whose values are modified. For each method M, lprof first initializes both sets to be empty. It then analyzes each instruction in M. When it encounters a log point, the variables whose values are printed (as identified by the previous step) are added to the RIC set. If an instruction modifies a variable v, v is added to the MV set and removed from the RIC set. If the instruction is a call instruction, lprof first merges the RIC and MV sets of the target method into the corresponding sets of the current method, and then, for each variable v in the MV set, lprof removes it from the RIC set if it contains v.

As an example, consider the following code snippet from writeBlock():

```
1  LOG.info("Receiving "+ block);
2  block.setGenerationStamp(latest);
```

The setGenerationStamp() method modifies the generationStamp field in block. In bottom-up order, lprof first analyzes setGenerationStamp() and adds generationStamp to its MV set. Later when lprof analyzes writeBlock(), it removes generationStamp from its RIC set because generationStamp is in the MV set of setGenerationStamp().
Identifying top-level methods: the request identifier analysis stops at the root of the call-graph: either a thread entry method (i.e., run() in Java) or main(). However, a thread entry method might not be the entry of a service request. Consider the HDFS example shown in Figure 3. The DataXceiver thread uses a while loop to handle read and write requests. Therefore lprof needs to identify writeBlock() and readBlock() as the top-level methods instead of run().

lprof identifies top-level methods by observing the propagation of variables in the RIC set and uses the following heuristic when traversing the call-graph bottom-up: if, when moving from a method $M$ to its caller $M'$, many request identifier candidates are suddenly removed, then it is likely that $M$ is a top-level method. Specifically, lprof counts the number of times each request identifier candidate appears in a log point in each method and accumulates this counter along the call-graph bottom-up. (See Figure 7 for an example.) Whenever this count decreases from method $M$ to its caller $M'$, lprof concludes that $M$ is a top-level method. The intuition is that developers naturally include identifiers in their log printing statements, and modifications to these identifiers are likely outside the top-level method.

In Figure 7, both writeBlock() and readBlock() accumulate a large count of request identifiers, which drops to zero in run(). Therefore, lprof infers writeBlock() and readBlock() are the top-level methods instead of run(). Note that although the count of generationStamp decreases when the analysis moves from setGenerationStamp() to writeBlock(), it does not conclude setGenerationStamp() is a top-level method because the accumulated count of all request identifiers is still increasing from setGenerationStamp() to writeBlock().

4.3 Partial Order Among Log Points

In this step, lprof generates a Directed Acyclic Graph (DAG) for each top-level method (identified in the previous step) from the method’s call graph and control-flow graph (CFG). This DAG contains each log point reachable from the top-level method and is used to help attribute log messages to top-level methods.

It is not possible to statically infer the precise order in which instructions will execute. Therefore, lprof takes the liberty of applying a number of simplifications:

1. Only nodes that contain log printing statements are represented in the DAG.
2. All nodes involved in a strongly connected component (e.g., caused by loops) are folded into one

3. Similarly, if there is a strongly connected component due to recursive calls, then those nodes are also folded into one.
4. Unchecked exceptions are ignored, since they will terminate the execution. Checked exceptions are captured by the CFG and are included in the DAG.

As an example, Figure 8 shows the DAG generated from a code snippet. The asterisk (*) next to log 2 and log 3 indicates that these log points may appear 0 or more times. We do not maintain an ordering of the log points for nodes with multiple log points.

In practice, we found the DAG particularly useful in capturing the starting and ending log points of a request — it is a common practice for developers to print a message at the beginning of each request and/or right before the request terminates.

4.4 Thread Communication

In this step, lprof infers how threads communicate with one another. The output of this analysis is a tuple for each communication pair: (top-level method 1, top-level method 2, communication type, set of request identifier pairs), where one end of the communication is reachable from top-level method 1 and the other end is reachable from top-level method 2. “Communication type” is one of local, RPC, or socket, where “local” is used when two threads running in the same process communicate. A “request identifier pair” captures the transfer of request identifier values from the source to the destination; the pair identifies the variables containing the data values at source and destination.

Threads from the same process: lprof detects two types of local thread communications: (i) thread creation and (ii) shared memory reads and writes. Detecting thread creation is straightforward because Java has a well-defined thread creation mechanism. If an instruction r.start() is reachable from a top-level method, where r is an object of class C that extends the Thread class or implements the Runnable interface, and C.run() is another top-level method, then lprof has identified a communication pair. lprof also infers the data-flow of request identifiers, as they are mostly passed through the constructor of the target thread object. In addition to explicit
thread creation, if two instructions reachable from two
top-level methods (i) access a shared object, and (ii) one
of them reads and the other writes to the shared object,
then a communication pair is identified.

As an example, consider the HDFS code in
Figure 3. lprof generates the following tuple:
(writeBlock, PacketResponder.run, local, <DataX-
ceiver.block.poolID, PacketResponder.block.poolID>,
..), indicating that writeBlock() could communicate
with PacketResponder via local thread creation, and
poolID is the request identifier used on both ends for the
data value passed between the threads.

**Threads communicating across the network**: Pairing
threads that communicate via the network is more chal-
lenging. While Java provides standard socket read and
write APIs for network communication, if we naively pair
the read to the write on the same socket, we would effec-
tively end up connecting most of the top-level methods
together even though they do not communicate. Con-
sider the HDFS example shown in Figure 3. While
readBlock() and writeBlock() do not communicate
with each other, they share the same underlying socket.

Instead of pairing socket read and write, we observe
that the sender and receiver that actually communicate
both have to agree on the same protocol. Specifically,
whenever lprof finds a pair of invoke instructions whose
target methods are the serialization and deserialization
methods from the same class, respectively, the top-level
methods containing these two instructions are paired. De-
velopers often use third-party data-serialization libraries,
such as Google Protocol Buffers [15]. This further eases
lprof’s analysis since they provide standardized serial-
ization/deserialization APIs. Among the systems we
evaluated, Cassandra is the only one that does not use
Google Protocol Buffers, but implements its own serial-
ization library. For Cassandra, a simple annotation to pair
C.serialize() with C.deserialize() for any class
C is sufficient to correctly pair all of the communicating
top-level methods. lprof also parses the Google Proto-
ocol Buffer’s protocol annotation file to identify the RPC
pairs, where each RPC is explicitly declared.

**Improvements**: To improve the accuracy of “log stitch-
ing”, we add two refinements when pairing communi-
cation points. First, even when a thread does not con-
tain any log point (which means it does not contain any
top-level method), it will still be included in a commu-
nication pair if it communicates with a top-level method.
In this case, its run() method will be used as the com-
munication end point. The reason is that such a thread
could serve as a link connecting two communicating top-
level methods A and B. Not including the communication
pair would prevent lprof from grouping the log messages
from A and B.

---

**Top-level methods**:

<table>
<thead>
<tr>
<th>Method</th>
<th>DAG</th>
<th>Log Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DataXceiver.writeBlock, DAG#1, [id1,id2..])</td>
<td>entry → log1 → log4 → exit</td>
<td></td>
</tr>
<tr>
<td>(DataXceiver.readBlock, DAG#2, [id1,id2..])</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DAGs**:

1. Entry: log1
2. Exit: log4

**Regex**:

- <id1: block.poolID, id2: block.block.blockID>
- Comm. (writeBlock, PacketResponder.run, local, <id1, id1>, ..)

**Communication pairs**:

- (writeBlock, PacketResponder, socket, <id1, id1>, ..)

---

The second improvement is to infer the number of times
a top-level method can occur in a communication pair. For example, a communication pair “(M1, M2*, local, ..)”,
where M2 is followed by an asterisk, means that method
M1 could communicate with multiple instances of method M2 in the same request. The log analysis uses
this property to further decide whether it can stitch mes-
ages from multiple instances of M2 into the same re-
quest. The inference of such a property is straightforward:
if the communication point to M2 is within a loop
in M1’s CFG, then M2 could occur multiple times.

**4.5 Summary of Static Analysis**

The output of lprof’s static analysis is a file that contains
the log printing behavior of the system. Figure 9 shows
a snippet of the output file for HDFS. It consists of the
following four segments:

1. **Top-level methods**: a list of tuples with (i) the name
   of the top-level method, (ii) an index into the DAG
   representation of the log points, and (iii) a list of
   request identifiers;
2. **DAGs**: the DAG for each top-level method;
3. **Log point regex**: the regular expressions for each log
   point and the identifier for each wildcard;
4. **Communication pairs**: a list of tuples that identify
   the communication points along with the identifiers
   for the data being communicated.

To speedup log analysis, this output file also contains a
number of indexes, including: (i) an index of regular ex-
pressions (to speedup the matching of each log message
to its log point) and (ii) an index mapping log points to
top-level methods. This output file is sent to every ma-
chine in the cluster whose log is analyzed.

**5 Log Analysis**

The log analysis phase is implemented as a MapReduce
job to group together information from all the log mes-
sages printed by each request. The map and reduce func-
tions use a common data structure, called a request ac-
cumulator (RA), for gathering information related to the
same request. Each RA contains: (i) a vector of top-level methods that are grouped into this RA; (ii) the value of each request identifier; (iii) a vector of log point sequences, where each sequence comes from one top-level method; (iv) a list of nodes traversed, with the earliest and latest timestamp. The map and reduce functions will iteratively accumulate the information of log messages from the same request into the RAs. In the end, there will be one RA per request that contains the information summarized from all its log messages.

Map: Intra-thread Grouping

The map function is run on each node to process local log files. There is one map task per node, and all the map tasks run in parallel. Each map function scans the log file linearly. Each log message is parsed to identify its log point and the values of the request identifiers using regular expression matching. We also heuristically parse the timestamp associated with each message.

A parsed log message is added to an existing RA entry if and only if: (i) their top-level methods match, (ii) the identifier values do not conflict, and (iii) the log point matches the temporal sequence in the control flow as represented by the DAG. A new RA is created (and appropriately initialized) if the log message cannot be added to an existing RA. Therefore, each RA output by the map function contains exactly one top-level method.

Note that a sequence of log messages can be added to the same RA even when each contains the values of a different subset of request identifiers. Figure 10 shows an example. The 5 log messages in this figure can all be grouped into a same RA entry even though 4 of them contain the values of a subset of the request identifiers, and one does not contain the value of any request identifier but is captured using the DAG.

Combine and Reduce: Inter-thread Grouping

The combine function performs the same operation as the reduce function, but does so locally first. It combines two RAs into one if there exists a communication pair between the two top-level methods in these two RAs, and the request identifier values do not conflict. Moreover, as a heuristic, we do not merge RAs if the difference between their timestamps is larger than a user-configurable threshold. Such a heuristic is necessary because two RAs could have the same top-level methods and request identifiers, but represent the processing of different requests (i.e., two writeBlock operations on the same block). This

4Note that if a request identifier is not shared by all of the communicating top-level method, it cannot be used in the shuffle key because different communicating RAs might have different request identifier (e.g., one RA only has poolID while the other RA has blockID).

Figure 10: The grouping of five log messages where four print a subset of request identifier values.

Figure 11: The RAs that combine 9 log messages from 6 threads on 3 nodes belonging to a single write request in HDFS. Value is currently set to one minute, but should be adjusted depending on the networking environment. In an unstable network environment with frequent congestion this threshold should have a larger value.

After the combine function, lprof needs to assign a shuffle key to each RA, and all the RAs with the same shuffle key must be sent to the same reducer node over the network. Therefore the same shuffle key should be assigned to all of the RAs that need to be grouped together. We do this by considering communication pairs. At the end of the static analysis, if there is a communication pair connecting two top-level methods A and B, A and B are jointed together into a connected component (CC). We iteratively merge more top-level methods into this CC as long as they communicate with any of the top-level methods in this CC. In the end, all of the top-level methods in a CC could communicate, and their RAs are assigned with the same shuffle key.

However, this approach could lead to the assignment of only a small number of shuffle keys and thus a poor distribution in practice. Hence, we further implement two improvements to the shuffling process. First, if all of the communicating top-level methods have common request identifiers, the identifier values will be used to further differentiate shuffle keys. Secondly, if an RA cannot possibly communicate with any other RA through network communication, we do not further shuffle it, but instead we directly output the RA into the request database.

Finally, the reduce function applies the same method.
as the combine function. Figure 11 provides an example that shows how the RAs of log messages in the HDFS writeBlock request are grouped together. After the map function generates req.acc.1 and 2 on node 1, the combine function groups them into req.acc.3, because writeBlock() and PacketResponder.run() belong to the same communication pair, and their request identifier values match. Node 2 and node 3 run the map and combine functions in parallel, and generate req.acc.4 and 5. lprof assigns the same shuffle key to req.acc.3, req.acc.4, and req.acc.5. The reduce function further groups them into a final RA req.acc.6.

**Request Database and Visualization**

Information from each RA generated by the reduce function is stored into a database table. The database schema is shown in Figure 1. It contains the following fields: (i) request type, which is simply the top-level method with the earliest time stamp; (ii) starting and ending time stamps, which are the MAX and MIN in all the timestamps of each node; (iii) nodes traversed and the time stamps on each node, which are taken directly from the RA; (iv) log sequence ID (LID), which is a hash value of the log sequence vector field in the RA. For example, as shown in Figure 11, the vector of the log sequence of a writeBlock request is “[[LP1],[LP1],[LP1],[LP2,LP3],[LP2,LP3],[LP2,LP3]]”. In this vector, each element is a log sequence from a top-level method (e.g., “[LP1]” is from top-level method writeBlock() and “[LP2,LP3]” is from PacketResponder.run()). Note the LID captures the unique type and number of log messages, their order within a thread, as well as the number of threads. However, it does not preserve the timing order between threads. Therefore, in practice, there are not many unique log sequences; for example, in HDFS there are only 220 unique log sequences on 200 EC2 nodes running a variety of jobs for 24 hours. We also generate a separate table that maps each log sequence ID to the sequence of log points to enable source-level debugging. We use MongoDB [28] for our current prototype.

We built a web application to visualize lprof’s analysis result using the Highcharts [21] JavaScript charting library. We automatically visualize (i) requests’ latency over time; (ii) requests’ counts and their trend over time; and (iii) average latency per node. Figure 12 shows our latency-over-time visualization.

One challenge we encountered is that the number of requests is too large when visualizing their latencies. Therefore, when the number of requests in the query result is greater than a threshold, we perform downsampling and return a smaller number of requests. We used the largest triangle sampling algorithm [39], which first divides the entire time-series data into small slices, and in each slice it samples the three points that cover the largest area. To further hide the sampling latency, we pre-sample all the requests into different resolutions. Whenever the server receives a user query, it examines each pre-sampled resolution in parallel, and returns the highest resolution whose number of data points is below the threshold.

### 6 Evaluation

We answer four questions in evaluating lprof: (i) How much information can our static analysis extract from the target systems’ bytecode? (ii) How accurate is lprof in attributing log messages to requests? (iii) How effective is lprof in debugging real-world performance anomalies? (iv) How fast is lprof’s log analysis?

We evaluated lprof on four, off-the-shelf distributed systems: HDFS, Yarn, Cassandra, and HBase. We ran workloads on each system on a 200 EC2 node cluster for over 24 hours with the default logging verbosity level. Default verbosity is used to evaluate lprof in settings closest to the real-world. HDFS, Cassandra, and YARN use INFO as the default verbosity, and HBase uses DE-BUG. A timestamp is attached to each message using the default configuration in all of these systems.

For HDFS and Yarn, we used HiBench [22] to run a variety of MapReduce jobs, including both real-world applications (e.g., indexing, pagerank, classification and clustering) and synthetic applications (e.g., wordcount, sort, tersort). Together they processed 2.7 TB of data. For Cassandra and HBase, we used the YCSB [11] benchmark. In total, the four systems produced over 82 million log messages (See Table 1).

<table>
<thead>
<tr>
<th>System</th>
<th>LOC</th>
<th>workload</th>
<th># of msg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS-2.0.2</td>
<td>142K</td>
<td>HiBench</td>
<td>1,760,926</td>
</tr>
<tr>
<td>Yarn-2.0.2</td>
<td>101K</td>
<td>HiBench</td>
<td>79,840,856</td>
</tr>
<tr>
<td>Cassandra-2.1.0</td>
<td>210K</td>
<td>YCSB</td>
<td>394,492</td>
</tr>
<tr>
<td>HBase-0.94.18</td>
<td>302K</td>
<td>YCSB</td>
<td>695,006</td>
</tr>
</tbody>
</table>

Table 1: The systems and workload we used in our evaluation, along with the number of log messages generated.
Threads 90.6% Incomplete 21 193 92 44 8 97.0% 4.6%
Top-lev. meth. ≥ 1 id. per DAG*

<table>
<thead>
<tr>
<th>System</th>
<th>Threads</th>
<th>≥ 1 log</th>
<th>Top-lev.</th>
<th>Log points</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>44</td>
<td>95%</td>
<td>167</td>
<td>79%</td>
</tr>
<tr>
<td>Yarn</td>
<td>45</td>
<td>73%</td>
<td>79</td>
<td>66%</td>
</tr>
<tr>
<td>Cass.</td>
<td>92</td>
<td>74%</td>
<td>74</td>
<td>45%</td>
</tr>
<tr>
<td>HBBase</td>
<td>85</td>
<td>80%</td>
<td>193</td>
<td>74%</td>
</tr>
<tr>
<td>Average</td>
<td>67</td>
<td>81%</td>
<td>129</td>
<td>66%</td>
</tr>
</tbody>
</table>

Table 2: Static analysis result. *: in these two columns we only count the log points that are under the default verbosity level and not printed in exception handler — indicating they are printed by default under normal conditions.

<table>
<thead>
<tr>
<th>System</th>
<th>Correct</th>
<th>Incomplete</th>
<th>Incorrect</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>97.0%</td>
<td>0.1%</td>
<td>0.3%</td>
<td>2.6%</td>
</tr>
<tr>
<td>Yarn</td>
<td>79.6%</td>
<td>19.2%</td>
<td>0.0%</td>
<td>1.2%</td>
</tr>
<tr>
<td>Cassandra</td>
<td>95.3%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>4.6%</td>
</tr>
<tr>
<td>HBBase</td>
<td>90.6%</td>
<td>2.5%</td>
<td>3.5%</td>
<td>3.4%</td>
</tr>
<tr>
<td>Average</td>
<td>90.4%</td>
<td>5.7%</td>
<td>1.0%</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

Table 3: The accuracy of attributing log messages to requests.

### 6.1 Static Analysis Results

Table 2 shows the results of lprof’s static analysis. On average, 81% of the statically inferred threads contain at least one log point that would print under normal conditions, and there are an average of 20 such log points reachable from the top-level methods inferred from the threads that contain at least one log point. This suggests that logging is prevalent. In addition, 66% of the log points contain at least one request identifier, which can be used to separate log messages from different requests. This also suggests that lprof has to rely on the generated DAG to group the remaining 34% log points. lprof’s static analysis takes less than 2 minutes to run and 868 MB of memory for each system.

### 6.2 Request Attribution Accuracy

With 82 million log messages, we obviously could not manually verify whether lprof correctly attributed each log message to the right request. Instead, we manually verified each of the log sequence IDs (LID) generated by lprof. Recall from Section 5 that the LID captures the number and the type of the log points of a request, and the partial orders of those within each thread (but it ignores the thread orders, identifier values, and nodes’ IPs). Only 784 different LIDs are extracted out of a total of 62 million request instances. We manually examined the log points of each LID and the associated source code to understand its semantics. The manual examination took four authors one week of time.

Table 3 shows lprof’s request attribution accuracy. A log sequence A is considered correct if and only if (i) all its log points indeed belong to this request, and (ii) there is no other log sequence B that should have been merged with A. All of the log messages belonging to a correct log sequence are classified as “correct”. If A and B should have been merged but were not then the messages in both A and B are classified as “incomplete”. If a log message in A does not belong to A then all the messages in A are classified as “incorrect”. The “failed” column counts the log messages that were not attributed to any request.

Overall, 90.4% of the log messages are attributed to the correct requests.

5.7% of the log messages are in the “incomplete” category. In particular, 19.2% of the messages in Yarn were mistakenly separated because of only 2 unique log points that print the messages in the following pattern: “Starting resource-monitoring for container_1398” and “Memory usage of container-id container_1398...”. lprof failed to group them because the container ID was first passed into an array after the first log point and then read from the array when the second message was printed. lprof’s conservative data-flow analysis failed to track the complicated data-flow and inferred that the container ID was modified between the first and the second log points, thus attributing them into separate top-level methods. A similar programming pattern was also the cause of “incomplete” log messages for HB and HDFS. Cassandra’s 0.1% “incomplete” log messages were caused by a few slow requests with consecutive log messages whose intervals were over one minute.

1.0% of the log messages are attributed to the wrong requests, primarily because they do not have identifiers and they are output in a loop so that the DAG groups them all together. This could potentially be addressed with a more accurate path-sensitive static analysis.

3.0% of the log messages were not attributed to any request because they could not be parsed. We manually examined these messages and the source code, and found that in these cases, developers often use complicated data-flow and control-flow to construct a message. However, these messages are mostly generated in the start-up or shut-down phase of the systems and thus likely do not affect the quality of the performance analysis.

Inaccuracy in lprof’s request attribution could affect users as follows: since the “incomplete requests” are caused by two log sequences A and B that should have been merged but were not, lprof would over-count the number of requests. For the same reason, timing information separately obtained from A and B would be underestimations of the actual latency. The “incorrect requests” are the opposite; because they should have been split into separate requests, “incorrect requests” would cause lprof to under-count the number of requests yet overestimate the latencies. Note that administrators should quickly realize the “incorrect requests” because lprof provides the sequence of log messages along with their source code information. The information about the “failed” messages,
6.3 Real-world Performance Anomalies

Figure 13: The cumulative distribution function on the number of log messages per unique request. For Cassandra, the number of nodes each streaming session traverses greatly, therefore the number of log messages in each streaming session request also varies greatly (it eventually reaches 100% with 1060 log messages, which is not shown in the figure).

Table 4: Evaluation of 23 real-world performance anomalies. However, will be lost.

Number of messages per request: Figure 13 shows the cumulative distribution function on the number of messages printed by each unique request, i.e., the one with the same log sequence ID. In each system, over 44% of the request types, when being processed, print more than one messages. Most of the requests printing only one message are system’s internal maintenance operations.

Table 5: The most useful analyses on real-world performance anomalies. The percentage is over the 15 anomalies where lprof is helpful. An anomaly may need more than one queries to detect and diagnose, so the sum is greater than 100%.

Overall, lprof is helpful in detecting and diagnosing 65% of the real-world failures we considered. Next, we discuss when and why lprof is useful or not-so-useful.

Table 5 shows the features of lprof that are helpful in debugging real-world performance anomalies we considered. The “request count” analysis is useful in 73% of the cases. These cases, the performance problems are caused by an unusually large number of requests, either external ones submitted by users or internal operations. For example, the second performance anomaly we discussed in Section 2 belongs to this category, where the number of verifyBlock operations is suspiciously large. In these cases, lprof can show the large request number and pinpoint the particular offending requests.

Another useful feature of lprof is its capability to associate a request’s log sequence to the source code. This can significantly reduce developers’ efforts in searching for the root cause. In particular, among the cases where lprof is helpful, 67% of the bugs that introduced inefficiencies were in the same method that contained one of the log points involved in the anomalous log sequence.

lprof’s capability of analyzing the latency of requests is useful in identifying the particular request that is slow. The visualization of request latency is particularly useful in analyzing performance creep. For example, the anomaly to HDFS’s write requests discussed in Section 2 can result in performance creep if not fixed. In addition, lprof can further separate the requests of the same type by their different LIDs which corresponds to different execution paths. For example, in an HBase performance anomaly [19], there was a significant slow-down in 1% of the read requests because they triggered a buggy code path. lprof can separate these anomalous reads from other normal ones.

In practice, the user might not identify the root cause in her first attempt, but instead will have to go through a sequence of hypotheses validations. The variety of performance information that can be SQL-queried makes lprof a particularly useful debugging tool. For example, an HBase bug caused an unbalanced workload — a few region servers were serving the vast majority of the requests while others were idle [18]. The root cause is clearly visible if the administrator examines the number of requests.
per node. However, she will likely first notice the request being slow (via a request latency query), isolate particularly slow requests, before realize the root cause.

In the cases where lprof was not helpful, most (75%) were because the anomalous requests did not print any log messages. For example, a pair of unnecessary memory serialization and deserialization in Cassandra would not show up in the log. While theoretically one can add log messages to the start and end of these operations, in practice, this may not be realistic as the additional logging may introduce undesirable slowdown. For example, the serialization operation in Cassandra is an in-memory operation that is executed on every network communication, and adding log messages to it will likely introduce slowdown. In another case, the anomalous requests would only print one log message, so lprof cannot extract latency information by comparing differences between multiple timestamps. Finally, there was one case where the checksum verification in HBase was redundant because it was already verified by the underlying HDFS. Both verifications from HBase and HDFS were logged, but lprof cannot identify the redundancy because it does not correlate logs across different applications.

If verbose logging had been enabled, lprof would have been able to detect an additional 8.6% of the real-world performance anomalies that we considered since the offending requests print log messages under the most verbose level. However, enabling verbose logging will likely introduce significant performance overhead.

### 6.4 Time and Space Evaluation

The map and combine functions ran on each EC2 node, and the reduce function ran on a single server with 24 2.2GHz Intel Xeon cores and 32 GB of RAM.

Figure 14 shows the size of intermediate result. On average, after map and combine, the intermediate result size is only 7.3% of the size of the raw log. This is the size of data that has to be shuffled over the network for the reduce function. After reduce, the final output size is 4.8% of the size of the raw log.

Table 6 shows the time and memory used by lprof’s log analysis. lprof’s map and combine functions finish in less than 6 minutes for every system exception for Yarn, which takes 14 minutes. Over 80% of the time is spent on log parsing. We observe that when a message can match multiple regular expressions, it takes much more time than those that match uniquely. The memory footprint for map and combine is less than 3.3GB in all cases.

The reduce function takes no more than 21 seconds for HDFS, Cassandra, and HBase, but currently takes 19 minutes for Yarn. It also uses 7.2GB of memory. Currently, our MapReduce jobs are implemented in Python using Hadoop’s streaming mode, which may be the source of the inefficiency. (Profiling Yarn’s reduce function shows that over half of the time is spent in data structure initializations.) Note that we run the reduce job on a single node using a single thread. The reducer could and should be parallelized in real-world usage.

### 7 Limitations and Discussions

We outline the limitations of lprof through a series of questions. We also discuss how lprof could be extended under different scenarios.

1. **What are the logging practices that make lprof most effective?** The output of lprof, and thus its usefulness, is only as good as the logs output by the system. In particular, the following properties will help lprof to be most effective: (i) attached timestamps from a reasonably synchronized clock; (ii) output messages in those requests that need profiling (multiple messages are needed to enable latency related analysis); (iii) the existence of a reasonably distinctive request identifier, and (iv) not printing the same message pattern in multiple program locations.

Note that these properties not only will help lprof, but also are useful for manual debugging. lprof naturally leverages such existing best-practices. Furthermore, lprof’s static analysis can be used to suggest how to improve logging. It identifies which threads do not contain any log printing statements. These are candidates for adding log printing statements. lprof can also infer the request identifiers for developers to log.

2. **Can lprof be extended to other programming languages?** Our implementation relies on Java bytecode and hence is restricted to Java programs (or other languages that use Java bytecode, such as Scala). Similar analysis can be done on LLVM bytecode [24], but this would most
likely require access to the C/C++ source code so it can be compiled to LLVM bytecode.

(3) How scalable is lprof? While the map phase is executed in parallel on each node that stores the raw log, the reduce phase may not be evenly distributed. This is because all of the RAs that contain top-level methods that might communicate with each other need to be shuffled to the same reducer. This can result in unbalanced load. For example, in Yarn, 75% of the log messages are printed by one log point during the heartbeat process, and their RAs have to be shuffled to the same reducer node. This node becomes the bottleneck even if there are other idle reducer nodes. How to further balance the workload is part of future work.

(4) How does lprof change if a unique per-request ID exist? If such an ID exists in every log message, then there would be no need to infer the request identifier. The log string format parsing could also be simplified since now our log parser only needs to match a message to a log printing statement, but does not need to precisely bind the values to variables. However, the other components are still needed. DAG and communication pairs are still needed to infer the order dependency between different log messages, especially if we want to perform per-thread performance debugging. The MapReduce log analysis is still needed. If such an ID exists, then the accuracy of lprof will increase significantly, and we can better distribute the workload in the reduce function by using this ID as part of the shuffle key.

(5) What happens when the code changes? This requires lprof to perform static analysis on the new version. The new model produced by the static analysis should be sent to each node along with the new version of the system.

8 Related Work

Using machine learning for log analysis: Several tools apply machine learning on log files to detect anomalies [4, 29, 42]. Xu et al., [42] also analyzes the log printing statements in the source code to parse the log. lprof is different and complementary to these techniques. First, these tools target anomaly detection and do not identify request flows as lprof does. Analyzing request flows is useful for numerous applications, including profiling, and understanding system behavior. Moreover, the different goals lead to different techniques being used in our design. Finally, these machine learning techniques can be applied to lprof’s request database to detect anomalies on a per-request, instead of per-log-entry, basis.

Semi-automatic log analysis: SALSA [40] and Mochi [41] also identify request flows from logs produced by Hadoop. However, unlike lprof, their models are manually generated. By examining the code and logs of HDFS, they identify the key log messages that mark the start and the end of a request, and they identify request identifiers, such as block ID. The Mystery Machine [10] extracts per-request performance information from the log files of Facebook’s production systems, and it can correlate log messages across different layers in the software stack to infer the performance critical path. To do this, it requires developers to attach unique request identifiers to each log message. Commercial tools like VMWare LogInsight [26] and Splunk [38] index the logs, but require users to perform keyword-based searches.

Single thread log analysis: SherLog [43] analyzes the source code and a sequence of error messages to reconstruct the partial execution paths that print the log sequence. Since it is designed to debug functional bugs in single-threaded execution, it uses precise but heavyweight static analysis to infer the precise execution path. In contrast, lprof extracts less-precise information for each request, but it analyzes all the log outputs from all the requests of the entire distributed system.

Instrumentation-based profiling: Instrumentation-based profilers have been widely used for performance debugging [6, 8, 16, 17, 23, 32, 34]. Many, including Project 5 [1], MagPie [3], X-Trace [14], and Dapper [36], just to name a few, are capable of analyzing request flows by instrumenting network communication, and they can profile the entire software stack instead of just a single layer of service. G2 [17] further models all the events into an execution graph that can be analyzed using LINQ queries and user-provided programs. In comparison, lprof is non-intrusive. It also provides source-level profiling information. However, it cannot provide any information if requests do not output log messages.

9 Conclusions

This paper presented lprof, which is, to the best of our knowledge, the first non-intrusive request flow profiler for distributed services. lprof is able to stitch together the dispersed and intertwined log messages and associate them to specific requests based on the information from off-line static analysis on the system’s code. Our evaluation shows that lprof can accurately attribute 90% of the log messages from widely-used, production-quality distributed systems, and is helpful in debugging 65% of the sampled real-world performance anomalies.

Acknowledgements

We greatly appreciate the anonymous reviewers and our shepherd, Ed Nightingale, for their insightful feedback. This research is supported by NSERC Discovery grant, NetApp Faculty Fellowship, and Connaught New Researcher Award.
References


